Design Rationale

The ‘Dinosaur’ class and the ‘Player’ class extends the class ‘Actor’. This is because that they both have names, hitPoints fields and so on in common. The ‘Dinosaur’ class defines the common properties that a dinosaur would have such as behaviors, foodLevel (which refers to the hitPoints inherited from the ‘Actor’) and gender. By inheriting from the class ‘Actor’, the ‘Dinosaur’ and the ‘Player’ class can use the existing fields and methods, which avoids re-producing the codes. If needed, they can override those methods to create their unique properties. This way of inheritance achieves the Don’t Repeat Yourself (DRY) principle.

There are 3 classes that extend the ‘Dinosaur’ class, ‘Stegosaur’, ‘Brachiosaur’ and ‘Allosaur’. Since different types of dinosaurs are closely related to the ‘Dinosaur’, creating an abstract class ‘Dinosaur’ would work better than creating an interface “Dinosaur’.

The ‘Dinosaur’ class ensures that the different types of dinosaurs have common properties as mentioned above. This reduces duplicated code and achieves the DRY principle. By doing so, these 3 subclasses would have their properties defined separately while still be treated as a ‘Dinosaur’ and allows for creating different types of dinosaurs. Furthermore, it is easier to add features to the ‘Dinosaur’ in the future without changing the main structure of the code (which would be costly), this improves the understandability and maintainability of the code.

To reduce dependencies, the ‘Dinosaur’ class and its subclasses (‘Dinosaur’, ‘Stegesaur’, ‘Brachiosaur’ and ‘Allosaur’) are put into a package called *game.dinosaur*. This way of packing strongly related classes provides a good encapsulation boundary for those classes and improves the readability and understandability of the code.

There are serval classes that implements the interface <Behaviour>. The <Behaviour> interface creates different instances of ‘Action’ and allows the ‘Action’ to be executed.

The classes implement the <Behaviour> interface interacts with the (subclasses of) the ‘Dinosaur’ class, the ‘Dinosaur’ has different <Behaviour> stored in it as a field. The ‘Dinosaur’ currently has the following behaviours:
The ‘wanderBehaviour’, which returns a move action that allows the ‘Dinosaur’ to wander to a random location;
The ‘TrackFoodBehavior’ creates a move action for the ‘Dinosaur’ to move one step close to the target food (the food here refers to the ‘Fruit’ generated from the ‘Tree’ or the ‘Bush’ depends on different types of dinosaurs);
The ‘TrackSpouseBehavior’ class is called when a dinosaur tries to find its mate of the same species with a different gender, and step closer the its target mate;
The ‘FollowBehavior’ allows an allosaur to step closer to a target stegosaur for hunting and attacking.

Similarly, a package is created named *game.behavior* (<Behavior>, ‘WanderBehavior’, ‘TrackFoodBehavior’, ‘TrackSpouseBehavior’ and ‘FollowBehavior’) to hold the <Behavior> and the classes that implement it to achieve the DRY and the ReD principle.

‘Fruit’, ‘VegetarianMealKit’, ‘CarnivoreMealKit’, ‘Egg’ (‘StegosaurusEgg’, ‘BrachiosaurusEgg’, ‘AllosaurusEgg’) and ‘Corpse’ are the subclasses of ‘PortableItem’ applying the DRY principle. The reason is that they can all be eaten by the dinosaurs to increase their food level and can be picked up and put into the player’s inventory. The ‘Item’ class is the parent of the ‘PortableItem’ class. As a result, the above subclasses inherit the ‘PortableItem’ would also inherit from the ‘Item’ class, which means that they can have different capabilities such as being eaten by different types of dinosaurs and a lifetime. The inheritance saves lots of work by re-using the pre-defined code.

The ‘Corpse’ class may be argued that it is too big to fit in the player’s inventory and should not be portable. In this case, the ‘Corpse’ is assumed to be portable. Otherwise, the ‘Corpse’ should extend the class ‘Item’ instead, since the ‘Item’ class has a field *boolean portable,* by setting it to *false*, the ‘Corpse’ would be recognized as a non-portable ‘Item’.

‘StegosaurusEgg’, ‘BrachiosaurusEgg’ and ‘AllosaurusEgg’ extend the abstract class ‘Egg’. The dinosaur eggs all have a hatching time and the type of ‘Dinosaur’ they would be after the hatching time. The reason to have an abstract class ‘Egg’ here is to improve readability and understandability by grouping the related classes together. Furthermore, there can be possibly many instances of ‘Egg’, by grouping the dinosaur eggs using the ‘Egg’ class allows for consistency throughout the game to avoid some buggy code that may appear throughout the coding process.

The package (*game.portableItem*) is created for holding those ‘PortableItem’ (‘Fruit’, ‘VegetarianMealKit’, ‘CarnivoreMealKit’, ‘Corpse’ and ‘Egg’). Inside this package, there is a *game.portableItem.egg* package which includes the ‘Egg’, ‘StegosaurusEgg’, ‘BrachiosaurusEgg’ and ‘AllosaurusEgg’ class. This way of creating packages is in line with the ReD principle.

Furthermore, a package (*game.ground*) is created to holds the classes related to ‘Ground’, including ‘Dirt’, ‘Bush’, ‘Tree’ and the ‘VendingMachine’.

The Class ‘Dirt’, ‘Bush’, ‘Tree’ and ‘VendingMachine’ extends the abstract class ‘Ground’. The ‘Ground’ class provides different methods such as to check if an ‘Actor’ can stand on it or not and allows subclasses to define their capabilities. These subclasses can then behave differently by overriding the methods in the ‘Ground’ class applying the polymorphism, which achieves the DRY principle.

A ‘Dirt’ object has a chance to grow up to a ‘Bush’ object. Both the ‘Tree’ and the ‘Bush’ are responsible for growing fruit (an instance of ‘Fruit’). Throughout the game, the ‘Actor’ interacts with those subclasses of the ‘Gound’. Those subclasses provide different allowable actions for the ‘Actor’ to perform when standing next to them. For example, the player can stand in the same location as a ‘Tree’ or a ‘Bush’ object, and search through it to see if it grows any fruits.

The ‘VendingMachine’ maintains a list (ArrayList) of ‘PortableItem’ that can be sold (‘Fruit’, ‘VegetarianMealKit’, ‘CarnivoreMealKit’, ‘Egg’ (‘StegosaurusEgg’, ‘BrachiosaurusEgg’, ‘AllosaurusEgg’), by calling the ‘SellAction’ class later to allow the player purchasing those items.

The ‘LaserGun’ inherits from the ‘WeaponItem’ which is originally in the engine design, which causes damages to the ‘Stegosaur’.

The ‘PortableItem’ illustrated above interact with the ‘Actor’ through ‘Action’. The ‘Action’ class has serval subclasses, ‘BreedAction’, ‘EatAction’, ‘FeedAction’, ‘AttackAction’, ‘FireAction’, ‘PickUpItemAction’, ‘HaverestAction’ and ‘SellAction’.
The ‘BreedAction’ allows the ‘Dinosaur’ to breed with another ‘Dinosaur’ with the same species but different gender.
The ‘EatAction’ is responsible for an ‘Actor’ performing the eating process on the ‘PortableItem’.
The ‘FeedAction’ allows the player (an ‘Actor’) to feed those ‘PortableItem’ to the ‘Dinosaur’.
The ‘AttackAction’ allows the ‘Allosaur’ to attack the ‘Stegosaur’, which is in the original design.
The ‘FireAction’ allows the player to use the ‘WeaponItem’ to fire at the dinosaurs.
The ‘PickUpItemAction’ is in the original design of the codebase, which allows the player to pick up the item (mostly, the ‘Fruit) from the ‘Ground’.
The ‘HarvestAction’ is when the player search through the ‘Bush’ or the ‘Tree’ in the current location to see if there is any ‘Fruit’.
The ‘SellAction’ serves for the ‘VendingMachine’ to sell the ‘PortableItem’ and the ‘WeaponItem’ (‘LaserGun’).

The above classes tight up within the ‘Action’ class. A package is created for the purpose of ReD, named *game.action*, which includes the subclasses mentioned above, the ‘BreedAction’, ‘EatAction, ‘AttackAction’, ‘PickUpItemAction’, ‘HarvestAction’, ‘SellAction’, ‘FeedAction’ and ‘FireAction’.

There is also a class ‘EcoPoint’ that keeps track of the eco points in the game and is responsible for increasing or decreasing the eco points whenever possible. It would have a private integer attribute that stores the eco points and having getter and setters to modify that value. The outside world cannot simply modify the eco points value without using the setters. This avoids the privacy leak due to the immutable property of integers.