**FIT2099 Assignment 1**

**Tutorial 9 Team 9**

**Prepared by:**

**Joanne Ang Soo Yin**

**Nelly Tay Yee Ting**

**Jonathan Wong Leong Shan**

**Inheritance**

Inheritance is used in the design. When one class extends from another class, it becomes the child class of the class it is extending from. By extending from the parent class, the child class will be able to inherit the attributes and methods from the parent class. As there will be similarities between the child and parent class, these methods and attributes can be reused, thus reducing repetitive code which complies to the *Don’t Repeat Yourself* principle.

Classes involved in inheritance:

|  |  |
| --- | --- |
| **Child class** | **Parent class** |
| Bush | Ground |
| Dirt | Ground |
| Tree | Ground |
| Fruit | Item |
| MealKitsItem | PortableItem |
| Egg | PortableItem |
| LaserGun | WeaponItem |
| BuyAction | Action |
| BreedingAction | Action |
| LayEggAction | Action |
| PlayerFeedAction | Action |
| SearchItemAction | Action |
| DinoActor | Actor |
| Allosaur | DinoActor |
| Brachiosaur | DinoActor |
| Stegosaur | DinoActor |
| Player | Actor |

**Enum**

Enum classes are used in the design. There are a lot of fixed sets of constants used throughout the program which include attributes and methods. By using Enums, attributes are not needed to store the fixed values in certain classes. It is also more type-safe compared to using constants as it gives a more descriptive type name. Methods that are constant can also be accessed easily through the Enum class instead of needing to go through each class to find a method we need. It is cleaner, more readable and more maintainable to use Enum classes.

Enum classes:

* DinoEncyclopedia
* FoodType
* PregnancyStatus
* PregnantCapability
* Sex

**DinoActor - extends Actor**

**Stegosaur, Brachiosuar and Allosaur – extends DinoActor**

All dinosaurs share a lot of common characteristics, such as feeding, breeding etc.

Hence, to adhere to the *Don’t repeat yourself* principle and improve maintainability, we will create an abstract class DinoActorthat inherits from Actor. In turn, all Actors which are dinosaurs, namely *S*tegosaur, Brachiosaur and Allosaur, will inherit from DinoActor*.* These three classes will inherit all methods from dinosaurs and only override the getAllowableActions method or the playTurn method if necessary

New attributes with appropriate setters and getters will be added to this base class for simulating the dinosaur Actor’s functionality, which will be specified in the corresponding sections later for clarity. However, there are two important methods of DinoActor that will be highlighted. As rule of thumb, the roles of the two methods will be as such:

1. getAllowableActions method

Returns Actions that mimic interactions between twoActors on adjacent squaresonly. This is because this method will only be called every turn when another actor is on adjacent squares to the other actor. In this situation, we **benefit from polymorphism** by **overriding this method in DinoActor’s child classes to return an Actions object that contains BreedingAction and / or AttackAction (by calling getAction on BreedingBehaviour and AttackBehaviour classes respectively) and / or PlayerFeedAction .**

Note that actions will always be added in the order BreedingAction, AttackAction, then PlayerFeedAction - its importance will be explained in the playTurn method.

1. playTurn method

When the playTurn method is called for the dinosaur actors, the playTurn method in their respective classes will call the base class DinoActor’s playTurn method.

The playTurn method in DinoActor has the responsibilities to:

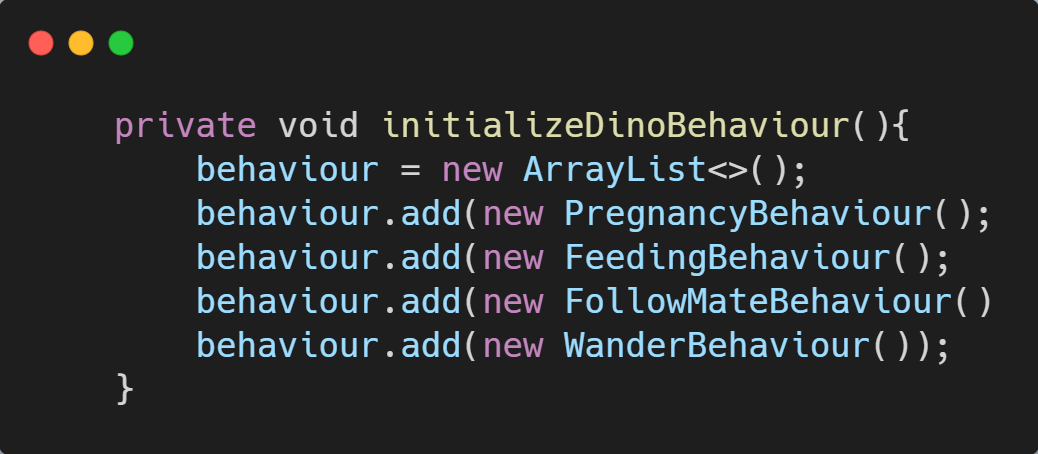
* Update the state of Actor: increment age, decrement food level, display hungry message
* Given all possible actions the Actor can take, determine and return the actual action taken by the NPC

In order to fulfill the second responsibility, i.e. returning the actual action taken by the NPC, we will create classes that implement Behaviour interface:

* **Behaviour-implementing classes** are **utilized for doing necessary processing of Actor state** and **reduces clutter in the playTurn method,** by having all the condition checking statements (of whether an Action can be done by an Actor) inside the getAction of a Behaviour. An action is returned when calling the getAction method if all conditions are satisfied, otherwise null is returned.

Implementation of the decision making on which action to return will be as such:

* Whenever creating a new instance of DinoActor, Initialize the ***behaviour***attribute with Behaviour-implementing classes in an decreasing order of priority, i.e: first one to be added is the most important. The following example is the priority we have decided on in the preliminary design:



Then, In the playTurn method of DinoActor:

1. Receives *actions* argument, which in the case of dinosaurs will only have BreedingAction and / or AttackAction and / or PlayerFeedAction. As mentioned above, they will always be in order.
2. Loops through all Behaviour objects in the behaviour call getAction, but only stores the first non-null outcome in a local variable. Since our behaviours attribute is initialized based on a priority, it will help us to get the Action of highest priority in behaviours.

Note that it is important to loop through the *behaviour* attribute and call getAction for each and every one, since we can do some necessary processing of an Actor’s state. For example,

* + If the Actor is pregnant but not due to lay egg yet, in the getAction method it will help update the number of turns the Actor has to wait, then return null

1. Decide which is the actual Action taken
   * If the lastAction has a next action (checked by calling getNextAction), that Action will be chosen
   * Otherwise, if the actions argument received has at least one Action, the first action is chosen. (refer 1)
   * Otherwise, the first non-null result that we obtained from looping through all behaviours is chosen. (refer 2)
   * If the *lastAction* argument has no next action, the *actions* argument has no Action objects and all Behaviour-implementing class in *behaviours* return null when getAction() is invoked on them, return DoNothingAction.

**DinoEncyclopedia Enum class**

There are a lot of values that we need to keep track of for dinosaur Actors, such as: number of turns till the pregnant dinosaur lays an egg, number of turns till a baby dinosaur reaches adulthood, initial food level etc.

These values are constants and belong to their corresponding dinosaur classes, not to a specific any object. Hence, in order to have **a standardized set of values** necessary for initialization or any other usage, they will be stored in the DinoEncyclopedia class.

Note that the following code snippet is only an illustrative example, not all enum keys or other values to keep track of are included:



A private static final field of type DinoEncyclopedia, say *DINO\_TYPE* will be declared and initialized to their corresponding Enum values for Stegosaur, Brachiosaur and Allosaur classes. Whenever a new dinosaur object is created, the constructor simply needs to access the appropriate values in this Enum class for initializing instance variables.

The motivation behind this:

* Cleaner code in Stegosaur, Brachiosaur and Allosaur due to less fields needed to store constants
* Standardized values, can be access by other classes apart from dinosaur Actors too
* Separation of concerns and single point of change, whenever we want to change a value simply look into this class
* Hence, easier maintenance

**Dinosaurs growing up**

The dinosaur actors shall be represented in the console with the first letter of their names, whereby the lowercase form represents a baby dinosaur and the uppercase form represents a grown up dinosaur, eg: ‘a’ - baby Allosaur, ‘B’ - adult Brachiosaur, ‘S’ - adult Stegosaur

Dinosaurs added to the map at the beginning of the game are adult dinosaurs. Dinosaurs that hatch from eggs are baby dinosaurs.

Required instance variable for DinoActor:

* age – an integer that represents the age of the dinosaur

To simulate the process of baby dinosaurs growing up:

* In the *playTurn* method for dinoActor base class, have a method that increments *age*, and check if *age* has reached target for maturity for that dinosaur, if matured, change the display character of to uppercase form to indicate adulthood

Note that maturity age is an example of what can be stored in the DinoEncyclopediaclass mentioned above.

**Breeding**

**Sex Enum Class**

* Has values MALE, and FEMALE
* Used as possible values for an instance variable in DinoActor to indicate the sex of the dinosaur
* Defined to **make code less error prone** by preventing using literals to represent sex

**BreedingBehaviour - extends Behaviour**

Check the following conditions of whether breeding is possible to take place by defining corresponding getter methods in the DinoActor class:

1. Actor and target of same species and different sex
2. Both are adult dinosaurs
3. The female dinosaur is not pregnant - checked using hasCapability(PregnancyStatus.PREGNANT) - see below PregnancyStatus enum class

If all conditions are met, breeding takes place, otherwise null is returned.

**BreedingAction - extends Action**

* Should be returned in getAllowableActions method for all dinosaurs since it involves two actors interacting at adjacent squares
* When object is created, constructor takes in a DinoActor object as the target

In the execute method, use the static method in Probability class to generate a probability to simulate the chance of whether the female dinosaur will become pregnant.

If the female dinosaur should be pregnant, a setter method that will be in DinoActor, say *setPregnant* that takes in a boolean, should be called.

If true is passed in as an argument, the method will:

* Use addCapability() to add the PregnancyStatus.PREGNANT to the female dinosaur
* Initialize a pregnancyPeriod instance variable the female dinosaur

Either the female dino is pregnant or not in the end, breeding has occurred hence the execute method will return a descriptive message by calling menuDescription method

**Pregnancy and laying eggs**

Classes involved include:

**PregnancyStatus Enum class**

* Only one value, i.e. PREGNANT
* Meant to be used as a capability, added to the actor capabilities when an dinoActor gets pregnant using addCapability() method.
* Doing it this way helps us to **reduce number of instance variables** and **reuse existing functionality of the codebase**

**PregnancyBehaviour - extends Behaviour**

Required instance variables:

* pregnancyPeriod - represents the time left for it to lay egg

In this class’s getAction method, two things to model include:

* If a female dinosaur is pregnant but not due to lay egg yet
  + Decrement pregnancyPeriod and return null
* Time for female dinosaur to lay egg
  + Set the dinosaur as not pregnant using setPregnant method declared in DinoActor - see above BreedingAction section
  + Return a *LayEggAction*

We will determine whether the actor is pregnant by using a method defined in the dinoActor class that will return true if the actor has the capability of PregnancyStatus.PREGNANT (use hasCapability() method).

**LayEggAction - extends Action**

This class simply adds an Egg object to the location of the pregnant actor in the execute method. When creating the object, the actor’s display character is passed to Egg class’s constructor to be utilized by the object to identify what species of dinosaur will be born.

**Egg class - extends PortableItem**

Extends PortableItem since can be picked up by player and kept in inventory

Required instance variables:

* *waitTurn* - integer to keep track of the number of turns left to wait till egg hatches.
* *parent* - char type to identify the species of the parent by

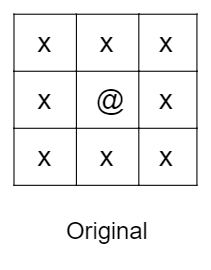
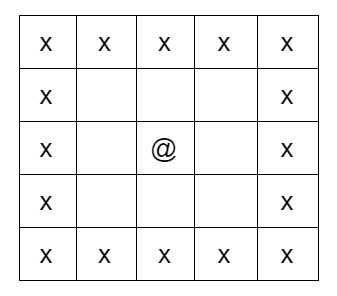
Two things to model in Egg class, both of which should be implemented by overriding the tick function, include:

* Waiting to be hatched
  + *waitTurn* initialized based on the type of dinosaur and will be decremented by one in the tick method until it reaches zero eventually
* Hatch
  + Detect if ready to hatch by checking if *waitTurn* has reached zero.
  + When ready, remove the Egg item from that location (using removeItem() from Location class) and add a DinoActor object corresponding to the species of the parent (using addActor of Location class). Ecopoints will also be added by calling the ecopoints.incrementEcopoints method.

**FollowMateBehaviour** **- extends Behaviour**

This class is modified from the FollowBehaviour class given in the codebase. The reason for doing so is that the original code was only able to detect actors from the immediate adjacent squares to be able to follow them - but when the actors are already beside each other, they can perform any interactions necessary, defeating the need to follow them.

Hence, in an attempt to **make the behaviour more meaningful**, we will allow the actor to detect and follow actors that are two squares away from it. To visualize this, images below illustrate a portion of the GameMap, where @ represents the actor and x represents the squares that it can detect other actors:

Original After modified

This can be done by utilizing the getExits() method appropriately on the immediate diagonal squares to the actor. In essence, we can define a method to get all the Exits that are two squares away, we will use the modify the getAction method to do the following:

For all exits, if another actor is detected on that location (with isAnActorAt method in GameMap)

* Check necessary conditions, namely both dinosaurs must be of the same species, different sex and the female dinosaur must not be already pregnant
* If all conditions are met, reuse a portion of the FollowBehaviour code to check which adjacent square the player has to move to in order to get closer to the target, and finally return a moveCloser action

If there are no exits that have an Actor and fulfills all conditions to be a potential target to follow, return null.

In effect, the FollowMateBehaviour can help a DinoActor to detect and follow other DinoActor two squares away that it can breed with.

Next page

**The “nature” part of the game:**

**Dirt** – **extends Ground**

The display character of Dirt is ‘.’. This class does not have any additional attributes and overrides only one method from Ground:

1. tick method

The role of this method is to let the Dirt experience the passage of time through updating the state of every piece of Dirt every turn. The Dirt can have two states – it either remains as a Dirt, or it grows a Bush.

To have the chance to grow into a Bush, it must not have trees on any square next to it. This can be done by using the input location to acquire the Exit, which is a List containing all the location of the adjacent squares, and check for their type. If there is an instance of Tree as we loop through all the exits, the loop breaks and the Dirt stays the same. This way of checking for the location of the surrounding squares is cleaner and reduces duplicate as we are reusing the methods that have been already provided for us.

If there are no Tree present in any of the adjacent squares, it will have the initial probability of 1% to grow a Bush. A higher chance of 10% to grow a bush can happen when there are two or more bushes surrounding the square. Therefore, as the loop checks for the presence of trees, it also keeps track of the number of bushes (if any) in any of the exits. The probability is generated randomly and will be introduced later on in the Probability class.

Since the number of loops will be known when the exits are checked and we will be directly working with the elements in the exits list, “for each” loops are used to ease the access of each elements as we do not need to bother with the indices. This reduces the chance of getting indexing errors.

**Bush – extends Ground**

Display character of the Bush is ‘~’ when it is still young and will change into ‘\*’ when it is fully grown. This class have two additional attributes which are the age, that is used to keep track of how old is the bush, and an ArrayList of bushFruits that stores Fruit item grown on the bush. Two methods are overridden in this class:

1. tick method

The role of this method is to let the Bush experience the passage of time through updating the state of the Bush every turn. Two situations can happen in this method.

The first thing to model is the Brachiosaur stepping on the Bush and having a 50% chance of killing it. At every turn, if there is a Brachiosaur actor on the location of the Bush (detected using getActor method of Location class), and if the probability meets the 50% chance (using the generateProbability method from Probability class to decide randomly), the Bush object of that location is replaced with a Dirt object using setGround method of Location class, simulating the death of the bush as the Brachiosaur walks on it.

If there is no Brachiosaur in the Bush square and the 50% chance is not met, the second situation happens. When the bush is not killed, it grows and, on every turn, the age attribute is incremented. Once it reaches the age of 10, it will be fully grown and is then capable of growing fruits. It has a 10% chance to grow a Fruit. The overall EcoPoints of the player will be incremented by 1 each time a Fruit object is created and the Fruit object is added to the bushFruit ArrayList.

If-else condition is used here so that the two situations do not overlap. ArrayList is used to store the Fruit object instead of Java array because ArrayList is mutable. A Bush can have unlimited number of Fruit ‘grown’ on it so it is only logical to use ArrayList instead of Java array.

1. decrementBushItem method

Removes the first item in the ArrayList when the method is called and returns the removed item. The item is removed using index rather than immediately removing the item itself because removal through index will automatically shift any subsequent elements to the left thus reducing the risk of index error. Removing the item in such a way reduces privacy leaks as the classes calling this method will not have access to the original list and will not be able to make changes outside of this method.

1. allowableActions method

Returns a list of Actions that is allowed to be done to the current Bush object. A SearchItemAction is added to the list to allow the action of the player searching for fruits from this Bush.

**Tree – extends Ground**

Display character of the Tree is ‘+’ when it is still young, the character will be ‘t’ and will change into ‘T’ when it is fully grown. This class have two additional attributes which are the age, that is used to keep track of how old is the bush, and an ArrayList of treeFruits that stores Fruit item grown on the tree. Two methods are overridden in this class:

1. tick method

The role of this method is to let the Tree to experience the passage of time through updating the state of the Tree every turn. At every turn, the age of the Tree object will increase by 1. Once it reaches the age of 10, the character ‘+’ will grow into ‘t’ and when the age of 20 is reached, it will change into ‘T’ which represents a fully grown tree. This also means that the tree can now bear fruits. A tree will have a 50% chance of bearing a fruit, a Fruit object will be created and when the 50% chance is reached the EcoPoints will increment by 1. Each of the Fruit item will be stored in the treeFruits ArrayList.

The Fruit that is now in treeFruits will have a 5% chance of falling to the ground. If that chance is reached, the Fruit will now be portable and added to the ground as an item. It will be removed from the treeFruits ArrayList as it is no longer on the Tree.

The length of time all Fruit that stay on ground will be monitored and increased every turn. Once the groundTime of the Fruit reaches 15, it will be removed from the ground to mimic the rotting of a fruit. If condition is used here so to make sure the Tree can only bear fruits when the age is over 20.

One note to be taken is that items are allowed to be added to the Ground. The reason for creating a new ArrayList instead of using the existing method to add the fruit to the ground is because a fruit can have two states, one is fallen to the ground, and one is still on the Tree (or Bush). One condition check will need to be done to differentiate the two states. Items on the ground can be of any type other than Fruit which leads us to a second condition check to differentiate the types of items. Therefore, separating the fruit on the ground and fruit on Tree/Bush into two different lists will be able to reduce the need for condition checks which results in a cleaner code and it is more maintainable.

**Fruit – extends Item**

This class have one additional attribute which is the groundTime that is used to keep track of the how long the Fruit has been on the ground. Excluding the constructor used to initialize the attributes, there are three methods implemented in this class:

1. setPortability method

As the name suggests this method is just to set the portability of the item. When a Fruit object is created, the portability will be false. This method is called only when the Fruit meets certain conditions and falls to the ground, the portability is then set to be true.

1. getGroundTime method

Returns the current groundTime of Fruit object.

1. tick method

The role of this method is to let the item on the ground to experience the passage of time through updating the state of the item every turn. groundTime is increased by 1 every time this method is called.

**Utility:**

**Probability**

This class only has one method:

1. generateProbability method

As the name suggests, it generates a probability. This is done through getting a random float number and check if it is less the input value. This method is declared as static as there is no need to create a new instance every time this method is needed. This results in a cleaner and less convoluted code. Having a class to specifically check for probability of whether certain code should run is useful in a way such that duplicate code can be reduced as it is needed throughout different classes.