Design Rationale: FIT2099 Assignment 2

Team:
Runtime_Terror

Members:
Vladislav Pikulin
Daiki Kubo
Abdalla Abdellatif

Encapsulation and Data Hiding

Description: Throughout our code we have used private keywords for all attributes of classes in order to prevent unwanted access and modification to these attributes from external classes, as well as for some methods of those classes that are not used by any other class. This allows us to hide the implementation of the methods from external classes.

We have used protected keywords for methods that are only going to be used in the class hierarchy and only need to be accessed at a package level, as this does not allow classes that are not in the class inheritance hierarchy to access these methods, and hence creates a layer of encapsulation. Specifically, we have always used protected constructors in abstract classes since they are not able to be instantiated anyways.

We have also focused on using static and final keywords for attributes that are not likely to change in any method, or throughout the game at all, and are meant to stay the same regardless of the class's instance. This allows us to ensure that there are no unwanted modifications to those fields, and hence reduces dependencies with those attributes.

Actions

Description: actions can be implemented without creating a class, and written down in the actors tick method. Nevertheless, in our design we have implemented a variety of different actions for the sake of abstraction, maintainability and extensibility of the code. Not to mention that by using actions we attempt to abide by the single responsibility principle. The tick method is responsible for updating the instance's data, checking it against condition, and returning an appropriate action. To perform the action is not the tick methods responsibility. This allows us to divide the job done by the eat method and create some encapsulation boundaries in order to ensure there is less dependency on exterior classes.

Abstract Classes

Description: Abstract classes allow us to follow the DRY principles and reuse the code that is written in the abstract class for its child classes. For instance, in the Dinosaur class, we have created a tick method that is used by every child class of Dinosaur (Stegosaur and Allosaur). Hence, using abstract classes we managed to reduce the need to repeat our code for classes that may implement similar behaviour or methods by writing those behaviours / methods in the abstract classes. This also in turn, makes our code a lot more maintainable as we do not have to change the code in as many places if there was a need for change.

Class: Egg

Description: Egg is an abstract class which implements Eatable and Purchasable interface. The abstract class is responsible for holding the tick method that is identical for all eggs. The tick method checks the hatchTime of the egg against the time required for the egg to hatch. This way we can check how long the egg has been sitting on the ground against the required time taken to hatch. If the hatchTime is greater than timeNeededToHatch then we can use the location argument of the tick method to remove the item using location.removeItem(this), check the instance of the dinosaur egg, and spawn the appropriate BabyDinosaur instance on the same location using location.addActor method.

Class: MealKit

Description: Abstract class MealKit implements Feedable and Purchasable, this abstract class is used to extend vegetarian and carnivore mealkits. This class is otherwise used for casting purposes to perform shared behaviours among all mealkits so that we do not have to repeat the code for each mealkit type. It is a feedable item that can be fed to dinosaurs. This can be checked by looping through a player inventory in the dinosaur's getAllowableActions method and check if the type of dinosaur is a Stegosaur or an Allosaur and if the player has any Portable item of type MealKit that is either an instance of CarnivoreMealKit or HerbivoreMealKit. This should allow us to allow the player to feed the appropriate mealkit to the appropriate dinosaur. MealKits can also be bought from the vending machine, hence, each child of this abstract class has its own implementation for it's cost. The Eco and Food level it adds to the actor is defined in the MealKit abstract class since they both add the maximum amount of foodLevel (100), and the same amount of eco when fed This class is implemented to obey the DRY principle and to not repeat code that is already written, but reuse it instead.

Class: BabyDinosaur

Description: BabyDinosaur is an abstract class that extends from dinosaur. It has an attribute of age. The reason for having this class is to override the allowable actions for BabyDinosaurs in order to give them their own unique behaviour that differs from

Adult Dinosaurs. To do this we override the allowableActions method in the BabyDinosaur class. On top of that, we need to override the tick method so as to invoke the incrementAge method in the BabyDinosaur class, and check its value every time playTurn runs. This way, if the BabyDinosaur's age is >= 30 we can check its instance and perform location.removeActor(this), and spawn a Dinosaur of the same dinosaur type as the baby (Allosaur / Stegosaur).

Class: Dinosaur

Description: Dinosaur class is a public abstract class that is extended from Actor class that belongs to edu.monash.fit2099.engine folder. This class was created to hold methods and behaviours that both dinosaur types have, hence we are able to reuse the code for Stegosaur and Allosaur classes.

In this class, we have decided to add private attributes of food level that is set to 50 in default, male in boolean type that is randomised by using (Math.random() < 0.5), conscious and pregnancy that are set to false in default, and pregnant counter that counts up to give birth to a baby dinosaur. Inside a constructor of the class, we check if an instance of Dinosaur is Stegosaur or Allosaurs. Depending on the type of dinosaur that is instantiated, either Stegosaur or Allosaur, we add appropriate behaviours to an ArrayList of Behaviour. For example, we have added BreedBehaviour and WanderBehaviour if Dinosaur is instantiated from Stegosaur class. For behaviour, we decided to use ArrayList here since ArrayList is a re-sizeable array, and this way we can have multiple behaviours. We have overridden getallowableActions from Action class. The reason for this is to add actions as some dinosaurs have specific actions that a player can do to them. For example, we add FeedAction that allows a player to feed a targeted dinosaur an item that is casted for Feedable. We also have overridden playTurn as well. The reason behind this is so that we will follow DRY principles. Since both Stegosaur and Allosaur extend Dinosaur, and have similar code for playturn, we can write the playTurn for Dinosaur class instead of repeating the code twice for both dinosaur types. We have added an abstract method called getHatchEco.

Moreover, the Dinosaur class has a boolean attribute of conscious. If the Dinosaur's foodLevel is 0, the Dinosaur's conscious attribute is turned to false. In the Dinosaur tick method, this is constantly checked. If the Dinosaur conscious is false, we increment the unconsciousCounter attribute. If unconsciousCounter attribute >= 20, we check this dinosaur's instance, spawn a corpse of this Dinosaur's instance using location.addItem, we then remove this dinosaur from the map using the location.removeActor(this) method.

Class: DinosaurCorpse

Description: This class is an abstract class that extends PortableItem since the player should be able to carry the corpse. We decided to make the class abstract as we have two types of dinosaur that have different types of name and displayChar. This enables the class to be more extensible and maintainable, hence this is a good design decision. The class contains a public constructor that has capability of EAT_BY_SELF_CARNIVORE and FEED_CARNIVORE as corpses are edible for carnivores and gives ADD_FOOD of 50.

This class extends PortableItem since the player should be able to carry the corpse. We decided to not use an abstract method since stegosaur and allosaur corpses may have very similar behaviours and attributes, hence there may be a lot of repetition. Instead, we have specified an attribute of type Dinosaur. We can check if the dinosaur is dead in the playTurn method of the actor. When a dinosaur dies, we may use the gameMap.locationOf(this), generate the corpse item at this location by checking if the instance of this dinosaur is a Stegosaur or an Allosaur using the dinosaurType attribute. We can then perform gameMap.removeActor(this) to remove this actor from the map.

Interfaces:

Description: Interfaces allow us to make sure that our code follows the ReD principle, and hence, makes our code a lot more maintainable. For instance, using the Feedable interface, all we have to do in order to allow a player to feed an item to a dinosaur is to create a PortableItem that implements the Feedable interface. In this case, we do not have to change any allowable actions in the Dinosaur or Player class. Hence, there are a lot less dependencies in our code due to the presence of these interfaces.

Interface: Feedable

Description: This interface is aimed at helping us identify the portable items that the player can pick up, and feed to dinosaurs. Unlike the design in assignment 1, this interface now only enforces the getAddEco(), as opposed to getAddEco() and getAddHealth(), method. Is is performed due to the fact that some items can be feedable to another actor and some items can also be eaten by an actor once they are lying on the ground. This creates a conflict as some items that implement feedable may not be feedable to an actor, but rather an actor can eat it by themselves on playTurn and hence, will not need to use the getAddHealth method at all. This can potentially cause extensibility and maintainability problems in the future as it is an unnecessary dependency between two types of items. Therefore, we have only included one method in this interface, which is getAddEco since, every time an item is fed to a dinosaur, we would need to increase a players eco points.

Interface: Purchasable

Description: Purchasable interface is used for vending machines and is used along with the BuyAction. It enforces classes to implement getCost() method, which return the cost of the item. Hence, all purchasable items need to have a cost. The BuyAction is responsible for checking the cost of a specific item that is being bought by using the getCost method, reducing the player's eco points by that specific amount using Player's addEco method, if he has enough eco points to buy the item. The Purchasable interface is used to instantiate the BuyAction, and then the Vending machine will have an array of BuyActions. This is helpful to ensure that no item that is not purchasable can be purchased from the vending machine by checking the item's data type.

Class: Application

Description: We have decided to add a method to the application class that allows for any block of dirt to randomly be turned into grass at 2% chance. To do this we can loop through the entire map and check the instance of ground of each block using location.getGround(). If the instance of ground on a certain block is Dirt, then we generate a random number using Math.random()*100, if the randomly generated number is below or equal to 2, we set the ground type to Grass using location.setGround(new Grass()).

Class: BuyAction

Description: The BuyAction is a class that extends Action. The purpose of this class is to create an action for the player that allows them to buy items with eco points that implement the purchasable interface. This class has an attribute of Purchasable item. The item is a portable item, however the casting is done to ensure that no item that is not meant to be sold in the vending machine is added to the vending machine. The BuyAction is also responsible for checking the player's Eco points by using its Actor argument in its execution method, and returning an appropriate output based on whether or not the player has enough eco points to buy the specific item.

Class: VendingMachine

Description: VendingMachine extends Ground. This is done to make sure that the player can interact with the vending machine without having to step on the same block as the machine, and to make sure we can set this ground type so that actors cannot enter this block by using the method canActorEnter(). The vendingMachine has attributes of type BuyAction. This way, we can construct the vending machine to have all of the items for purchase that implement the purchasable interface. We can then override the getAllowableActions to return all of the BuyActions in the vending machine. This should allow the player to get all of the buy actions when near this ground type.

Class: Grass

Description: The class Grass extends ground. Actors can enter this ground type. The player is able to harvest grass. To allow this, we can change the return of the Grass class’s allowableActions method to return actions list with CollectGrassAction in it.

Class: Dirt

Description: We decided to change the tick method in the Dirt class so that it loops through the exits in its locations to generate grass if the ground type nearby allows it to. Since the arguments provided to tick method is location, we can get the exits by using location.getExits() method and check the ground type for each obtained exit with the location.getGround(). If the exit's ground type is a tree then we can generate a random number using Math.random()*100 and check if that value is <=5 (5%). If it is then we may use the location.setGround() method to change the grountype to new Grass(). Alternatively, we can check for the grountype on the exists and if it is Grass, we increment grass counter, if grass counter >=2, and a randomly generated number is lower or equal to 10 (10%), then location.setGround(new Grass())

Class: Tree

Description: The tree class extends the Ground class. It was already implemented, however we need to change it's tick method in order to drop a fruit on any turn. Since it needs to be able to drop a fruit on any turn, we need to utilize the tick method of this ground type. We have decided to add a random number with Math.random()*100, if on any turn, this number is less than or equal to 2, the tree drops a Fruit on the same block. We can do this by using the tick method's location argument, location.addItem(new Fruit()). We have to override the allowableActions method in Tree so as to return an actions list with a PickFruitAction to allow the Actor to pick a fruit from a tree.

Class: Fruit

Description: Fruit is a new class that extends PortableItem to specify that this item can be picked up by actors and implements the interfaces Feedable and Purchasable so that it can be fed to dinosaurs, and purchased from a vending machine. This class has an attribute rot, and a tick method that increments the rot by one on each turn if it lays on the ground (if it's an item in a location), if the rot is larger than a certain amount (which we have to experiment with), then the fruit has to disappear from the location. To do this we can use the tick methods argument location: location.removeItem(this). It also has a cost which is required by the purchasable interface, and addHealth, addEco which are required by the feedable interface. If the fruit is fed to a dinosaur, the FeedAction adds the appropriate amount of eco points to the player based on the addEco attribute of the item being fed.

Class: LaserGun

Description: LaserGun extends WeaponItem. Hence, it has attributes of damage, and verb. Other than that, we have made it implement a purchasable interface so that the player can buy it from the vending machine. Hence, it also has a cost attribute in order to be able to implement the getCost() method of the interface.

Class: Hay

Description: Hay extends PortableItem, since it is portable, and implements the Feedable and Purchasable interface. This allows for this item to be casted into the interface data type so that the player can purchase hay, and feed hay to dinosaurs.

Class: StegosaurEgg && AllosaurEgg

Description: Both StegosaurEgg and AllosaurEgg extend from the Egg class. These two classes have an attribute of cost, and a getCost() method since the Egg class implements purchasable. It was necessary to separate them into individual classes so as to be able to identify the instances of the Egg item in order to know what kind of baby dinosaur it would hatch into. And, since they have different costs, they have different implementations of getCost method and a different value for cost attribute. This allows these two eggs to have different prices in the vending machine.

Class: BabyStegosaur & BabyAllosaur

Description: BabyStegosaur and BabyAllosaur extend BabyDinosaur, the reason as to why it is important to have these classes is to be able to identify their instances and for them to hold their own implementation of getHatchEco method and hatchEco attribute. These will be used to give Player the right amount of eco depending on what type of BabyDinosaur hatches.

Class: CollectGrassAction

Description: The CollectGrassAction is a class that is extended from the Action class that belongs to edu.monash.fit2099.engine folder.
The purpose of adding this class is to allow an actor to take an action of collecting grass and add hay to an inventory by using the addItemToInventory method in the Actor class. If an actor who collects grass from the ground is a player, then the player will obtain 20 eco points using the methods and attributes that we aim to add in the Player class: getEco, addEco.

Class: FeedAction

Description: The FeedAction is a class that is extended from the Action class that belongs to edu.monash.fit2099.engine folder. The purpose of adding this class is to allow an actor to feed a target actor feedable item, which is done by using FeedAction where Feedable item is casted from Feedable Interface. At the execution, its target will heal itself by adding health and the item used for feeding will be removed from the inventory, using removeItemFromInventory method in the Actor class. A player who feeds a target will gain eco points, by invoking the purchasable item's getAddEco() method using the player's addEco() method.

Class: PickFruitAction

Description The PickFruitAction is a class that is extended from the Action class that belongs to edu.monash.fit2099.engine folder.
The purpose of adding this is to allow an actor to pick up a fruit that is on the ground and add it to the inventory, using addItemFromInventory method in the Actor class. The chance of successfully picking up a fruit is 40%. To randomise a chance of the successful action, we have added math.random function. If a randomised chance is greater than a chance of successfully picking up a fruit, then we can pick up a fruit successfully. Otherwise, an actor fails to do so, and an appropriate message is displayed.

Class: HerbivoreMealKit & CarnivoreMealKit

Description: both of these classes extend the MealKit abstract class and use its implemented behaviours. The purpose of separating them is to be able to identify different instances when it comes to feeding a dinosaur, and to give them their own cost attribute.

Class: Stegosaur

Description: The Stegosaur class is a class that is extended from Dinosaur class. The purpose of adding this class is to use super under the constructor of Stegosaur to add name, displayChar, and hitPoints. It is also used to be able to differentiate between allosaur instances and Stegosaur instances.

Class: Allosaur

Description:The Allosaur class is a class that is extended from Dinosaur class. The purpose of adding this class is to use super under the constructor of Allosaur to add name, displayChar, and hitPoints. It is also used to be able to differentiate between stegosaur instances and Allosaur instances.

Class: BreedBehaviour

Description: The BreedBehaviour is a class that implements Behaviour. The class overrides getAction from Action class. Inside getAction, we find the current location of target by looping through the map to find the actor of the same instance as the actor and the opposite gender using the Dinosaurs getMale method. We then check the location of the actor and the target using map.locationOf(actor) and map.locationOf(target), we use this to then find the distance between them. If the distance between the actor and the target is larger than 1, we return a FollowBehaviour so that the actor moves towards the target. When the distance between the target and actor is less than or equal to one, we return BreedAction.

Class: HerbivoreBehaviour

Description: This class implements the behaviour interface and therefore implements the getAction method. As a herbivore, the dinosaur should eat grass and fruit. To achieve this, we have designed a class EatGrassAction and EatItemAction. Before returning a new instance of either one of these two actions, this behaviour is responsible for checking all of the exits for the dinosaur and checking if any of the items in the exit implement Eatable and if they are instances of Fruit. If it returns true, then return new EatItemAction. If any of the exits that are looped through have a ground type of instance Grass, then return EatGrassAction

Class: CarnivoreBehaviour

Description: this class implements the behaviour interface and therefore implements the getAction method. As a carnivore, the dinosaur should eat eggs of other dinosaurs and eat corpses of other dinosaur types. To achieve this, we have designed class EatItemAction. Before returning a new instance of this action, this behaviour is responsible for checking all of the exits for the dinosaur, and checking if any of the items in the exit implement Eatable, and if they are instances of either Egg with an attribute dinosaurType of instance Stegosaur or StegosaurCorpse. If that is true, then return new EatItemAction. If however, one of the exits contains an Actor that is an instance of Stegosaur, return AttackAction on that Stegosaur. Otherwise return null.

Class: Player

Description: Inside the Player class, we have added eco in integer form, which is set to 0 in default. Then, we have also added a void method of addEco that accumulates eco points gained by a player and getter for eco.

Class: Leaves

Description: Upon consultation with Ms.Jasbir, we decided not to include leaves / leaf class due to the fact that we could not understand it’s use or implementation in the program. Herbivore dinosaurs may be able to eat it, however there is no indication of where this item may come from. Hence, we do not know how to implement this item.

Class: GroundInterface

Description: Ground interface contains getFood() method in order to have ground types be feedable in general. This, however, in our design does not make all ground types edible. In order for a ground type to be edible by an actor, it needs to have a certain capability that the actor can eat. Instead, the purpose of this interface is to ensure to have getFood() method ready for use in all ground types so that when actors eat a ground type, there has to be little to now down casting performed, as we can easily check the capabilities of the ground type and access its getFood() method. This makes the code very extensible as it reduces dependencies between ground types and actors that can perform actions on them, and allows us to easily add capabilities to ground types that an actor can check for on ground types in order to perform a certain action.

Hence, utilizing this interface allows us to perform less casting when checking if an actor can eat a certain type of ground as we would not have to check for its instance, cast it in order to access its getFood() method, instead we can check if the ground type has a capability for an actor to eat it, and use its getFood() method to get its ADD_FOOD attribute.

Class: DinoCapability

Description: This is an enumeration. It holds all the possible eating capabilities an actor can have. By implementing this, we can allow an item, ground type, and actors to have capabilities. This makes it significantly easier to check if an actor can perform a certain action on an item/ground/another actor and do so without performing excessive downcasting. This also allows the code to be more extendable since all that needs to be done in order to allow another actor to do something with an item or ground type or another actor is to add a capability. Hence, We believe this to be a good design decision as it is maintainable and extensible

Class: Allosaur & Stegosaur corpse

Description: This class extends DinosaurCorpse since the player should be able to carry the corpse. These classes are child classes from the parent class Dinosaur Corpse. These child classes should have a public constructor to have their own unique displayChar and name.

Class: Actor Interface

Description: This interface provides the ability to add methods to Actor. It holds the methods that every actor should possess, and hence allows us to reduce code smells by reducing the need for down casting when trying to access methods in the interface in an actor of a specific instance. We have used this to reduce the need for downcasting references in the game. We add basic getters, setters and a public void method that adds onto an actor's foodLevel. The class contains setPregnant, getPregnant, getMale, getFoodLevel, setFoodLevel, and addFood. These are the basic abstract methods that all actors should have.

Class: DieAction

Description: DieAction is a class extended from Action that controls the action of actors dying after being unconscious for a specific amount of turns. We check an instance of the actor first and then create a new Corpse depending on its instance and drop the corpse on the map at where the actor dies. The actor is removed from the map. Using this action, instead of doing the stated in the dinosaur class, allows us to utilize the concept of abstraction by hiding the implementation details, avoiding god classes by modularizing our code, and hence making it more maintainable and extensible.

Class: EatGroundAction

Description: EatGroundAction is a class that is extended from Action and this removes the ground type item, such as grass from the map and adds its food to the actor. Then, we set Dirt on the ground after the ground type item has been fed to the actor, and add the food level of the ground type to the actor by utilizing the GroundInterface method getFood(). Similarly, this allows us to hide the implementation details for dinosaurs eating ground types, and creates less dependencies by modularizing the action into its own class.

Class: EatItemAction

Description: EatItemAction is a class that is extended from Action and this removes an item from a location and adds its food to the actors by using the ItemInterface’s getFood() method so that the actors can gain the food level based on the item they have consumed. Using this action allows us to hide the implementation details for actors eating items from a location, and creates less dependencies with exterior classes by modularizing the action into its own class.

Class: BreedAction

Description: If the actor is male, then we set the setPregant setter for the target to be true. If the actor is not male, we simply set the setPregnant setter for the actor to be true. We have added the getTarget method in private for the entire map to find an actorTest from the same instance, but with a different gender. We have added a private distance method to find the current distance of the actor and the target.

Class: LayEggAction

Description: Used to spawn an egg on the dinosaur's location. This action is called when a dinosaur’s pregnant counter is larger than or equal to the turns required for it to lay an egg once it gets pregnant. This is used to modularize the code so that there are less dependencies between dinosaur class and the action to lay an egg. There may possibly be other actors who can lay an egg. We may only have to extend the LayEggAction to make that possible. Hence, it makes the code easily extensible.