

**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 the similar functionality. In each child class, we will implement these methods only once in their parent class. If we wanted 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 programme. With these helping methods, the specified new functionalities should work as expected in the system.

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, the particular dinosaur will move to the targeted 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 the same species, 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 successfully, 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.

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 bush is not killed when a Brachoisaur stepped on it, bushAlive will be returned, otherwise bushDead will be returned.