FIT2099 Design Rationale

Michael:

Having Dinosaur as the parent abstract class and then having abstract classes Carnivore and Herbivore inherit it reduces the need for repeated code (Don't Repeat Yourself) and data members for all dinosaurs. Doing so will also future proof the design as adding dinosaurs is now as easy as inheriting either the Carnivore or Herbivore class.

Classes are responsible for their own properties with Behaviour and Action classes explicitly being responsible for certain behaviours and actions that dinosaurs can take. Attributes only used in that class will only be declared in that class and variables used within methods only will only be declared within that method, declaring things within the tightest possible scope. This keeps classes very simple.

All behaviours also inherit from the Behaviour interface, once again removing the need to repeat code in each behaviour class. The same goes for action classes that make use of the abstract Action class code (HatchingAction being the exception, explained in the breeding section).

Some behaviours also make use of the same actions, such as UnconciousBehaviour and DyingBehaviour making use of the same DieAction class. This once again avoids the use of repeat code.

While Dinosaur don’t use HatchingBehaviour, HatchingBehaviour still inherits the Behaviour interface as it still uses Behaviour’s code. To combat the use of Dinosaur being able to use the HatchingBehaviour class, there will be exceptions to make sure Dinosaur will not be able to use the HatchingBehaviour class at initialisation, following the fail fast design principle. Despite being called HatchingAction, HatchingAction does not inherit from the Action class as it does not make use of the code within the Action class.

For dinosaurs to grow, a behaviour or action was not introduced. Instead there will be a boolean indicating if the dinosaur is an adult or not and after 30 turns the boolean will turn from false (for baby) to true (for adult). This reduces the need for extra classes, thus reducing dependencies.

The AttackBehaviour and AttackAction are part of Dinosaur rather than Allosaur since there are many other dinosaurs that can attack (including possibly the stegosaur). This reduces repeated code and supports easy extensibility and maintainability.

Stegosaurs possibility won’t have all the behaviours and actions given to the Dinosaur class. This can once again be ensured with the use of exceptions to make sure non-Stegosaur behaviours and actions aren’t included at instantiation, following the design principle of failing fast.

The abstract class EatAction is inherited by HerbivoreEatAction and CarnivoreEatAction. The two classes will use the code of EatAction except only take in HerbivoreFood and CarnivoreFood respectively. This eliminates repeated code while separating what herbivore and carnivore dinosaurs can eat.

MoveToFoodAction applies to both Herbivore and Carnivore dinosaurs so a parent class of CarnivoreFood and HerbivoreFood was made, called Food. MoveToFoodAction will take Food as a parameter meaning CarnivoreFood and HerbivoreFood can also be used as arguments, making use of polymorphism and eliminating repeated code.

Kenny:

Grass/Trees:

Grass needs to be visible on the map so it inherits from Ground and we will be making use of polymorphism to avoid repeated code especially setting a new ground for a particular location. Although it does not explicitly state that trees can be grown as well, we can still assume that Dirt could potentially turn into a Tree in the event that we need it to.

In terms of Tree dropping fruits it's almost the similar process with but rather then setting ground its adding Item to the specific location. For future purposes, potentially we may want to create a new class above Grass and Trees so that we don't have repeated code in terms of adding Item to a location as we could initialize what items it drops when instantiating the class. In addition, Fruit and Hay classes will be considered Item as well, so that the player can add them into their inventory.As it has methods that we need so we don't have to repeat ourselves.

For player harvesting crops, we discovered that the engine had already provided us a useful class so rather than implementing our own we will be using the PickupItemAction to harvest a specific crop, thus applying, trying not to repeat ourselves and reduce dependency principle . However, since potentially we may need to modify PickUpItemAction, HarvestAction will inherit it, so we don't modify any of the engine code and in case we need modifications to the class itself can be overridden. As such, classes will be taking responsibility for their properties via using allowableActions() and capabilities to ensure that the right Actor can perform actions(e.g. only Player can harvest crops).

Ecopoints and VendingMachine

Ecopoints was approached in various methods, but mainly there were two potential ideas in regards to how it was done. We could have an ecopoints as a class with static variables and a static method in which all classes could access, but the worrying thing was privacy and not following encapsulation principle. Furthermore, another assumption was made that there was only one player in a map, therefore we decided to initialize eco points in the players class. As ecopoints class will be used by the player and easier for the player to find the current balance of ecopoints.

In terms of accumulating those ecopoints, in HatchingAction,Grass,HarvestAction,EatAction Will be able to have access to Actor in some manner, thus being able to get access to the player which is very easy to call a method to increase the current balance.

VendingMachine will inherit from Ground since it needs to be seen on the map, it will have a BuyAction which is quite similar to how actions are which was explained by Michael. VendingMachine can only be used by the player so it can be restricted via allowableActions method or capabilities. VendingMachine will not initialize items until it's purchased which then will be added to inventory this is intended cause there is no point initializing items in vendingMachine if some of it is not going to be used as to reduce memory usages.

We also decided to categories items based on if a carnivorous dinosaur was able to eat it or a herbivore dinosaur can eat it. This covers maintainability. If we decided to implement more dinosaurs it will be a lot easy and it will reduce a lot of repeated code.

The only other interesting Item is the LaserGun since it is not edible it will be categories under Non-Food and also since it’s a weapon, there is already an existing class in the engine for us to use which was WeaponItem, so there was no need to create other unnecessary class. The laser gun will have ShootAction in which the player will be able to use it, if they want to kill a dinosaur.