Design rationale

Dirt, Bush, Tree

Dirt, Bush and Tree all inherit from the Ground class. This is because they need to be properly represented on any Location with a special character, and don't need any of the capabilities that the other classes like Item or Actor have, such as moving and being picked up.

The Bush class has a special interaction with Branchiosaurs, where every time its tick() method is called, it will check if a Branchiosaur is standing on its location, and has a chance to be killed (be replaced with dirt) if there is one. This is the most straightforward way of implementing this interaction that doesn't involve editing the engine, and it's also the most modular, as Branchiosaur won't know anything about this interaction whatsoever, and any changes to the interaction will only affect one class (Bush).

Dirt has a chance to grow into a Bush, which will simply replace the ground at its location. Both Bush and Tree hold an ArrayList of Fruits and have a chance to produce one each turn during the tick() method call. They also have a chance once every tick to drop a Fruit from their array onto the Location where they exist (Where it'll be stored in the Location's ArrayList). This allows the Fruit's tick() method to be called using the Location's tick() method call to determine whether it is in an Actor's inventory or not, and thus whether it should rot. This also lets actors interact with the Fruit

Dirt and Tree both implement the hasFood() interface to show that they can be eaten from, and to ensure that they have the methods necessary to check their 'inventories'.

Fruit

Fruit inherits from the PortableItem class, mainly because it needs to be able to be picked up and added to a player's inventory. It also needs to be interactable for other Actors while on the ground and sold by the vending machine (which stores a Map of Items). The Fruit also doesn't have inbuilt Actions other than the ones provided by PortableItem. The FeedActions will be provided by the eligible Dinosaurs should the player have the Fruit in their inventory to allow them to feed the dinosaurs.

PickFruitAction

PickFruitAction inherits from the Action class for obvious reasons. It allows for Actors to pick/eat Fruits in Bushes or Trees. Limitations for each Actor are implemented in PickFruitAction's code as well, such as Stegosaurs being unable to eat from trees.

VendingMachine

VendingMachine inherits from the Ground class. It shouldn't act as an Item, and doesn't need the functionality that Actor provides (like playTurn()), and therefore Ground is the most suitable. It holds a Map of Items without multiplicities. The Items themselves are used as keys, and their prices are their values. This makes it so that theItems themselves do not have to know their prices, and prices can be different across different vending machines should the need arise.

BuyAction

BuyAction inherits from the Action class, and is returned by the VendingMachine class when a player is beside it. One BuyAction is returned per affordable Item. Each BuyAction knows the Item which the player will buy should it be executed, along with its price. Implementing BuyAction allows for a suitable menu description to be displayed, as well as let the act of buying an item take a turn as it should.

checkDead() and DieAction

The checkDead() method and DieAction class are mainly how Actors die. checkDead() is implemented because dinosaurs only die when killed by an Allosaur or after 20 turns of unconsciousness. They fall unconscious when their hitPoints drop to 0 instead of dying. Therefore we have to check for when they actually die after being unconscious. It is implemented in Dinosaur's playTurn method because we can't modify World to check at the start of each round. DieAction is also a consequence of this limitation, since an action needs to be returned once playTurn is called. DieAction handles the process of replacing the actor with a suitable corpse and returning the menu description of the death.

Corpse

Corpse inherits from the PortableItem class. This is so that they can be picked up and dropped. It also helps HungryBehavior identify it as food for Allosaurs.

Egg

Egg inherits from the PortableItem class, mainly because it needs to be able to be picked up and/or bought from the vending machine, and doesn't need to be able to move or take actions. When the egg is ready to hatch, it will create a new BabyDinosaur in its location.

Eggs have an enum attribute that determines what type of egg it is, and their time to hatch is also stored as an attribute according to their type. This is so that new subclasses don't need to be created just to store two different class attributes and no new methods.

MealKits

MealKits (VegetarianMealKit and CarnivorMealKit) inherit from the PortableItem class, mainly because they need to be able to be picked up and/or bought from the vending machine, and don't need to be able to move or take actions. The MealKits don't have inbuilt Actions other than the ones provided by PortableItem. The FeedActions will be provided by the Dinosaurs should the player have the MealKit in their inventory.

LaserGun

The LaserGun inherits from the WeaponItem class, mainly because it needs the damage and verb attributes. It also needs to be able to be picked up, dropped and/or purchased from the vending machine. The AttackAction for attacks should be provided by the Actors, and so won't be provided by the LaserGun

Dinosaur, AdultDinosaur and BabyDinosaur as Parent Class

The adult dinosaurs Stegosaur, Allosaur, Brachiosaur will inherit from a abstract parent class AdultDinosaur while the baby dinosaurs, BabyStegosaur, BabyAllosaur, BabyBrachiosaur will inherit from another abstract class BabyDinosaur. Both AdultDinosaur and BabyDinosaur will extend from a generalized Dinosaur class which will extend Actor. By having them split into adult and baby classes, the behaviours and methods necessary for each of their functionalities (e.g. adults will be able to mate and get pregnant while babies can grow up) can be split up so babies can't get pregnant. The Dinosaur class will implement attributes and methods shared between all dinosaurs that the child classes can inherit and override as needed. Not only does this reduce code duplication, it also makes it easier to maintain and debug.

Dinosaur, AdultDinosaur and BabyDinosaur are also abstract so that they can't be instantiated.

The alternative, by having Stegosaur, Allosaur, Brachiosaur all extend from actor will require the same attributes (e.g. isFemale, isPregnant) and it's methods to be implemented in all Dinosaur classes. This will make it harder to make changes to dinosaurs in general as you will need to make the change in all types of dinosaur classes and increase code duplication.

This can be further specified by having BabyStegosaur, BabyAllosaur, BabyBrachiosaur inherit from BabyDinosaur for the same reasons as mentioned above. Babies share a lot of functionality like not being able to mate and growing into an adult dinosaur after some time.

Stegosaur, Allosaur, Brachiosaur, Pterodactyl and dinosaur babies as separate classes

By having Stegosaur, Allosaur, Brachiosaur, BabyStegosaur, BabyAllosaur, BabyBrachiosaur be separate classes that share an ancestor class Actors will make it easier initialise them with different characters (‘A’, ‘a’ etc.) on the map. It also allows for us to add specific behaviours that only pertains to a specific class. For example, only BabyAllosaur and Allosaur will have an AttackBehaviour.

Otherwise, complicated logic operation would need to be done to determine the dinosaur type and its methods/behaviours/attributes which can be avoided by just having them be different classes.

DinosaurEnumType

DinosaurEnumType is an enum type that holds the different species of dinosaurs available. It reduces the number of classes needed for Corpse and Egg as we can represent all three types of dinosaur Corpses and Eggs by them having a DinosaurEnumType attribute. Implementing it in Dinosaur also allows for Eggs and Corpses to be produced using the Dinosaur’s DinosaurEnumType.

Behaviours

The behaviours will implement the Behaviour Interface as that will enforce each individual Behaviour to implement a getAction() method. This will make sure that each class will have the proper method to be used as behaviours.

AttackBehaviour

BabyAllosaur and Allosaur will be associated with AttackBehaviour as they can attack Stegosaurs and BabyStegosaurs when they’re nearby (in their exits). AttackBehaviour also has a dependency on Stegosaur, BabyStegosaurs and AttackAction. AttackBehaviour will return an AttackAction if the target is attackable. For example, whether an Allosaur has previously attacked the Stegosaur in the last 20 turns. The unattackable actors will be stored in a Hashmap in the instance of the Behaviour. This behaviour will also only attack stegosaurs when they’re nearby (in the exits) and will not follow the stegosaur to attack them.

HungryBehaviour, eat(), EatAction, food and fromTheseEatsThese

HungryBehaviour will be associated with every Dinosaur as they all can go hungry. It dictates how the Dinosaur will act when it's hungry and what food type it eats. Each dinosaur has a list of classes that it eats as food, and a map of grounds with the food that it eats from said grounds. This helps HungryBehavior determine if and where any available food is for any dinosaur, whether on the floor (stored in the Location instance) or in a Ground (Bush or Tree). It will then return a MoveActorAction if the food is too far away, or an appropriate EatAction if the food is close enough to be eaten. EatAction has two constructors; one for when the food is on the floor and one for when the food is in a Ground. It handled the logic needed to differentiate between the two and simply calls the dinosaur's eat() method to let it eat the right amount of food from the right source. This modularises the code as much as possible. The eat() method allows the dinosaurs to determine how much they gain from each food source themselves.

HornyBehaviour, BreedAction, FollowBehaviour and FollowAction

HornyBehaviour will be associated with AdultDinosaurs. It dictates how the Dinosaur will move to find a mate in order to procreate. It is dependent on Location, BreedAction and it will have an instance of FollowBehaviour. FollowBehavior allows an actor to keep following a target by using FollowAction and getNextAction. It will continue to follow the target until it reaches it. When it does, it will perform a pre-decided action towards the target, in this case a BreedAction. So by having HornyBehaviour use FollowBehaviour it would enable it to target a mate to breed with.

LayEggAction

The LayEggAction is an action for pregnant AdultDinos to lay eggs. It will call the layEgg() method of the adult dinosaur which will place an egg on the location the dinosaur is on.

GrowUpAction

GrowUpAction is used by babies when they need to grow up. This is decided by an attribute of the baby that counts down to 0 by the number of turns needed to grow up. When it reaches 0 a GrowUpAction will be executed that playturn that replaces the baby with an adult version of themselves.

Lakes, Fish and Rain

Lakes were implemented as a subclass of Ground, mainly because it made the most sense and had the required functionality for it. Since there was no editable global class, we decided to implement rain using the Lake class. One special lake will be a controller lake, which will handle the tick calls and whether it rains or not. This does unfortunately mean that it will not rain at all if there are no lakes in the game, but since there should always be at least one on startup and there is no way for lakes to disappear, this design works for now. Lakes also keep a record of how many fish they have in them, but do not actually store Fish instances within it. This simplifies the code, but might need to change if the functionality of fishes change, or if information about the fishes need to be retrieved from the lake.

Pterodactyls: Flying and Breeding

Pterodactyls have a simple variable to tell whether they are flying or not, which is set to true or false depending on how long they've flown for without landing on a corpse or tree. As for breeding, female and male Pterodactyls have different behaviors, as detailed in HornyBehavior. Female Pterodactyls will seek out a tree with an adjacent tree, and male Pterodactyls will seek out a female on a tree with an adjacent tree. This is a simple way to have them both breed only atop trees. Female Pterodactyls use a similar logic to find a tree to lay an egg on. Pterodactyls that are hungry and flying over a lake will also automatically catch between 0 and 3 fish during the playTurn function call. This doesn't require an action, and therefore reduces the amount of code required. It however does still utilise the eat() method for modularisation and maintainability

Drinking and Thirst

Thirst is handled in a similar way to hunger, having a maximum limit and being checked to determine if the dinosaur is conscious or dead in their respective methods. A dinosaur will drink from the first adjacent lake if it is next to one using the drink() method. This helps with modularisation and maintainability. This also doesn't require an action as to reduce the code required.

Game and Enum GameMode

This class is used to handle creating the game, setting up the world and the main menu before that. Most of the code previously in Application has been moved to this class. This is to allow for the ability to replay the game and to be able to implement different game modes. It also falls in line with object oriented programming to have code to set up the world be in a game class instead of in the main Application class.

Other than that, the GameMode class is nested in the Game class because code relevant to the game mode should be associated with the game.