

**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 |

- *To explain the choices we made, must explain both how your proposed system will work and why you chose to do it that way*

- *preliminary design documentation: explain how you are going to add the specified new functionality to the system*

*\*Classes in the UML Class diagram without any packages are part of the edu.monash.fit2099.game package.*

----------------------------------------------------------------------------------------------------------------

### **New interface & classes created**

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

This interface will then be implemented by classes such as **Stegosaur, Brachiosaur and Allosaur.** As we know, an interface works as a protocol/contract that classes implement this interface must follow 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, we could add additional features/attributes to only some particular kind of Actor instances (the dinosaurs), but still maintaining the shared attributes of all Actor instances (e.g.: hitPoints, displayChar). By doing so, we 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.

Besides that, by extending classes (parent-child relationship), e.g. : the three dinosaur classes extending Actor class, Tree class extending Ground class etc, we can greatly reduce repetitive code for methods that have similar functionality. In each child class, we will implement these methods only once in their parent class. If we want to provide specific functionality for the child class, then we could override the methods by changing the method signature and body. With this, we’ve just achieved the **‘Don’t Repeat Yourself’ (DRY)** design principle.

Looking into the class diagram, 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 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). We’ve also decided Stegosaur corpses will remain for … turns, Brachiosaur … turns, and Allosaur … turns in the game, if not picked up or eaten.

Fail-fast principle

don’t repeat yourself

liskov design principle

command query principal

----------------------------------------------------------------------------------------------------------------

### **Preliminary Design Documentation**

*preliminary design documentation: explain how you are going to add the specified new functionality to the system*

\*20/4: kexin: will check everything again

In order to handle all possible situations that could happen while this game is ongoing, we decided to add some new methods to this program. With these helping methods, the specified new functionalities should work as expected in the system.

**Stegosaur, Brachiosaur, Allosaur**

Since a Dinosaur (Stegosaur/Brachiosaur/Allosaur) has the ability to eat, breed or get pregnant, additional methods such as eatFruit(), breed(), isPregnant() is added to the system to handle different actions.   
With the **eatFruit()** method (only applies to Stegosaur, Brachiosaur), the particular dinosaur will move to the target destination and eat the fruit. Subsequently, the food level of the dinosaur will be increased accordingly.   
Besides, if the dinosaur is well-fed and if a same specie, opposite sex dinosaur exists in the 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. A static variable named counter is also created to keep track of the number of turns. If the counter reaches a specified number, then the dinosaur will automatically die.

(will complete later)

Regarding Allosaur which is a carnivore, their weapon would be their teeth, which would deal 20 damage to a Stegosaur ( Stegosaur.hurt(20) ).

AttackAction.execute will deduct the Stegosaur’s foodLevel (= health) by 20.

**Bush, Tree, Fruit**

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 called 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 stepped on it, **bushAlive** will be returned, otherwise **bushDead** will be returned.

**VendingMachine**

The VendingMachine class would have methods displayOptions(), displayOptions2(), and could return an Item instance.  
**displayOptions()** is used to display 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, this class **returns an Item instance**, since the instances to be returned all are from classes that extend the Item class.

**Player**

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

1. purchase(), which involves methods enterHitPoints(hitPoints), enterOption(option), deductHitPoints(hitPoints), putInventory(item)
2. pickFruit(), which involves searchFruit(), searchFruitOnGround(), putInventory(fuit)

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.