DESIGN RATIONALE

Changes are highlighted in Assignment 2
Changes are highlighted in Assignment 3

Dinosaur

Summary of the class (at least):
Attributes:

- behaviours - *List of behaviours*
- foodLevel
- maxFoodLevel

Methods:

- playTurn
- addBehaviour
- setFoodLevel
- getSpecies
- getActorClass

- **Protoceratops**

Summary of the class (at least):
Attributes:

- age

Methods:

- getAllowableActions

- **Velociraptors**

Summary of the class (at least):
Attributes:

- age

Methods:

- getAllowableActions

- **Plesiosaurs**

Summary of the class (at least):
Attributes:

- age

Methods:

- getAllowableActions

- **Pteranodons**

Summary of the class (at least):
Attributes:
- age

Methods:
- getAllowableActions

- **TRex**

Summary of the class (at least):
Attributes:
- age

Methods:
- getAllowableActions

Having a parent class which encapsulates all the common methods between Protoceratops, Velociraptors, Plesiosaurs, Pteranodons and TRex avoids repeated codes. The common implementation can be kept in the parent's playTurn methods whereas the uncommon implementation can be in their respective subclasses by overriding. The Dinosaur class is refactored to an abstract class as the common method playTurn has different implementation between the five. Object then can only be of type Protoceratops, Velociraptors, Plesiosaurs, Pteranodons or TRex. Subsequently, all the Dinosaurs have different food capacity. Having subclasses ables maxFoodLevel to hold different value for different object of different subclass. The attribute maxFoodLevel avoids literals which then reduces errors when changes to these values need to be made to all the code body using it.

Egg

Before moving on to the Baby class, there exists an egg class which object only exist when the dinosaurs reproduce; a behaviour implemented by ReproduceBehaviour (see below). Eggs can be sold in the shop for certain prices which are different among different species of Dinosaur. The original implementation was to have ProtoceratopsEgg and VelociraptorsEgg classes extend from abstract Egg class because they have different prices. Contrastly, it seems like there wasn't a need to create two different classes as all of their implementation is the same which only left the constructors in the subclasses. This was intended to create an abstraction between the subclasses and the StockItem class (which is also forfeited, see below) because there was no need for StockItem class to know/understand the mechanism of (Protoceratops)VelociraptorsEgg aside from retrieving the item's price for money transaction. Note the same application for the extended code from Assignment 3. Previously forgotten to mention, the Egg was created with an Actor stored inside. An Actor object of the same type as its parent were created and passed to Egg upon creating Egg. This prevents the use of instanceof and multiple if conditions to determine which parent laid the egg for

hatching. It also made sure the line of code for creating the baby Actor is inside the correct method of the correct class ensuring good encapsulation for each class.

Baby

- ~~BabyProtoceratops~~
- ~~BabyVelociraptors~~

Again, the class is created with the intention to apply the same concept as the dinosaur and its species classes. The Baby class encapsulates all the common methods between BabyProtoceratops and BabyVelociraptors then uses method overriding for modification. However, it was later understood that the baby turning to adult phenomenon is not a difference in type but rather state. So the idea is forfeited and the phenomenon is depicted visibly by the change of display character of the objects running under the class Protoceratops and/or Velociraptors. Their age and food level are brought forward and their maximum food level and hit points are updated accordingly. Should BabyProtoceratops is created instead of reusing the Protoceratops class with different values for its babies, then there would be a lot of dependency between these two classes as the values from BabyProtoceratops must be transferred to Protoceratops upon replacement. For example, protoceratops.hitpoints = hitpoints, protoceratops.setFoodLevel = foodLevel and etc. if there is anymore.

Food

- ~~HerbivoreFood~~

Just like Egg, Food provides abstraction from StockItem class to HerbivoreFood. The StockItem class does not need to know the function of HerbivoreFood aside from its price and stock count, hence, other data are hidden to classes outside this class. Information hiding reduce dependencies which then reduces the probability of error to arise from future code extension. Also like Egg, there wasn't a need for subclasses as the implementation between HerbivoreFood and CarnivoreFood is the same.Note the same implementation for the extended code from Assignment 3.

StockItem

The implementation of StockItem was forfeited because the methods needed such as getPrice and getCost can be declared in the ItemInterface. Some of these items that are meant to be sold are production of Dinosaurs such as Egg or Corpse which is different from Food. Having ActorSpecies enum class (see below) to associate with the items produced by Dinosaurs easily maps each species to its own prices and/or cost which simplifies price and cost retrievals. Whereas the price and cost retrieval is different from that mentioned above as up to this point (before ActorSpecies implementation was considered), it is retrieved from the instance variable instantiated during creation. The interface declaration allows for different code body of the same methods in different classes implementing the interface. It

is also not bad to have these methods in items not meant for buying and/or selling which just indicates the items to have zero values.

ActorSpecies

ActorSpecies is an enum class of type of Actors. There are namely Protoceratops, Velociraptors, Plesiosaurs, Pteranodons, Fish, TRex and Humans. This class allows us to retrieve information such as nutrients, price and cost of objects associated with ActorSpecies. The information is stored in a map data structure using an anonymous static method. Making the map data structure static and final ensure it to be consistent and unmodifiable to all associated objects. Additionally, the follow behaviour can then use the mapping of each species to their target actors. The getTarget returns an unmodifiable list to prevent privacy leaks from returning variable of reference type. Modification can be done using add and remove methods in the class. Aside from controlling privacy leakage, having an enum class is better than having a global variable (as the data are the same for all objects of the same species(type)) which increases connascence in the whole project code. It also prevents the usage of instanceof and getters which further reduces dependency. A code extension to this class would not affect the other classes and additional classes that may be associated with this class is properly handled by making necessary methods compulsory through ActorInterface.

Summary of the class (at least):
Attributes:

- EGGTONUTRIENTS
- EGGTOPRICE
- EGGTOCOST
- ACTORTONUTRIENTS
- ACTORTOCOST
- CORPSETOCOSE
- velociraptorsTarget
- plesiosaursTarget
- pteranodonsTarget
- trexTarget
- speciesToTarget

Methods:

- getEggNutrients
- getEggPrice
- getEggCost
- getActorNutrients
- getActorCost
- getCorpseCost
- addTarget

- removeTarget
- getTarget

Behaviours

- **FeedBehaviour**
- **FollowBehaviour**
- **ReproduceBehaviour**
- **SearchFoodBehaviour**

These classes are inherited from Behaviour class. The original implementation was to make Grass extend from Actor rather than Ground so that the Protoceratops could "follow" or move towards the grasses when hungry like the Velociraptors follow the Protoceratops to attack and feed on their corpses (avoids repeated codes). However, the Grass do not need a lot of the methods in Actor especially playTurn which only confuses the Player when feedback is displayed after each turn. That would also mean limited location on the map for the dinosaurs to step on when a lot of grass and trees grow. This is where SearchFoodBehaviour comes in. Each species will move towards their respective food source by moving towards objects of classes implementing FoodInterface. Additionally, an ActorClass enum class (see below) categorise the animals to their diet and maps each diet category to the location food source corresponding to the diet.

ActorClass

ActorClass is an enum class which categorise the animals according to their diet category namely Herbivore, Carnivore and Omnivore and their habitat category namely Land-based, Water-based and Amphibian. It maps the diet category to a list of the location of food supply in accord to the diet category. Animals of respective category can then easily retrieve food locations using the method getFoodSource. Additionally, the getFoodSource returns an unmodifiable list to prevent privacy leaks from returning variable of reference type. Modification can be done using add and remove methods in the class. Aside from controlling privacy leakage, having an enum class is better than having a global variable (as the data are the same for all objects of the same class) which increases connascence in the whole project code. It also prevents the usage of instanceof and getters which further reduces dependency. A code extension to this class would not affect the other classes and additional classes that may be associated with this class is properly handled by making necessary methods compulsory through ActorInterface.

Summary of the class (at least):
Attributes:

- herbivoreFoodSource
- carnivoreFoodSource
- omnivoreFoodSource
- classToFood

Methods:

- addFoodSource
- removeFoodSource
- getFoodSource

Actions

- **BuyAction**

Summary of the class (at least):
Attributes:

- Item - *A reference to the item to be bought.*

Methods:

- execute
- menuDescription
- hotkey

- **SellAction**

Summary of the class (at least):
Attributes:

- Item - *A reference to the item to be sold.*

Methods:

- execute
- menuDescription
- hotkey

- **TagAction**

Summary of the class (at least):
Attributes:

- Actor - *A reference to the actor to be sold.*

Methods:

- execute
- menuDescription
- hotkey

The existence of these classes allows buy, and sell item(s) and tag Dinosaur(s) options to appear on the menu for users to select. Both inherits from the existing Action class which could then just override method for their implementation. Transaction works by using added setCash method in Player to add or subtract cash. A message will be displayed by the end of turn with information on whether it is a successful transaction or otherwise which includes reasons such as not enough cash.

Shop

Shop class is inherited from the existing Ground class in the engine. The Shop has a static anonymous method which creates all the objects for selling. Having it as static reduce memory usage by not creating duplicates of the same object when another Shop is placed on the map.

Summary of the class (at least):

- allowableActions
- canActorEnter

Dirt, Grass, Water, Reed

Grass class is inherited from the existing Ground class in the engine. Dirt class make use of the tick method to implement growing Grass and Trees. It reduces dependency in a way that makes use of the current existing dependency (tick currently called in World) rather than creating a new one. On the other hand, Grass make use of both the tick method and the Status enum class that was created in demo files. This avoids repeated codes. New definitions were added to the Status enum class without deleting any just so any class that may use it for previous definitions can still work. Note the same implementation for Water and Reed. Water uses tick to grow Reed while Reed uses tick to generate fish.

Gate

Gate class is inherited from the existing Item class in the engine. It makes use of the allowableActions method in Item to allow actor to move between maps. It allows for MoveActorAction custom definition to display movement feedback and menu options for map transfer to the Player using the added addAction method. The advantage of making it as an item rather than Ground allows for different displaying character for different map entries.

Summary of the class (at least):

- addAction - *to add MoveActorAction to the list of allowable actions.*

Accessor and Mutators

Up until this point, there are quite an abundant public methods. These public methods are mostly accessor and mutators or otherwise abstract method meant to be public for overriding. The advantage of overriding is obviously reduced code repetition but also enough flexibility to allow additional changes/modification apart from the current one. The accessors are fine as well as they provide controlled access to, for instance, instance variables without allowing the world to change its value. The number of mutators are very minimal. As much as they are bad, perfect independence is impossible. Either give up on connascence between classes or connascence between methods of the same class. One plus

point though is the existence of setters allows methods/variables within the class to make use of it so future changes only need to be made within the setters themselves rather than changing the methods/variables format.