Design Rationale

1. Dinosaur is an abstract class

Since dinosaurs share a common priority in terms of making a decision in their playTurn method, we created an abstract class that all types of Dinosaurs extend. If we were to repeat the code between all the dinosaur types, this would involve a lot of repeated code and thereby violate the Do not Repeat Yourself principle. In addition, we design this class such that it also maintained the Single Responsibility Principle since it was solely responsible for defining the shared attributes and methods between all the dinosaur classes.

2. Dinosaurs store an age tracker and uses the DinoAge interface to determine whether it is a baby or not

Since a baby dinosaur is identical to an adult dinosaur apart from a few attributes that relate to its age, we decided to store age as an attribute in each Dinosaur rather than making a BabyDinosaur class, which minimised repeated code (DRY). Using a DinoAge interface maintained the Open-Closed Principle since the Dinosaur class is still open to extension, but closed for modification since the adult related methods have been implemented with an interface. If we were to add a method to check if a dinosaur is an adult within the base Dinosaur class, then we'd need to check the instance of each Dinosaur. This would break the Open-Closed design philosophy since the Dinosaur class would no longer be extendable, which is what the DinoAge interface aims to address.

3. Util Class incorporating many different functions.

A Util class is useful to store methods and constants that are shared by several classes since they can just be imported by the classes that use them. This follows the Do not Repeat Yourself (DRY) design philosophy, and if we wish to make changes to any of these methods/constants, we only need to change them in one place rather than in many places if we were to not use a Util class. Many of the functions used in the Util class are functions that are generally very useful for multiple classes such as LocateItems and RetrieveItem. Another benefit of putting the functions in the Util class is that it means any further classes or classes that we want to incorporate later on can use these functions very easily meaning expanding the project later on is much easier.

4. Vending Machine inherits from Ground

We believe Vending Machine should be treated like a Ground rather than an Item because it acts much more like a set object on the map (like Ground) rather than an individual item that can be moved and placed in the inventory. This removes a lot of the unnecessary methods and attributes pertaining to Items, which would otherwise violate the Liskov Substitution Principle since the Item class would not be suitably replaced by the vending machine class. However, since the Ground type for VendingMachine is different to other Ground types like dirt, we needed to override the method in Ground so that actors would not be able to pass through it and items cannot be dropped at that location.

5. There are classes for each type of Egg that inherit from the big Egg class rather than using the one Egg class and changing each instance's attributes to meet the requirements of each type of egg.

This follows a similar design principle to the Dinosaur class in that it defines the shared attribute of timeUntilHatch as well as a tick method that decrements this attribute. This ensures that the DRY principle is not violated. If we were to only have one egg class that contains different methods depending on which type of Dinosaur will hatch from it, then this will make the code much harder to follow and hence maintain.

6. Created different classes for action to eat fruit and non-fruit

These actions may be perceived as being similar since they are both called in the FindFoodBehaviour, however, the code in each action differs drastically depending on the target food. This is because eating fruit involves calling methods specific to Flora ground types, whereas eating Items such as eggs and corpses only involves removing those corpses from the Location. Therefore, making use of the Single Responsibility Principle, we made two separate classes for these two actions to span the different responsibilities. If we were to combine them into one action, this would make the control flow of the class significantly more convoluted.

7. EcoPoints has all static methods and static variables

EcoPoints refer to the points that are gained for different actions taking place in the world and are used by the player to purchase from the vending machine. It initially seems to make more sense for the player to store an instance of EcoPoints. However, some of the actions that increase eco points do not involve the player at all (e.g. Fruit being produced by a tree) and if we were to have EcoPoints stored in Player, the player would have to be passed through to Tree. Having EcoPoints have static methods and variables means that it can be referenced by any class without having to go through Player which overall reduces dependencies (RED). A negative of this is that this class is not as well protected due to it being accessible by everything.

8. Flora Interface added for Bush and Trees

Bush and Tree share many of the functionalities pertaining to the Fruit class. To avoid repeating code (DRY), a Flora Interface was created so that methods that are the same for both classes can be defined and implemented from the Flora Interface. Since this interface was designed to only deal with the Ground types that interact with Fruit, it enforces the Interface Segregation Principle. This ensures that only the Ground types dealing with Fruit are required to implement such methods. If we were to include it in the GroundInterface, then this would force all other Ground types to also implement the Fruit related methods, thereby creating interface pollution.

9. Made new GameDriver class (removed World)

The old World class is no longer called by Application. The reason for this is that it did not have the functionality required for a more sophisticated game driver such as being able to count the number of turns and end the run function when a certain number of turns was reached. It was decided that a whole new class would be better as many of the methods would have to be rewritten to account for the new functionality. Some of the code was copied but any methods that were not needed such as endGameMessage were removed. This means that the World class is now taking up space for no reason anymore however this did not outweigh making the added functionality much easier to code up which is important for the Open/Closed Principle (OCP).

10. Made an action made to quit the game

We made QuitAction to allow a player to quit and return to the menu in the middle of a game. This enabled us to list this as an option on the menu with its own dynamically created hotkey. If we decided to not use QuitAction, we would need to manually assign a hotkey and change the Menu class. Furthermore, by integrating it with the Menu class, we could handle the quit action in the GameDriver class rather than the Player class. This enforces the Single Responsibility Principle since GameDriver is responsible for handling the control flow for the game modes. It is also a better use of the current system where we are using actions to incorporate quitting the game rather than treating it separately which makes it harder to understand the code and harder to debug.

11. Ticked rain in GameDriver and stored as static boolean in Util

By determining if it rained in GameDriver, we were taking into account the Single Responsibility Principle since GameDriver was responsible for running each turn of the game. Since rain is determined before the actors' turns are processed, we stored it in the Util class as a static boolean variable. This serves as a global variable that all the locations in the GameMap can refer to when interacting with rain, thereby utilising the SRP again as the Util class contains global methods/attributes accessed throughout the Application. Instead, if we were to determine if it rained within the Location or Ground classes, we would need to pass the parameter through several classes which complicates the control flow of the data. However, having rain tick in GameDriver but stored in Util means that rain methods and attributes are separated into different classes which does not support SRP well.

12. Added method to revive dehydrated dinosaurs in GroundInterface

We added the method to revive dehydrated dinosaurs in the GroundInterface class since it needed to be called for every class that extended Ground. We kept in mind the Interface Segregation Principle - this interface was only responsible for the revival of dinosaurs due to dehydration. If we were to convolute the interface with several methods involving the tick method, then it would cause interface pollution. This would potentially make it a lot more difficult to design classes in the future that may only want to implement certain aspects of the interface, so we avoided polluting GroundInterface.

13. A Hashmap is used to store the amount of EcoPoints gained when certain actions occur and for how much different goods cost at the vending machine

Previously, the points were hard-coded into the code. For example, in the Tree class, it was hard-coded to increase eco points by 1 if the tree produced a fruit. This suffers from connascence of meaning as the program assumes that the hard-coded value of 1 represents the number of eco points to increment. To address this, it was changed so that it would use the HashMap stored in EcoPoints to determine the amount of points to increase by. This is done so that it is much easier to change the ecopoints values at a later date. Rather than having to sift through the code, the hashmaps in EcoPoints and VendingMachine can be easily edited which makes the code a lot more maintainable. Furthermore, this resolves the connascence of meaning as it is made explicitly clear how much to change the eco points by. There are no real significant disadvantages to this.