FIT2099 Assignment 1: Design Rationale with Preliminary Design Documentation

Team: **Tute03Team100**

Team members:

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

New interface & classes created

See ClassDiagram.png

To ensure our implementation works as expected, we have created a few new classes and one new interface. Having said that, the interface **DinosaurInterface** has been created.

This interface will be implemented by classes **Stegosaur, Brachiosaur and Allosaur.** As we know, an interface works as a protocol/contract such that classes that implement this interface, must also implement all methods in the interface. Here, in our case, the dinosaurs have some additional features that a Player Actor does not have. For example, food level, hunger and breeding ability. With this DinosaurInterface, 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. By doing so, we've successfully achieved the **'Reduce dependencies' (ReD)** design principle. We now will have more flexibility in switching between the functionalities. Indirectly, **'Polymorphism'** is achieved as well, since we are now able to pass different data types to the main class. This also promotes **code reusability**.

Besides that, by inheriting classes (parent-child relationship), e.g. : the three dinosaur classes inheriting 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.

Next, looking into the class diagram, we can see that new classes such as Brachiosaur, Allosaur, Bush, Fruit, VendingMachine, LaserGun, MealKit, Egg and Corpse are created. We can also see that a dependency relationship is maintained between class VendingMachine and classes LaserGun, MealKit, Egg and Fruit. This is because VendingMachine only needs to return new instances of these classes, and does not need to store them as attributes.

For the Corpse class, due to its portability, it will extend/inherit the PortableItem class. When a dinosaur dies, a new Corpse instance would be created according to the type of dinosaur. Thus, depending on the dinosaur's type, each corpse will have a different displayChar and will remain in the game for different periods of time (unless picked up by Player and stored in inventory).

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 give an ecological balance in space, since each corpse would occupy one Ground/Floor.

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, being attacked by Allosaurs or by 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 opportunities for Allosaurs to eat, leading the chances of them being unconscious (which will possibly lead to extinction) to be greatly reduced.

Other classes and their relationship with any pre-existing/newly created classes are all in the Class Diagram.

Preliminary Design Documentation

In order to handle all possible situations that could happen while this game is ongoing, we will be adding new methods (along with the new classes) to this program. With these helping methods, the specified new functionalities should work as expected in the system.

Stegosaur, Brachiosaur, Allosaur

See Stegosaur.playTurn.png, Brachiosaur.playTurn.png, Allosaur.playTurn.png

Since a Dinosaur (Stegosaur/Brachiosaur/Allosaur) has the ability to eat, breed or get pregnant, additional methods such as eat(:Item)*, breed(), isPregnant() is added to the system to handle different actions.
** in the format: methodName(objectName(optional) : objectType/ClassName)*

Instead of creating an independent method to handle each situation where a dinosaur will eat (eg eatFruit, eatCorpse etc.), we created an **eat(:Item)** method that will take in an Item instance as input accordingly (which thus obeys the **DRY principle**!). For example, in cases of Stegosaur and Brachiosaur, these particular herbivore dinosaurs will move to the target destination and eat the fruit, leading the input to the 'eat' method being a Fruit instance (e.g. : eat(fruit)). On the other hand, if a player would like to feed the dinosaur, then the input to 'eat' method will be a fruit or vegetarianMealKit, depending on what the player wishes to feed. Subsequently, the food level of the dinosaur will be increased according to what it ate.

This is all similar to Allosaurs which are carnivores. They could feed on eggs, corpses and carnivore meal kits, thus the input parameter to 'eat' would just be an instance of either Egg, Corpse or MealKit class.

A static variable named counter is also created to keep track of the number of turns of unconsciousness of a dinosaur. If the counter reaches a specified number, then the dinosaur will automatically die and turn into a corpse.

Besides, if the dinosaur is well-fed and there's an opposite sex dinosaur of the same specie in any adjacent square, then there is a possiblity to breed, so **breed()** method is added to handle this situation. Consequently, if breeding is successful, then **isPregnant()** method will be called to the female dinosaur.

Regarding Allosaurs which could attack Stegosaurs, if an Allosaur attacks a Stegosaur, the ‘execute’ method in AttackAction class will be called. In this method, the Allosaur’s weapon would be its sharp, pointy teeth, which would deal 20 damage to a Stegosaur (through Stegosaur.hurt(20) in the ‘execute’ method), and thus deduct the Stegosaur’s foodLevel (= health) by 20, increasing the Allosaur’s by 20. The remaining processes are all as illustrated in the Allosaur.playTurn sequence diagram.

Bush, Tree, Fruit

See Stegosaur.playTurn.png, Brachiosaur.playTurn.png

Moreover, we can either have ripe/unripe fruits on a tree or fallen fruits from a tree on the same square. Therefore, methods such as hasFruitOnGround(), removeFruit() and hasFruit() are included.

Since some dinosaurs have limitations and are only able to eat from ground, **hasFruitOnGround()** is used here to check if there are any ripe fallen fruits on ground.

By using **hasFruit()** method, we are able to check if the tree has any ripe fruits and is ready to be eaten by a dinosaur.

Once a fruit is eaten by a dinosaur, **removeFruit()** will be invoked on the Bush or Tree instance that the dinosaur just fed on, to remove the fruit from the game map.

To handle possible actions that might happen in class Bush, we created similar methods as well. Here, a dinosaur can either eat or skip the fruit from bushes, similar methods such as hasFruit() and removeFruit() are created. Having said that, we will be able to check if there is valid fruit from the bushes. Once a fruit is eaten by a dinosaur, removeFruit() will be called to remove the fruit from the game map. Interestingly, if a Brachiosaur stepped on bushes, there is a 50% probability that it may kill the bush. So, if a bush is not killed when a Brachoisaur steps on it, **bushAlive** will be returned, otherwise **bushDead** will be returned.

VendingMachine

See Player.purchase.png

The VendingMachine class would have methods displayOptions(), displayOptions2(), and its instance (i.e. a vending machine) could return an Item instance.

displayOptions() is used to display all the items for sale and their hitPoints cost.

displayOptions2() is used to display three numbers each representing an item of carnivore meal kit, brachiosaur egg and laser gun. This is because these three items all cost 500 hitPoints, therefore further input is required from the player to know which of these he would like to purchase.

Finally, an instance of this class could **return an Item instance**, since the instances to be returned are all instances of subclasses of the Item class.

Player

Now let's look at the Player class. We've created methods:

1. purchase(), which involves methods enterHitPoints(hitPoints: int), enterOption(option: int), deductHitPoints(hitPoints: int), putInventory(:Item)
   See Player.purchase.png
2. pickFruit(), which involves searchFruit(), searchFruitOnGround(), putInventory(:Item)
   See Player.pickFruit.png
3. removeFromInventory(:Item)

The method **purchase()** is used to handle the purchasing of items from a vending machine.

From the sequence diagram, we can see that firstly the vending machine would call displayOptions(). The player would then use the method **enterHitPoints** to enter the amount of hitPoints, which is also the cost of the item he wishes to purchase. Then based on the hitPoints entered*, the vending machine would instantiate a new instance from the item's class, and return it back to the player, after which the player's hitPoints would be deducted that amount (via method **deductHitPoints**). Lastly, the Item instance returned from the vending machine will be stored by the player in his inventory, using the **putInventory(item)** method, where item is the item purchased.

As stated in the VendingMachine class, if hitPoints = 500, displayOptions2() and **enterOption(option) would be called, but the process of instantiating a new instance, deducting the hitPoints, and storing into inventory is the same.*

The next method, **pickFruit()** is used to handle the interaction between the plants and the player.

If the player is in the same square with a bush, he can call **searchFruit()** to search for fruits in the bush. The bush will return a result indicating whether the search is successful or not. If it's successful, the bush would return a Fruit instance and remove it from itself using removeFruit() method, and the player could store that fruit in his inventory, via **putInventory(fruit)**. If the search for a fruit fails (60% fail rate), a failure message, such as “You search the tree or bush for fruit, but can’t find any ripe ones” would be displayed.

This is the same when the player is in the same square with a tree, except the method called would be **searchFruitOnGround()**, where the player only searches for fruit that is lying on the ground of the tree.

Finally, the method **removeFromInventory** would take in an Item instance as input, and just remove that item from the Player’s inventory. This method is present in the sequence diagrams, after a Player has fed a dinosaur.

Other classes (MealKit, Corpse, LaserGun, Egg)

Each MealKit instance could have one of two types: vegetarian or carnivore.

Each Corpse and Egg instance could have one of three types: Stegosaur, Brachiosaur or Allosaur.

LaserGun class has no significant methods/information regarding it as of now.