FIT2099 Assignment 1 Design Rationale by 1337Gamers

Food Class

All potential food objects are created with their own classes but they derived from another class in engine to ensure they are in the game when the game is running.

CorpseProtoceratops, CarnivoreFood and HerbivoreFood is chosen to be a subclass of Item which will allow it to be purchased or sold in shops and will also have properties that allow it to be eaten by dinosaurs. However, Grass and Tree will be a subclass of Ground as this allows these objects to be a part of the terrain generated for the map. These two objects when eaten will be turned into dirt by the method setGround in the Location class.
Due to number of different foods, edibleItem, the Food interface and Plant are introduced to ease the workload between them for CorpseProtoceratops, Egg, CarnivoreFood, HerbivoreFood, and Plant for Grass and Tree. It will make the coding a lot easier and achieving abstraction between item and trying to categorise all these items to their new respective classes. The Food interface was introduced because not all items are food, plant is also a food and main reason for this is java only allowed one inheritance at a time but at the same time it achieved **dependency inversion** which states high level module should not depend on low level modules. Low level can be seen as edible items and plant class are implementing food interface to depends on high level module and and **Program to interface** principles which allows all class implementing it will only be used and not only that **Program to interface** helps all class implementing it, hide all implementation with only essential methods is used, other method and data are hidden as well. This adheres to **Encapsulation** principle. Moreover, polymorphism can also be easily achieved if coding is done right. **Open Closed** design principles is adhere here because dinosaur food and plant can extend a parent class to have their own property and will leave the source code unharm and will also makes coding much easier and maintainable..

Inventory

The shop and player will have their own inventory. The inventory here is defined as a collection that accepts Item object. Thus, Shop and Player will have zero or many items because Shop and Player can have 0 items or more in a collection. However, all actors already have an inventory in their implementation This was probably done as either the shop or the actors can only have 1 inventory and not multiple because it will be irrelevant to create a shop class that could potentially be called multiple times by a single entity. **Open Closed** design principles is adhere here because inventory is not required at all since actor has already implemented on it owns

Dinosaur Tagging

Tag will be a subclass of Item so that it is capable of being sold in shops and carried in an inventory. It will associated with the TagDinosaur action to provide a new functionality that allows player to tag a dinosaur. Player will have the option to call TagDinosaur through the menu when nearby to a dinosaur. The class will associate that tag with a unique ID that will also be stored on the dinosaur instance and will then store the tag back in the inventory of the player. The dinosaur will then be deleted through a method in the TagDinosaur class but at the same time the tag player has will no longer be able to tag anyone and should the player choose that tag to sell at the shop.
TagDinosaur action will be an action that is hidden until player has acquired at least one tag in their inventory and player is next to a dinosaur.
The design of Dinosaur Tagging that needs to incorporate more than one class is adhere to **Single Class Responsibility,** TagDinosaur and Tag is used to ensure that all these classes are responsible for their own properties and it would have breached this principle if the player was allowed to just call Tag class.

Shop

The shop will be the floor tiles that are in the building at the top left hand corner of the map. When the player is placed on a shopfloor tile, an action will be available through the menu to see their inventory or see shops inventory. If they select their own inventory, their own inventory will be displayed as part of the menu and the player can select an item in the same way they select a direction to walk. Once an item is selected depending on whether it is the shops or the players inventory, it will be bought or sold through the itemTransaction class. The item will be sent into the class and its price will be withdrawn or deposited into the players balance attribute. If the player is buying an item, the item will then be given to the player through additemtoinventory method. If it's being sold, it will then be left with the itemtransaction class. Transaction Type is added to reduce the amount of literal and code duplication for dependency.

A new class is created to handle transactions as it is a poor design choice to let either the shop or the player classes have control over the transaction. Furthermore, creating a class called ItemTransactions will adhere to the design principle. **Reduce amount of literals as much as possible** where enums replace literals of its own and **Don’t repeat yourself** for coding the same implementation for sales and purchase.
Encapsulation which states that implementation is hidden to ensure data hiding and this principles apply in this situation because it ensures that Inventory of both classes are not modified in any manner and Player class can only wish to add or remove items to shop. We also covered the **Delegations** principle as we ensured that Player and Shop do not need to work hard trying to change their inventory and cash but instead rely on one class which makes both Player and Shop only responsible of their own work.

Tree Growing

When Gamemap called the tick for each location (height and width loop) on the map each Ground subclass will first check all 4 boxes in direct contact with the current box to see if any are trees and then calculate the chance that the current square will become a tree. Then call the setGround method in Location class and change the square to a tree. The design principle adhered to here is **Single class responsibility**, where tree does not need to be concerned about its neighbour's whereabouts but map will tick all locations of GameMap. We can then utilise the tick method for all locations and just check if it's not a tree and neighbour has at least one tree around it, then it should have a chance to grow a tree. Thus, a new location which named TropicalLocation and TropicalMap is created to ensure the requirements are met and TropicalLocation and TropicalGameMap is not going to be a standalone module but extends from Location and GameMap class from engine to adheres to **Open Closed** design principles.

Grass Growing

Dirt as well as Ground will be set the tick method, that every turn it has the .5% chance to turn into grass. This can be done with changing the ground type into Grass instance which is subclass of Ground and use a method in Location class named setGround() method. However, this particular behavior is not required to be implemented because abstract class Ground will have its tick method that will be running every turn so setGround() method can be implemented in tick() method and random chances are calculated as well within the scope of method. **Single class responsibility** is adhered to here because all dirt is responsible for transforming itself to Grass and Grass does not need to do anything but wait till its object is created.

Dinosaur Tagging

Tag will be a subclass of Item so that it is capable of being sold in shops and carried in an inventory. It will associated with the TagDinosaur action to provide a new functionality that allows player to tag a dinosaur. Players have the option to call TagDinosaur through a menu option when nearby to a dinosaur actor. The class will associate that tag with a unique ID that will also be stored on the dinosaur instance and will then store the tag back in the inventory of the player. Should the player choose that tag to sell at the shop, the dinosaur will then be deleted through a method in the ItemTransaction class.
TagDinosaur action will be an action that is hidden until player has acquired at least one tag in their inventory and player is next to a dinosaur. This complicated implementation is done via checking if all actor near dinosaur has skill named Tag skill which allows dinosaur and tag dinosaur finding out who has tag skill which adheres to **Delegation** principles.
The design of LookforFood that needs to incorporate more than one class is adhere to **Single Class Responsibility,** TagDinosaur and Tag is used to ensure that all these classes are responsible for their own properties and it would have breached this principles if the player was allowed to just call Tag class.

Dinosaur Growing

Changed from assignment 1, the original growDinosaur class that we had was removed as all Baby and Adult classes were removed as there was too much unnecessary repeated code between the separate classes that were almost identical except for a few different actions (**Don't repeat yourself**). The growing was therefore placed into the new Dinosaur super class that we have over both the carnivore and herbivore dinosaur super classes which, in-turn, house the subclasses of all dinosaurs. The idea with this was that a dinosaur when it reaches a certain age, which is stored as a variable and iterated every playturn, the growDinosaur method will be called by "DinosaurBehaviour" to grow the dinosaur from a baby to an adult merely changing a new boolean called adult to true and updating the displayChar to the capital letter of the associated lowercase letter allocated to the dinosaur. This reduces the **dependency** between classes as if GrowDinosaur had been changed, it would have affected all dinosaur classes.

Hatch class was also changed in the same way. Initially, a LayEgg class was also added to create the new egg class and drop it at the feet of the dinosaur that laid it but for similar reasons to GrowDinosaur being changed both Hatch and LayEgg classes became methods. LayEgg was placed into the Dinosaur superclass and Hatch was placed into the Egg superclass and is still called by the tick method. As specified, this was done to reduce the **dependency** between classes as changes to LayEgg or Hatch would result in all Egg and all Dinosaur subclasses needing changes and for simplicity. If the extra class was not needed and could be a method it may as well be. These single classes being made into methods do not mean that we repeat ourselves either as they are placed into the superclass of all dinosaurs and are therefore usable by all dinosaurs.

Attack Protoceratops

Veloceratops will attack Protoceratop on when protoceratop near to it. Since attacking Protoceratop is a characteristic of Veloceratops to ensure it can eat meat, AttackProtoceratops is chosen to be a behavior subclass and will call AttackAction when Veloceratops is next to Protoceratop. AttackAction will takes in instance of Protoceratop and reduce its health. This design adheres to; **Program to interface** principles as AttackProtoceratops is extending the behavior interface which can help maintainability and improve flexibility of programming; **Delegation** principles because AttackProtoceratops behavior is not doing all things alone, it will also delegate AttackActions to helps reduce the health of instance protoceratops and **Single Class Responsibility,** AttackProtoceratops and AttackAction is used to ensure that all these classes are responsible for their own properties.

DeathAction

When dinosaur has 0 food level, it will call the DeathAction.This can be implemented with just calling Death action when the dinosaur's current food level is 0. **Delegation** principle apply here where the dinosaur does not kill itself but delegates the task of removing itself from map to an action. **Class manages its own properties** is applied, the dinosaur figures out whether he died of hunger by the method returnCurrentFoodLevel, this is applied in the DinosaurBehaviour class.

LookforFood

LookforFood is a subclass of behavior because it is a characteristic of dinosaur that will exhibit everytime it becomes hungry. LookforFood class will find the nearest food source location for dinosaur by iterating the code with MoveActorAction from engines to move the dinosaur towards a location that has food is placed on Ground, similar to the follow behaviour already implemented.
When the dinosaur is on the same square as the food item, LookforFood behavior will use the Eating action to eat the food for the respective instance of actor and the actor's food level will be updated.

Eating is created as a subclass of Action to allow dinosaur to Eat their food source when they are hungry. It is chosen to be a subclass of Actions because it is a type of Action that dinosaur will perform and call this class when they meet certain conditions. It will also allow behavior class to perform Eating actions when appropriate and not before their turn is over. (Eating must take a whole turn)

Protoceratops is a Herbivore dinosaur and will only eat trees, grass and herbivore food. These all implement the Food interface. Once HerbivoreFood is consumed, it will be deleted by the eating class using the eaten method present under the Food interface and then update the food level of Protoceratops that called the class. However, Tree and Grass will be set to dirt by the setGround method instead and Eating will then update the food level of Protoceratops based on the returned food value from the item or ground.

Veloceratops is a Carnivore dinosaur and will only consume CorpseProtoceratops, CarnivoreFood and Egg. When a Velociraptor has access to any food on the ground at their location, the Eating function will be called by the LookForFood class that will take the food as input as well as the dinosaur. The item will be deleted and the dinosaurs hunger value updated. they will vanish from the map and then update the food level of Veloceratops. Protoceratops cannot be eaten immediately, it must first be killed in which time it will call a new ProtoceratopsCorpse item and then delete itself, leaving the item at the location at which it died.

The design of LookforFood that needs to incorporate more than one class adheres to the **Single Class Responsibility.** LookforFood, Eating and MoveActorAction is used to ensure that all these classes are responsible for their own properties.The principle of **Delegation** also applies here because LookforFood class will delegate two responsibilities to two classes, calling MoveActorAction to move dinosaur toward its food source and Eating to perform eating action by dinosaur. Using Food interface to helps code for all item that is considered an edible item will help ease the coding and **Program to interface** principles which allows all class implementing it will only be used and not only that **Program to interface** helps all class implementing it, hide all implementation with only essential methods is used, other method and data are hidden as well. This adheres to **Encapsulation** principle. Moreover, polymorphism can also be easily achieved if coding is done right.

Movement

For the third assignment movement had to be overhauled as the current movement system didn't allow for Pteranodon to move two spaces on a single turn. To be able to make this work, a new class had to be added called GameCharacter, this is the single subclass of actor and all other actor classes are subclasses of GameCharacter. This technically allows additions to the actor class without making direct changes to the engine.
This was required so that the method returnExits could be added. This function looks at the amount of moves that each actor is allowed and runs a for loop to find the initial set of exits, then for the amount of moves, adds the exits of the exits in the current list to that list. This gives a list of all possible exits with the specified amount of moves around the actor.
This list can then be used in the same fashion in wander and lookforfood behaviours, to read through it and find the most reasonable place to move to, meaning that the Pteranodon can move either 1 or 2 spaces.
This implementation is better than such implementations as creating a separate floating class to house only that method as that would create an unnecessary dependency.
It is also superior to adding the return exits for loops to every single behaviour that requires them as this would be a lot of repeated code that would need to be updated in every behaviour separately whenever a change needed to be made.

Since we now also have different kinds of dinosaurs that require different kinds of movements, we added in an enum called MovementType which houses; SWIM, WALK and FLY. This is utilised by any of the behaviours that require movement as the Ground method canActorEnter can be overwritten in all of our ground classes to only return true if an actor is of the appropriate movement type. As this was already a part of the engine there is no extra dependencies created or repeated code.

Jurrasic World

World have run() method that will run processActorTurn(Actor actor) method in a while loop which depends whether player is still in map. Therefore, game outcome which consist of wins, lose and quit can be done by just removing actor or modify stillRunning() method so that it will consider when a captive T-rex has grown to adult, it should help to terminate is run() method and will not interrupt post instruction after a while loop so that it can still achieve game outcome while terminating program safely and successfully. Thus, a new class called JurrasicWorld which will extend World class and modify stillRunning() and run() method to ensure Jurrasic World class will be able to terminate safely and successfully. run() method will print extra string win or lose and stillRunning() needs to consider when a captive TRex grows, player wins.**Open Closed** design principles is adhere here because stillrunning() method will just need to be modify slightly to consider captive TRex grows and game outcome, and this can be done via extends on World class. **Class manages it owns property** because New World class responsibility is to keep the flow of the game going and this involves initialising games requirement, processing turn and ending game.
.

AI Behavior

AI behavior is means artificial intelligence behavior. This class is a factory of behavior for all character which are not player. AI behavior is a behavior that determine what AI will do and it is generic class for all AI interaction. This class can be extended for different behavior of actor. This class resembles iterable class where it will takes in different abstract data type and act for different data. **DRY principle** is also adhered here AIBehavior can be used for all actor and all they needs to do is add new behavior to it and it will do action factory for actors.

Land & Water

Land and Water is a form of abstraction of a ground class because most ground in the system are only land or water. Land is an abstract class for plant class and dirt class because both class are tasked with responsibility to ensure land creature can walk on it while Water class is not an abstract class because instantiation of it will be needed and Water is task to allow all water creature to walk on it. **Open Closed** is adhere here because land and water class can easily extends Ground class and make use of all the ground class property while still remain unharmed on source code. Furthermore, **DRY principle** is also adhered here because both classes are form of abstraction and are tasked to ensure all its base class have the same property as parent class without having repeating code.

Reed

Reed is a class that extends water because it is part of water where only water creature can walk on it. This adhere to **Open Closed** is adhere here because Reed class can easily extends Water class and make use of all the Water class property while still remain unharmed on source code for ground class.

Reed Grows

When Gamemap called the tick for each location (height and width loop) on the map each Ground subclass will first check all 4 boxes in direct contact with the current box to see if any are reeds and then calculate the chance that the current square will become a reed. Then call the setGround method in Location class and change the square to a Reed. The design principle adhered to here is **Single class responsibility**, where Reed does not need to be concerned about its neighbour's whereabouts but map will tick all locations of GameMap. We can then utilise the tick method for all locations and just check if it's not a reed and neighbour has at least one reed around it, then it should have a chance to grow a reed. Thus, a new location which named TropicalLocation and TropicalMap is created to ensure the requirements are met and TropicalLocation and TropicalGameMap is not going to be a standalone module but extends from Location and GameMap class from engine to adheres to **Open Closed** design principles.

Generating Fish

Reed as well as Ground will be set the tick method, that every turn it has the .10% chance to generate a Fish. This can be done with adding Fish instance into current location if there is no actor on it and use a method in Location class named addActor() method. However, this particular behavior is not required to be implemented because abstract class Ground will have its tick method that will be running every turn so addActor() method can be implemented in tick() method and random chances are calculated as well within the scope of method. **Single class responsibility** is adhered to here because all Reed is responsible for add Fish and Fish does not need to do anything but wait till its object is created.

Boat

Boat in the requirement is not specify what behavior it has, therefore assumption is made that boat will be a purchasable item in shop and it will not be allowed to be dropped because that could affect functionality of game. Boat in this game has two roles, first player can tag dinosaur and feed dinosaur in case fish does not appear too frequently.
The boat add skill which is swim skill which allow player to enter both water and land location.**Open closed** design principles is adhered here developer can just create a new class which extends purchasable item without modifying skill class and purchasable item class while still performing it task allowing player to move on water.

Bonus

Criminal

Criminal in this game will extends from actor base class to add functionality to it so that that actor will behave as a thug who wants to kill everyone on sight. They can turn into agent when they kill enough agent, they will then gain Teleport and Shooting ability and this makes the game more challenging. **Open closed** principle is adhered here because actor is open to extension and can lead actor class unharm. **Classes responsible for thier own property** is adhered here because criminal turn into agent by itself when kill enough actor and gain new ability which is teleport and shooting.

Neo

Neo is a new playable character which allows player to shoot other actor, The game is so tough that adding red pill to allow neo to finally become 'The one' and at that moment, neo is not killable at all and will instead does the killing more effectively but player should ensure that main objective of TRex Growing is remain in order to win the game which also ultimately balance the game with some advantages and disadvantages. **Open closed** principle is adhered here because actor is open to extension and can lead actor class unharm. **Classes responsible for thier own property** is adhered here because Neo turn into The One by itself when it have red pill.

Gun

Gun is a weapon item which allows one actor to hurt another actor and can be implemented by extending weapon item class. **Open closed** principle is adhered here because weapon item is open to extension and can lead weapon item class unharm when gun is introduced as weapon.

Red pill

Red pill is a purchasable item which allows Neo to turn into The One and can be done by extending Purchasable item. **Open closed** principle is adhered here because Purchasable item is open to extension and can lead weapon item class unharm when Red is introduced Purchasable item.

Shooting Behavior

Both neo and criminal have chance to shoot all target but can only shoot vertically or horizontally because otherwise, actor can shoot anywhere and game will end earlier than it should. Shooting Behavior is an action that implement behavior and this make it both action and behavior. Shooting behavior has to be action in order to enable player to shoot different target and this can be done with getnext action without introducing new method for it. Thus, **Open closed** principle is adhered here because ShootingBehavior extends action and use its functionality to its fullest without modify the source code nor changing its states. **Program to interface** principles which allows all class implementing it will only be used and not only that Program to interface helps all class implementing it, hide all implementation with only essential methods is used, other method and data are hidden as well. This adheres to **Encapsulation** principle. Moreover, polymorphism can also be easily achieved if coding is done right.

TeleportBehavior

When criminal kill enough actor, he becomes agent. Agent can fly and teleport almost any location. Agent can do this by implementing behavior interface and does its functionality. **Program to interface** principles which allows all class implementing it will only be used and not only that Program to interface helps all class implementing it, hide all implementation with only essential methods is used, other method and data are hidden as well. This adheres to **Encapsulation** principle. Moreover, polymorphism can also be easily achieved if coding is done right.