DESIGN RATIONALE FOR NEW FEATURES

Overall responsibilities of classes

New / modified class	**Overall responsibilities**
BuyAction	Allow an actor to buy an item and update their balance
SellAction	Allow an actor to sell an item and update accordingly
PlaceTagAction	Allow an actor to place tag on another actor
FeedAction	Allow an actor to feed an item in their inventory to another actor
EatAction	Allow an actor to eat an Item or Ground and gain food point
Dinosaur	Contains attributes and methods shared between all dinosaur species to reduce code repetition (i.e. feed, grow, breed, age, food level). Species-specific behaviours can be implemented by overriding methods such as getAllowableActions() and playTurn() in the base class to exhibit polymorphism.
SeekFoodBehaviour	Allow a dinosaur to check surrounding environment for food source and can return an AttackAction, EatAction or the action returned by FollowBehaviour to move toward the target.
FollowBehaviour	Right now this allows an actor to follow another actor by returning an action to move toward the target. We plan to modify this to make it works for also targeting a Location. If not possible, we will implement a Behaviour such as GoToLocationBehaviour.
Vegetation	As there will be grass and trees that will both possibly share similar growth functionality as well as serving as a food source for herbivores on the map. We think it is in our best interest to generalise the growth of these two classes by having them both inherit Vegetation which will in turn inherit Ground. Vegetation may have a constructor that specifies the food points of that piece of vegetation.
Grass	The class that will take the place of dirt when the x% chance that a piece of dirt changes into grass. Inherits from Vegetation as it can be eaten by herbivores and can also grow and has food point values.
Shrub	A subclass of vegetation. Shrubs could perhaps appear in the same way that grass does from dirt but instead grow from a patch of grass at an even smaller likelihood. Shrubs will have different sizes as they grow (small, medium, big) and will be edible by different dinosaurs depending on their size.

Further explanation

Growing Grass

The simplest approach is to add a grow() method in the Grass and Tree and call it in the tick() method. However, this functionality is quite similar for both classes so it may be a good idea to first implement these separately and then refactor by putting the method inside the GroundInterface to reduce code repetition.

The requirement for growing grass is very simple at this stage. That is, if an x% chance of dirt turning into grass is successful: then that location changes the ground from a dirt instance to a grass instance. This probabilistic evolution takes place in a willGrowGrass() method inside of dirt and then called by the tick() method also in dirt. However, it is possible that we may want grass to affect the likelihood that neighbouring locations that contain dirt also grow grass (similar to a tree). So, as we anticipate that trees and grass will be implementing similar functionality, the grass class will inherit the Vegetation class that in turn inherits the Ground class.

Growing Tree

We will generalise the entire growing process for any subclass of Vegetation in the superclass Vegetation itself. The vegetation class will check all neighbouring locations and whether they are suitable to grow vegetation in: if so, the neighbouring locations will have their ground changed to the same type of vegetation as itself. This decision will enable us to easily implement other types of vegetation that have the “growing” functionality with ease as we will not need to re-implement the code for ‘spreading’/’growing’ every time a new type of vegetation class is made.

Dinosaurs

Different species

Similar to the class hierarchy we are implementing in Ground→ Vegetation→ Tree/Grass/Shrub. For the same reason we are implementing an abstraction for the general Dinosaur class as all dinosaurs will share a multitude of behaviour and functionality. In order to eliminate as much code repetition as possible: we will be having each species of dinosaur inherit the Dinosaur abstraction instead. This is also in order to help us expand in the future in the sense that we can quickly add more dinosaur species with minimal timeinput. Furthermore, this allows the use of polymorphism as we can specify the unique behaviours of each species of dinosaur by overriding methods such as getAllowableActions() and playTurn().

Laying Eggs and Corpses

The dinosaurs also have an attribute of type Species which is an enum. This is useful to generalise behaviour like laying an egg or dying and leaving corpse, which requires creation of species-specific items. These items will take Species in the constructor so that they know how to correctly set the attributes and methods according the the species. Thus, code related to laying an egg and dying can be placed in the base Dinosaur class. However, this may result in the use of large switch cases, which generally signals a could-be-better design and may be worth refactoring in the future.

Baby dino

To implement baby dinosaurs, we can either add an attribute or method to indicate that it is a baby or we can create classes like BabyProtoceratops extending Protoceratops. We chose the first option for several reasons. First is that the baby dinosaur does not differ too much from its adult version aside from not being able to breed so it does not really need polymorphism. Second is that when the baby dinosaur grows up and transform into an adult, it would be a hassle to instantiate a new instance of the adult and then copy over the data. Third is that the latter method is not very great for expandability. Adding a class for a new species requires us to also add its baby class.

Food related

We think a good starting point is to implement a method isFood(object) in the Dinosaur base class to check if the dinosaur can eat that object. Each Dinosaur subclasses will have different lists that keep reference to what they can eat. An alternative way is for the food object to know if an actor can eat it or not but this can be harder to maintain and expand in the future. Adding a new species would require us to go to different food items and add the new dinosaur to its list.

There are 2 behavioural changes when a dinosaur is hungry. First, it moves toward the nearest food source. Second, it only ever attacks or eats when it is hungry.

These behavioural decision is implemented inside SeekFoodBehaviour. In this class, the locations nearby the dinosaur are checked if they contain Actor, Ground or Item that is food and then return AttackAction or EatAction. If there is no nearby food, further locations are checked and FollowBehaviour is used to determine the action to move toward that food source. The SeekFoodBehaviour is only added when the dinosaur is hungry.

Store

Store inherits from Ground because it can ‘offer’ actions like BuyAction and SellAction to nearby actors. It does not need to be an actor since it does not need to do any action offered.

Action

The new actions inherit from the base class Action because an action can be offered by Ground, Item and Actor. Action has access to information related to the actor that is executing it and the map they are in (and thus the location and ground as well). The action is constructed by the actor that is offering it and we can pass relevant information through the constructor. In particular, the execute() and menuDescription() methods are called within the World class in the engine when an actor has chosen to do the action. It is the easiest way to perform interactions between Actor, Item and Ground.

For example, for BuyAction and SellAction which allow the player to buy and sell at the store, their constructors need take in the reference of the item that is being bought or sold. In execute(), we have a reference to the actor performing the action and thus we can add or remove the item from the actor inventory. Similarly, PlaceTagAction should take in reference of the dinosaur (or target) to be place tag on while the EatAction should take the item or ground to be eaten and their location in their constructor.

Behaviour

Behaviour is an abstraction for complicated logic that returns an appropriate Action. We can essentially achieve the same functionality if we put all the logic code in determining the action inside say the playTurn() and getAllowableActions() methods of the Dinosaur class. However this would mean that Dinosaur will be directly dependent on a lot of other classes and there will be repeated code with other classes. Overall, using behaviour reduces dependencies and allows reusable code. Whenever we see complicated logic code used to determine an action that is not encapsulated, we should refactor that code inside a Behaviour class or refactor the Action to implement Behaviour.

The use of behaviour also allows behaviour chaining. An example from our design is that SeekFoodBehaviour is dependent on FollowBehaviour. SeekFoodBehaviour takes care of the logic of finding the nearest food source and passes on the information of target to FollowBehaviour. FollowBehaviour job is to return a moveActorAction that moves the actor closer to the target. This modularity using Behaviour allows chaining and thus can achieve overall complicated behaviour from separate, simpler behaviour. This also offers better maintainability since we only have to work with and think about 1 aspect of a new behaviour at a time.