Nicholas Emmanouil
Dulan Waduge

Design Rationale

General Design and implementation

Use of Abstraction

In creating the necessary classes for the game, abstraction will be frequently used to serve as a baseline template for classes which share a significant portion of functionality. The use of abstraction serves to enhance the DRY aspect of the code, limiting repeated code and also allowing for any bugs to likely only appear within an abstract class thus being easier to fix.

Abstraction will be implemented within the Actor class to serve as a template for all animated objects within the game e.g. player and dinosaurs.

Alternate Implementation: Aside from utilising abstract classes, class behaviours can be managed utilising an interface.

Use of Inheritance

Extending classes will be a frequently used design principle within the construction of the game. Similar to abstraction in the case that one class may share attributes common to another, inheritance allows for a subclass to use attributes from a parent class. Inheritance will be applied to the baby dinosaur classes (A baby Stegosaur will inherit attributes of the Stegosaur class). Inheritance allows for maintaining consistency among classes which act nearly identically and as also demonstrated in astraction reduces repetitive code and aids in bug fixes.

Classes

Classes were utilised to encapsulate an object which contained multiple attributes. This can be seen within the dinosaur and player classes as each of these objects require various attributes and also methods to define how they interact in the game.

In contrast, any ‘object’ that does not require various attributes to define its existence in the game can simply be an attribute of a larger class. For example, instead of creating a separate class for a corpse object of a dinosaur, a dinosaur will contain a boolean attribute to simply state whether it is dead or alive. This attribute will itself change the functionality of the dinosaur class. Further if an object in the game has no attributes i.e. eco points, it is also an attribute itself of a class (eco points is an attribute of the player). By also refraining from creating superfluous classes the number of potential dependencies between classes can also be reduced (instead of the player being dependent on the class of the eco points object, it is not at all dependent by making eco points an attribute of the player).

Alternate implementation: As previously highlighted, attributes of classes may exist as their own classes such as the corpse of a dinosaur. This would give more flexibility in determining how such an object interacts with its environment.

Interactions (Sequence Diagrams)

Sequence diagrams describe how groups of objects collaborate and capture their behaviour of a single scenario. A simple scenario would be the player's interaction with a vending machine. When the player buys an item from the vending machine, it requests an item in exchange for eco points. We also added lifelines, which is basically an activation bar that shows when the participant is active in the interaction. This would be useful when we have multiple objects in the sequence diagram. For example, if there are two objects, Vending machine and Weapon, the activation bar highlights what object is being used at a certain time.

We created four sequence diagrams for the main actors, Player, Stegosaur, Allosaur and Brachiosaur since most interactions in the game are played by these actors.

Associations and Dependencies (UML Class Diagram)

Within the structure of the game, classes related to each other can exhibit associations and dependencies. In determining which classes must be dependent on another, from a programming perspective, any class A which calls class B in one of its methods depends on class B. Hence any classes dependent on another must have reason to call the latter class in their methods.

The act of a dinosaur laying an egg for instance requires the dinosaur to call a method to lay an egg which essentially creates a new instance of an egg class within the method of the dinosaur. Hence the function of laying an egg in the dinosaur class is dependent on the properties of the egg class.

Associations represent stronger relationships between our objects. Whilst the classes are not dependent on one another, they act on each other in some manner. For instance, an allousaur is not dependent on an egg object that it eats. However, it is associated with the action of eating the egg. The justification for an association is that there exists a relationship between two classes where there doesn't exist a dependency.

Iteration 2 adjustments and rationalisation:

During the implementation of the game, many aspects of the original design were changed. The following denotes the greater alterations of the original design:

Baby Dinosaur Implementation Using Enums:

Contrary to the previous design where dinosaur babies exhibited a separate class to the dinosaur itself, enumeration is now employed within the class in the form of an enum class labelled Types. This enum class offers a separate constructor for the dinosaur's states such as being a baby or corpse. In this way it is easier to either create a baby dinosaur from birth of an egg or allow a baby dinosaur to mature into an adult and thus inherit attributes of the adult type.

Capabilities replacing boolean attributes of a class:

In the previous design, attributes such as a dinosaur's state (e.g. corpse) or a fruit's rot were intended to be determined using boolean variables. However, given the game engine's restriction on accessing class methods, these boolean values which were intended to be set and retrieved using getters and setters are replaced by capabilities. Capabilities are implemented as enum constants from a certain enum class e.g. Status or DinosaurType. These constants are applicable to abstract definitions of a class such as Actor and Item and Ground and hence negate the need to downcast to a class that is not

Additional notes on implimentation:

Actions and Behaviour classes:

As the game is played, the player must perform an action at each step. While this occurs, the dinosaur NPCs must also perform certain actions that are not triggered by player input. To handle these, behaviours are implemented. Behaviours are used to create a sequence of actions that has a dinosaur reach its end goal or to simply fulfil a passive objective. These included:

-Breeding Behaviours
-Finding Food Behaviours
-Hunting Behaviours
-Unconsciousness Behaviours

At a particular stage, a dinosaur will choose a behaviour to complete a certain objective, by doing this they can interact with their environment and appear to have their own free will whilst the player plays the game.

The player does not exhibit any behaviours but only actions. This is because upon each turn the player can choose any single turn they wish, therefore creating a behaviour to map the player's actions would be pointless and restrict the player experience. Instead, the player exhibits possible actions upon each turn that allow it to do things such as:

-Pick up fruit
-Feed a dinosaur
-Make a purchase at the vending machine

Fruit 'dropping' from trees:

Initially, two arraylists were constructed to serve as the locations on the tree and on the ground, the two places in which fruit can be found on a tree block. However this resulted in the need for heavy downcasting and also was not compatible with the PickupItemAction as there already existed an arrayList to contain items on every block.

To rectify this as the player can only pick up fruit which is on the ground of the tree and can only pick up items that are portable. Newly produced fruit is thus not made portable and hence the player is unable to retrieve the fruit. When the fruit 'drops', using capabilities to determine if a fruit is on a tree or the ground, a non-portable fruit is removed from the location's items array and replaced with a portable one. Hence a fruit has dropped and the player is able to pick it up.

Items on the ground for vending machine:

As the player needs to purchase items from the vending machine, it was decided it would be easiest to place these items on the 'ground' of the vending machine tile. Instantly this allows for the player to pick up these items as if taking them from the vending machine. To further emulate a purchase, a buy action was created.

In the instance that a player stands on the vending machine block, the pickup item actions are removed from the Actions arraylist as possible actions and are instead for each item replaced with a buy action. This buy action requires an ecopoints transaction to take place for an item to be picked up and in order to make sure it can be bought again, picking up the item does not result in it being removed from the tile.

Nicholas Emmanouil
Dulan Waduge

Events triggering ecopoints increase:

Multiple events can trigger a player to gain ecopoints. In the events of eggs hatching, fruit being produced on trees or feeding a dinosaur, capabilities were implemented to aid in causing this payment to the player.

For example, each time a tree produces a fruit, the player must be rewarded with an ecopoint. To do this, a method within the player class is used to sweep all locations of the map for trees on a turn. Trees which have produced a fruit on that turn will be assigned a capability to indicate this. If the sweep identifies a tree with this capability, it will reward the player with a single ecopoint and then this capability is removed in preparation for the next turn.

This idea is applied to any item which exhibits an event that rewards the player. Another example is when a dinosaur hatches from its egg. This dinosaur is given a capability to show it was just born, and once this is identified by another similar method, the player is rewarded and the capability is removed.

Capabilities:

In order to adhere to good object oriented programming practice, capabilities are employed to avoid the need for frequent downcasting. Capabilities allow for any item or actor to be assigned an enum constant which can be accessible from their parent class capability methods. For example, instead of downcasting from actor to stegosaur, one can simply check if the actor possesses the stegosaur capability and not downcast.

Capabilities are also used to mark certain restrictions upon actors and objects, for example. The State enum class possesses a constant ‘ACTIVE’ which can be applied to objects that can be dormant or need a dormant state. This includes eggs to keep them from hatching within the vending machine.

Capabilities also aid in restricting unwanted behaviours such as cross species breeding. Capabilities are used to ensure that dinosaurs may only mate with those of compatible capabilities.

Nicholas Emmanouil
Dulan Waduge

iteration 3 adjustments and rationalisation:

Improved triggering ecopoints increase:

As opposed to the use of various methods in iteration 2 to engender the increase of ecopoints, the ecopoints variable is made as a public static variable instead. Thus, one ecopoints variable is accessible from the player class. This choice was made because of the fact that there exists only one player within the game and thus it is acceptable to make the variable static. Because of this, outside classes and methods can now alter the ecopoints value and the use of extra methods within the player class becomes redundant. The cleanliness of the code is increased and the amount of repetitive methods is decreased.

Second map and adjusting map variables to static variables:

To allow the player to travel between maps, an action is created such that when the player approaches the border of a map, they are given the choice to move to the next map. The map can be switched directly from the player class when such an action is selected. By making the maps static, it allows for the player class to access the map it needs to switch to with great ease and also allows for each map to run concurrently despite only one being printed onto the console.

Separate behaviour class for Pterodactyl breeding:

As opposed to utilising the current breeding method which governs how other dinosaurs find a mate. In order to separate and modularise the unique behaviour of a pterodactyl during breeding, its behaviour is given its own class. This allows for similar behaviours to be kept in their own classes and to save a single breedingBehaviour class from handling vastly different actions. For example, the Pterodactyl needs to find a tree upon which to perch before breeding. This means that it requires a target location as opposed to a target actor. As such, the Pterodactyl breeding behaviour class will need to import and utilise classes that are not used in the traditional breeding behaviour.

Unique corpse class extending portable items:

In place of the given corpse item declared within the attack action, a separate corpse class now replaces the need for this item. A corpse class was created to internally govern a corpse's behaviour and reduce its dependencies with other classes that may use or instantiate it (i.e. a dinosaur). With this class, it is easier to internally manage corpse food levels, capabilities (what type of dinosaur it came from) and also despawning which provides a much faster and cleaner alternative to using a class to sweep corpses off the map externally.

A terrain and location targeting class:

Complementary to the FollowBehaviour class, there are many instances in which a dinosaur needs to reach a particular location rather than an actor. To reduce the repetition in blocks of code written with double nested for loops to hunt a particular location on the map, this method is encapsulated within a separate class known as TerrainLocationBehaviour. This class is used to govern all actions by actors that require them to reach a point on the map and thus if actors are improperly reaching a location, the bug can be confined and constrained to a single class.

Pterodactyl feeding on fish implemented in the lake class:

Because Pterodactyls can have access to 1, 2 or 3 fish. It is easier to determine the quantity of food the Pterodactyls can actually access by performing the feed action from the class of the lake object over which the Pterodactyl is flying. In this case, less methods such as getters and classes of capabilities are necessary to transfer information between the lake and the Pterodactyl. Instead, the lake checks if a Pterodactyl is currently flying above the lake and if so, the Pterodactyl is ‘given’ health by the lake and fish are removed from the lake accordingly.

Vending Machine Menu:

As opposed to utilising a buyaction which restricts the player’s ability to interact outside of the vending machine on the block of the vending machine, for greater clarity, a menu is offered instead with ecopoints being displayed above the menu for each turn taken. This further allows the player to easily identify which items they are able to make purchases of and once done, the player can enter an exit integer to return to the world of the game.

Redesigned Game Driver

The current driver class is just the class that contains the main method and the game is run whenever the application is run/debugged. However, to make it more user friendly and more interactive, a start menu with multiple options for the player is now included. Once the new game driver is run, the user is presented with a welcoming menu and the user can select between two game modes; SandBox and Challenge. Sandbox mode is just a free roam/creative

mode where the game is just being played. However, when Challenge mode is selected, the user has to enter a target number of ecoPoints as well as a limited number of turns. The target ecopoints must be achieved within the given amount of moves to win the game. Else, the player loses.

Heavier use of method instruction through the Dinosaur abstract class:

Given feedback on the repetition of the code within the dinosaur class, to adhere to DRY coding principles, methods that shared common if not identical instructions were moved to the dinosaur abstract class such as the code for pregnancy and unconsciousness. In doing this, it not only makes debugging and identifying issues with the methods more contained, but it reduces the need for repetitious code within the dinosaur subclasses allowing for more clean code.