Michael:

Having Dinosaur as the parent abstract class and then having Stegosaur and Allosaur 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 news types of dinosaurs can be done by simply inheriting the Dinosaur class.

Classes are responsible for their own properties with most Behaviour and Action classes explicitly being responsible for certain behaviours and actions that dinosaurs can take.
The SeekFoodBehaviour class is an exception to the Behaviour calling on Actions for when it is time to eat. It was impossible (to my knowledge) for me to pass on the food level points the food being eaten gives as Action only takes Actor and GameMap as parameters. Not using an EatAction also reduced dependency as there is now one less class.

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.

Instance variables are also declared private or protected exposing as little data as possible to outside classes that do not need to know about it.

All behaviours also inherit from the CommonStuffBehaviour abstract class which implements the Behaviour interface, once again removing the need to repeat code in each behaviour class. CommonStuffBehaviour stores data members and method that are all used by the Behaviour classes that inherit it, reducing repeated code.

Some behaviours also make use of the same actions/behaviours, such as AttackBehaviour and BreedingBehaviour making use of the same Followbehaviour class and its method to follow a target or breeding partner. This once again avoids the use of repeat code.

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.

SeekFoodBehaviour can be used by both carnivores and herbivores with the help of Capabilities. This reduces the need for separate classes and repeated code for the specific feeding type and even allows for omnivores where omnivore dinosaurs can take in both carnivore and herbivore capabilities (not implemented in code, but is possible with some form of data structure to store the Enums in the dinosaur). This also reduces the need for Herbivore and Carnivore interfaces further reducing dependencies and classes.
The use of Capabilities also helps us easily tag which food can be eaten by which dinosaur type and which dinosaurs can be or can't be attacked.
Eggs also can't be eaten by dinosaurs of the same type (Allosaurs can't eat Allosaur Eggs). This is implemented by passing the dinosaur that laid the egg and storing it's class as an instance variable. Classes of the dinosaur eating the egg are then compared with the class

stored inside the egg and if they are the same, then the egg cannot be eaten. The Class instance variable is also used to get the constructor with the getConstructors method to instantiate a new dinosaur once it is time for hatching. Doing this supports modularity.

Behaviours are kept in an array list and to be iterated over through, reducing the lines of code needed to call all the behaviours in the dinosaur class. Behaviours are customised within the subclasses of Dinosaur using static methods. The methods return an array list of behaviours customised to the specific dinosaur which is passed into the Dinosaur constructor, reducing the need to store behaviours and repeat the code for calling behaviours in the subclasses, instead calling on them in the Dinosaur parent class. The methods that are used to create and return the array list of behaviours are static (and private) because it is the only way to call a method before an instance of a dinosaur is made (at least to my knowledge).

Dinosaur also inherits Actor can therefore have actions and behaviours called upon it with the use of polymorphism and casting. This follows the Liskov Substitution Principle and reduces the need to repeat code.

Excessive use of literals are also avoided with the help of constants.
Exceptions are also used in Dinosaur constructor to help with failing fast when there is an invalid input.

Kenny:

(In terms of Assignment 2 progression no real modifications in terms of design besides the ecopoints section)

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.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. In addition, Fruit and Hay classes will be considered Item, 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).
In order to ensure that we are giving vegetables to Herbivore or meat to carnivore, they also have addCapabilities to ensure that there is no confusing among player giving Hay to lets say a Carnivore Dinosaur.

In terms of how it works:

In the World class more specifically in the While stillRunning loop , it will call tick method on GameMap,inside the tick function in GameMap it will traverse through all possible locations and invoke the tick function inside of instances of Location. Then inside the tick method of Location invoke another tick method, but this time on Ground Class. The tick method in Ground class is the exact place where we need to go; firstly we will check if the ground is an instance of Dirt or Tree or Grass. If it is a Dirt we will check its surroundings since the location is passed as the parameter we have the option to call getExits() to check its surroundings to see if there Dirt we will have some sort of accumulator, to increase a number. Once done we will use math.random to get a number and if it succeeds then we will create a Grass class inside Dirt, and use the location.setGround method to change the instance of the ground of the location.Similar process as Grass using ticks, but rather setting a new ground where just adding an instance of a new Fruit onto the location called addItem via location. If the player wants to harvest fruits we use playTurn method to find the current location of the player and see if there are items on the location of the player and if so, then we grab the items and put them
onto the action parameter.

Ecopoints and VendingMachine

Ecopoints was approached in various methods, but mainly there were two potential ideas in regarding 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. **Ecopoints has been instantiated in WorldModified** and then it gets passed onto MapModified. In the event that there is more than one player then we can have an array of Ecopoints and use a hashMap via (Player,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 allowableActions is called, then it only adds items that purchasable 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 to reduce memory usages or cant be afforded with the current balance.
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.
Laser Gun no longer needs shootaction but inherits from WeaponItem.

For example, if grass is grown from Dirt then ecopoints must be increased, so it would be similar to how it would be done in the Grass section, but if the constructor of Grass would contain the reference of a location, then we use map() method to get a GameMap. Once we

have the GameMap reference it should be very easy, just locate the player reference and then add points to the ecopoints variable.
In terms of how it would be done for the others, the majority of them inherit from Action, and in the execute function there is a reference to GameMap that's all we need we do the same thing as what we did above.
Should probably also be mentioned that Grass and VendingMachine will be created inside the application in the FancyGroundFactory parameter, when FancyGroundFactory is initialized. As with other classes that are visible in the map, VendingMachine will inherit from Ground so that it can be in the map. As with anything with Ground class it will have a tick method, in here we will check if a player is located in the same spot, and it so give them a buyAction where they could buy one thing from the store for a turn. Although the way we're implementing only allows them to either move, stand-still or purchase items. Once they have purchased the item, it will be added to the player inventory via addItem().