Design Rationale

Dirt, Trees and Bushes

The Bush class inherits from Ground, however, its implementation would be done through the Dirt object, where we have implemented a method growBush(Location) that utilises the adjacent squares to determine the likelihood of growing a Bush. The Bush and Tree class would generate instances of the fruit object and place them in an arraylist as part of its attribute. All of these classes utilise a random number generator in order to generate a Bush or a Fruit. The Fruit rotting has been taken into account and is carried out by having an age attribute that determines when the Fruit will disappear from the map.

Dinosaurs

An abstract Dinosaur class has been implemented and the various dinosaur species will inherit this class. The attributes included in the Dinosaur class range from age, gender, adult age, hunger, display characters, breeding threshold, and many more. We chose to use more constants for the attributes as most of these values will not change throughout the game. In our implementation, a baby dinosaur will be an instance of their respective dinosaur classes (Stegosaur, Brachiosaur, Allosaur) with the exception that they will have a different display character and will not be able to breed.

Pterodactyl

The Pterodactyl dinosaur is similar to the other dinosaurs with the addition of two new attributes, an integer flyDuration and a boolean onGround. The flyDuration indicates how many turns the dinosaur has flown and when it reaches 30, it will land on the ground (sets onGround to true) and search for a tree. Every time it lands on a tree, the flyDuration will reset to 0 and the dinosaur can resume flying. For breeding, the Pterodactyl can breed only if both dinosaurs are on adjacent trees. Besides that, laying its egg must also be done on a tree, and this introduces new attributes and methods in the Tree class to check if an egg is already occupying a tree. The Pterodactyl is carnivorous and has more food options as it can consume fish from lakes, eggs, and corpses. When it is hungry, it will check its surroundings for the closest food source and will move towards it. If it reaches a corpse or an egg, it will land on the ground in order to eat it.

Behaviour

The behaviours we have created for the Dinosaur thus far are WanderBehaviour, AllosaurAttackBehaviour, ThirstyBehaviour, HungryBehaviour, and BreedingBehaviour. The WanderBehaviour allows the dinosaurs to roam around and if there is food at their location, it will return the suitable Eat Actions. This behaviour also accounts for Brachiosaur's eating conditions and its likelihood of stepping on bushes. The AllosaurAttackBehaviour is specifically designed for the Allosaur and allows it to hunt for Stegosaurs, this behaviour will be set when the Allosaur's hunger level drops below a certain threshold. The BreedingBehaviour enables a well-fed dinosaur to scan the map and find potential mating partners. It will then proceed to follow the nearest mate and will return a BreedingAction once it is on an adjacent square. The HungryBehaviour will force the dinosaur to actively search for a food source, and this behaviour will only be set if it's hunger levels drop below the well-fed threshold (minimum hunger level to breed).

Player Action

The actions we have created are FeedingAction, PickFruitAction, and the PurchasingAction. The FeedingAction is used when the Player wishes to feed a dinosaur and will increase the Player's eco points when it is executed. The PickFruitAction is used by the Player to retrieve fruit from trees and bushes, and it uses a random number generator to determine the success of the action.

Dinosaur Action

The actions we have created are AllosaurAttackAction, BreedingAction, CarnivoreEatAction, and HerbivoreEatAction. The Eat Actions are quite simple as they simply heal the dinosaur according to the Food input. The AllosaurAttackAction is used when an Allosaur tries to attack a Stegosaur, however, this action will fail if the target has been attacked recently by the same Allosaur. The BreedingAction will check for the female dinosaur and will create an Egg instance that matches the species of its parents. This Egg object will be added to the female's inventory and a pregnancy timer will be set.

Eating and Feeding

We have implemented an Eating Action for each dinosaur species as each of them have different food conditions. The Stegosaur and Brachiosaur species share the same Eat Action as they have similar diets, whilst the Allosaur will have its own different Eat Action. To speculate, Allosaurs tackle its eat functionalities with three different classes, which is AllosaurAttackAction, AllosaurAttackBehaviour and CarnivoreEatAction. When the allosaur is set to execute it's attack behaviour, it will locate the nearest available stegosaurus as its target to prey upon. The amount of damage done to the target is then the amount healed for the allosaur. If the Stegosaur survives the attack, the allosaur cannot reengage the same target for a set period of time. How we handle this is by adding the reference object of the stegosaur to an arraylist. If the Stegosaur is killed as a result of the attack, a corpse item of the stegosaur would be generated on the location. Corpses are another form of consumption for Allosaurs. At the location of a Corpse item, the Allosaur will consume the instance of the corpse object, removing it from the game map. This is handled by the CarnivoreEatAction, and it does not only apply for corpses but as well the egg object items.

Eggs and Breeding

The Egg class is a subclass of Food as it can be consumed by an Allosaur. The abstract Egg class is also inherited by the more specific StegosaurEgg, BrachiosaurEgg, and AllosaurEgg classes. In our mating implementation, after a Breeding Action has occurred between two dinosaurs, a pregnancy timer will be set. When the timer has reached its limit and it is time to lay the egg, it will then be removed from the dinosaur’s inventory and placed at the location of the female dinosaur. This resets the pregnancy timer and sets the dinosaur’s attribute to not pregnant. Once the Egg object has been placed onto the map, it starts its aging process and will hatch once the threshold has been reached (threshold is different for each species). Once it hatches, the appropriate dinosaur object will be instantiated with an age of 0.

Death

To tackle the Death portion for relating Dinosaur objects, we created a subclass of item named Corpse. Within the corpse object, we have implemented several methods that would handle despawning the corpse object, removing it from the map. The corpse class as well takes in a parameter to assign its species, as we need to match the species of the dying dinosaur. The main method for death is initialized within dinosaur and it works in conjunction with other methods written to tackle certain aspects and conditions. As stated previously, upon a dinosaur’s death, we generate a Corpse object at its location, and remove the actor from the map. To specify other methods that are related to death, within dinosaur class we have a method that deals with dinosaurs that are unconscious. If the dinosaur stays unconscious for a specified amount of time, the death function will run to execute the said dinosaur.

Eco Points and Vending Machine

On handling currency for the game, an EcoPointStorage integer attribute was instantiated within the Player class. It is declared as a static variable to be accessible by all other classes, as each there are several methods in different classes that interact with EcoPoints. Although the variable has been declared as static, we still have written setter and getter methods within the Player class to interact with the eco points. The Vending Machine class was instantiated as a child of the ground object. With the creation of the Vending Machine object, a new action subclass was added to aid its functionalities, named as VMPurchasingAction, which extends the Action class. VMPurchasingAction executes the purchasing action to perform the transaction of eco points and the player obtains the item. If the player has insufficient eco points, no item would be added to their inventory. The item selection by the user is passed from the vending machine object to the VMPurchasingAction. That’s how the Vending Machine is handled.

Lakes, Water and Rain

The Lake class extends Ground and is placed inside the Environment package. Its attributes are an ArrayList of Fish that represents the fish population in the lake, and an integer, sips, that shows the available sips. Every turn, there is a possibility to add a new fish to the lake if it is not full and this probability is based on a random number generator. For the implementation of rain, we modified the Bush, Dirt, Tree, and Lake classes to check every 10 turns if it rains. The probability of rain is decided by a random number generator and if it passes a certain threshold, it sets the boolean variable, isRaining, to true. The isRaining variable is mainly used to revive the dinosaurs that are unconscious due to thirst.

Thirst

The thirst level of a dinosaur is initialised when it spawns and will go down by 1 every turn. When the dinosaur is adjacent to a lake, it can drink from it (if there are available sips) to increase its thirst levels by 30 points (80 for Brachiosaur). If the dinosaur's thirst levels go below a certain threshold, it will activate the ThirstyBehaviour that forces the dinosaur to actively search for a lake. If the levels reach 0, the dinosaur will become unconscious but can be revived if the location it is at experiences rainfall.

Second Map

For the inclusion of additional maps, modifications were done to the default Application class which has now been renamed as WorldBuilder. Initialization of an arraylist was necessary to store all the map instances which are beneficial for classes that require access to the maps.

WorldBuilder has a single method named generateMaps() which is the main function of creating the map instance and all its objects. Not only limited to creating the map, this method also handles the spawning of the Player object and all related dinosaur objects.

The movement between maps are handled within the Player class where modifications were made to it's playTurn implementation. When the Player is found to be at the most upper north of the first map, he/she will have a prompt to continue North previously unavailable to the Player. When selected, the Player will be moved to the second map instance, placed at the very south edge of the map. To return to the first map, the player now at the south edge of the map has a prompt to continue South. This functions identically to what has been prescribed beforehand. The naming convention of these map teleports is to preserve the player's immersion within the game, making it seem like the transition between maps are seamless, portraying the maps as a connected instance.

Sophisticated game driver

To enhance the user experience of playing the game, we created a new class which now acts as the new game driver for our Jurassic park game. Within the new driver, we had built the required new additional game modes of sandbox and challenge mode respectively. When the selection of challenge mode is initialized, the user has control over the winning conditions of the game. To tackle on how the game tracks these requirements, we initialized some class variables of challengeEcoPoints and challengeTurns which store the ecopoints and the amount of turns respectively. With the utilization of getter methods, the tracking of these conditions is done through the Player class. Within the Player class, we added a counter that determines the current turn of the game, and compared the user inputted values at every turn instance to determine if the user's objectives are in completion. If the user fails to obtain the specified amount of EcoPoints within his inputted turn restrictions, the game will end. The ending sequences are handled by an action class specifically generated to fulfill the sequence requirements. On the contrary, if the user does accomplish his goals within the constrict, he will be rewarded with a winning dialog which is handled by the GameCompletionAction. Additionally, the original game menu had no quit functionality and the only way to exit the game was to stop the program. We have addressed this issue in the new game driver with the implementation of a dedicated quit button bound to the character ‘0’. Now the player has full control over quitting the game.