MONASH University
Information Technology

FIT2099 Assignment 2: Revised Design Rationale

Team: **Tute03Team100**

Team members:

Student Name	**Student ID**
Tan Ke Xin	30149258
Marcus Lim Tau Whang	30734819

Introduction

In Assignment 2, our team has discussed and made changes according to the feedback given by the teaching team. With that, our application now includes some new functionalities (mainly new classes).

Object-Oriented Design Principles

Dinosaurs - the main stars of the assignment/game. We know that the dinosaurs have some additional features that a Player Actor does not have. For example, pregnancy status, maturity status (baby/adult), and breeding ability. Therefore, we've created a Dinosaur **abstract class** to function as a base for subclasses of each dinosaur type , i.e. classes Stegosaur, Brachiosaur and Allosaur. With this Dinosaur class, we could add additional features/attributes only particular to the dinosaurs instead of all Actor instances, while still maintaining the shared attributes of all Actor instances (e.g.: hitPoints, displayChar) among them. Also, the purpose of using an abstract class instead of an actual class is to prevent the direct instantiation of a Dinosaur instance.

By having this abstract class, we've successfully achieved the **'Reduce dependencies' (ReD)** design principle and also the **Dependency Inversion Principle** (of **SOLID Principle**). We now will have more flexibility in switching between / updating functionalities. Indirectly, **'Polymorphism'** is achieved as well, since we are now able to pass different data types to classes. This also promotes **code reusability**.

In addition to the above, we know that inheriting an Interface, which is a protocol/contract such that classes that implement this interface must also implement all methods in the interface, would serve the same purpose as extending a base class. However, we've decided that an abstract class would be better than an interface in this scenario. This is because we could create attributes of the dinosaur without being forced to initialize a value to them, and also immediately implement some concrete methods, eg getters and setters for each attribute in the base class(Dinosaur class). If we use an interface, the three dinosaur (Stegosaur, Brachiosaur and Allosaur) classes that inherit that interface would have to override all methods, making some methods eg the getter and setter methods appear in all three classes, which greatly defies the **DRY principle** (more explanation below) and make our code messy.

Besides that, by inheriting classes (parent-child relationship), e.g. : the three dinosaur classes inheriting Dinosaur and Actor class, Tree and Bush class inheriting Ground class etc, we can greatly reduce repetitive code for 1) methods that have similar functionality, 2) objects/instances that have similar attributes. Methods/attributes that are shared among parent(super) classes and children(sub) classes need only be implemented or declared once in the parent class. If we want to modify some functionality of the methods for the child class, then we could override the methods by changing the method signature and body; similarly if we want to add extra attributes in the child class, we could just add them in, without modifying the parent class. With this, we've just achieved the **'Don't Repeat Yourself' (DRY)** design principle. This also promotes **code readability** and makes updating or debugging our code easier, since any updates would only need to be done in one particular class, instead of changing it in every class that has that method.

Moreover, we've created a few actions classes, specifically to handle one action per class. For example, EatAction handles the process of eating, BreedAction the process of mating, SearchFruitAction the process of searching and picking fruit. With this, we've **increased cohesion** (**GRASP principle**) of our program, since the code in each of these classes is united and focuses on performing only one common task. This makes our program easier to maintain, extend and test because we could easily locate certain functionality. Classes like these also allow us to achieve the S in SOLID Principle, which is the **Single Responsibility Principle.**

Last but not least, we've also created a DinosaurGameMap and DinosaurLocation class, which extends GameMap and Location respectively. This allows us to make extensions to the game map and location, without modifying the source code, which is the GameMap and Location class. Therefore, we followed the **Open-Closed Principle** of the SOLID Principle. The usage of the Dinosaur abstract class as mentioned above also illustrates the following of this Open-Closed principle.

Game Design:

Quick view of all newly added objects and their display character:

Map Objects	Stegosaur	Brachiosaur	Allosaur	Vending Machine	Bush
Display character	d	b	a	$	v

Portable Items	Fruit	Stegosaur corpse	Brachiosaur corpse	Allosaur corpse	Egg	Meal Kit	Laser Gun
Display character	f	)	(	%	e	m	~

Dinosaurs *(dinosaurs package)*

Status Enum Class

An alive dinosaur could either be a baby or an adult, thus enumeration is used here. An Enum class named Status is created with values: ***BABY, ADULT.*** With this, we can easily know the maturity status of a dinosaur by calling **x.hasCapability(BABY)**/ **x.hasCapability(ADULT)** (where x is a dinosaur) and see whichever returns true.

Dinosaur (abstract) Class

In this Dinosaur abstract class, we have a count each for unconsciousness, pregnancy, and baby, to keep track of the number of turns of that status of this dinosaur. Each dinosaur would also have a gender represented by a M/F, an array list of Behaviour, and an indicator whether it is pregnant (true for pregnant; false otherwise). When creating a new dinosaur, it would have Wander behaviour and SearchNearestFood behaviour; its gender is randomly selected, it won't be pregnant, and its unconscious and pregnant count are set to 0.

Besides that, we've also created a method called **eachTurnUpdates**, which takes in an integer babyCount as input parameter (its usage shown below). As the name suggests, this method mainly handles these updates on each turn:

- If dinosaur is conscious: deduct a hit point (food level)
    - Else: increment its unconscious count
- If dinosaur is pregnant: increment its pregnant count
- If dinosaur is currently a baby AND its baby count is greater than/equal to babyCount: transform it into an adult (using the Enum Status.ADULT) and reset its baby count to 0
    - Else: increment its baby count

So as we can see, the input babyCount represents the maximum number of turns the dinosaur can stay as a baby, and since each dinosaur has different babyCount, we need to take it as input to make suitable changes on each dinosaur.

Lastly, this class also contains all **concrete getter and setter methods** to all attributes, so that classes that inherit this class wouldn't need to implement them once more.

Stegosaur, Brachiosaur, Allosaur Class

- **Constructor**

Each of these classes's constructor takes in an Enum as input parameter, to determine if the dinosaur instance being instantiated is an adult or a baby. All relevant attribute values (e.g. hit points) are initially set as the adult's initial values. Only after that would we check if the input parameter indicates that it's a baby, we overwrite the dinosaur's initial hit points, and also set the baby count to 1. The only exception is Allosaur class. Since Allosaurs only hatch from eggs, and can't be instantiated early in the game, therefore all its values are automatically set to the baby's initial values.

- Static ***dinosaur*Count** variable

Each of these classes would have a static variable, called ***dinosaur*Count**, where dinosaur is replaced by stegosaur/brachiosaur/allosaur. This static variable is mainly used to keep track of the number of each type of dinosaur instantiated, and used to be part of the name of the dinosaur, so that each dinosaur would have a unique name,eg Stegosaur1, Allosaur3. This is also the reason why it has to be static, so that each dinosaur instance of that class would share the same amount of dinosaurCount.

- **getAllowableActions**

In each of these classes, we've overridden the getAllowableActions method, to add suitable allowable actions for each dinosaur. All dinosaurs would be able to be fed by Player, therefore they would have a FeedAction. Only Stegosaurs can be attacked, (by Allosaurs or Player), hence Stegosaur class would also have an AttackAction allowed.

- **playTurn**

Methods such as eat(), breed() mentioned in Assignment 1 are all replaced by Action classes of their own, and are returned/handled in this playTurn method.

This playTurn method will be called each turn, and it is used to, in each turn, handle all updates of the dinosaur (this is when **eachTurnUpdates** method in Dinosaur

class is called), find all possible actions, and determine which action to be performed by the dinosaur.

The **priority of actions/behaviours** are as follows (most prioritised to least):

Lay egg → Breed → Attack (only for Allosaurs) → Eat (prioritise food that can fill up most hit points) → Follow (follow another nearby, same specie dinosaur to prepare for breeding) → Search for nearest food source → Wandering around → Do nothing

Each action/behaviour has its own class that inherits the Action class/implements the Behaviour class.

Also, however, dinosaurs that are unconscious can only do nothing and stay in its location, waiting to be fed by a Player. If it reaches a specific number of unconscious turns, it will die (DieAction is returned).

- Specific dinosaur class **extra explanation**

Brachiosaurs have a 50% chance to step on a bush and kill it. Hence, in the playTurn, if a Brachiosaur steps on a bush and probability is met, the bush location will be set to a Dirt instance, using **setGround** method from Location class.

For Allosaurs that can attack Stegosaurs, we've overridden the Allosaur class' **getIntrinsicWeapon** method, to return a new IntrinsicWeapon, which deals 20 damage for an adult Allosaur, and 10 for a baby.

Besides that, in the Allosaur class we can see a **ConcurrentHashMap** of <String, Integer> pair. Compared to a normal HashMap, a ConcurrentHashMap prevents the system from throwing a ConcurrentModificationException when we make changes to the hash map during the game. But more importantly, this hash map is used to store the name of the Stegosaur attacked by this current Allosaur, and the number of turns of cooldown, since we know that an Allosaur can't attack the same Stegosaur in the next 20 turns after an attack. We can tell that this update has to be made on each turn, thus the update would appear in Allosaur's playTurn method. It will check, for each Stegosaur in the hash map, if its cooldown turn =20, meaning now is the 20th turn, and if it is, remove the Stegosaur's name from it; else, just increase the cooldown count.

Bush, Tree, Fruit *(ground package)*

For **Dirt, Tree**, and **Bush** classes, we now added Capabilities(which is an Enumeration) to these classes. Since these capabilities are known and not going to

change, an Enum class named **Status** was created to indicate the Capabilities of Ground. The Status can be divided into:

- DEAD : unable to grow fruits
- ALIVE : able to grow fruits
- ON_TREE : fruits are on tree
- ON_GROUND: fruits dropped on the ground

Having said that, the proposed methods mentioned in Assignment 1 (e.g.:hasFruitOnGround(), removeFruit() and hasFruit() methods) are removed. To illustrate, **Dirt** class has a Capability of DEAD because of its inability to grow fruits, **Tree,Bush** classes a Capability of ALIVE because of its ability to grow fruits. While ON_TREE capability is added if the fruits are grown on a tree otherwise ON_GROUND status is added to indicate fruits dropped from tree to the ground, or basically any fruit that is on the ground.

In the given assignment specification, probability of growing a bush from dirt is 1%. Our team revised it to **0.5%.** This is because 1% is inappropriate since bushes will be growing at a very fast pace, leading to the possibility of having an imbalance game to increase.

Since some dinosaurs have limitations and are only able to eat from ground, we can simply check if the ripe fallen fruit is on ground with **x.hasCapability(ON_GROUND)** (where **x** is a Fruit instance), true will be returned if it is on the ground, otherwise false. This also shows that the proposed hasFruitOnGround() method is not needed at all, reusing the available code base will tackle the problem nicely.

Besides, to check if the tree has any ripe fruits and is ready to be eaten by a dinosaur, we can check the last item in the list of items on the current ground. Consequently, we check if the item has the Capability of being ON_TREE (e.g.: **x.hasCapability(ON_TREE)** where x is a Fruit instance). If the probability of ripe fruit falling from tree to ground is met, we can achieve this by: **x.removeCapability(ON_TREE)** (where x is a Fruit instance) and subsequently update it to on ground by: **x.addCapability(ON_GROUND)** (where x is a Fruit instance). Again, here shows that we reuse the available code in the engine and the proposed hasFruit() method is not required.

Once a fruit is eaten by a dinosaur, the EatAction will handle all the required processing. Therefore, removeFruit() method that suggested in Assignment 1, is being replaced with the **EatAction.**

To handle possible actions that might happen in class Bush, similar concepts are applied. Since fruits grown from bushes will only be on ground, **x.hasCapability(ON_GROUND)** (where x is a Fruit instance) will be used to validate.

Previously, our suggested solution is to use static variables named bushAlive or bushDead to check if the bush is stepped and killed by Brachiosaur. Now, our improved solution is to handle this situation directly in the **Brachiosaur's playTurn() method**.

Portable Items Classes *(portableItems package)*

Corpse Class

When a dinosaur remains unconscious for a certain number of turns (varies for each dinosaur), it will die and turn into a corpse (which is a portable item). A stegosaur's corpse is represented by a ')' in the game map, a brachiosaur's corpse is represented by a '(', lastly '%' represents an Allosaur's corpse.

In order to maintain a balanced ecosystem in this game, we've decided that **Stegosaur corpses will remain for 20 turns, Brachiosaur corpses 25 turns, and Allosaur corpses 30 turns.**

In our opinion, Stegosaur corpses should remain in the game for the shortest turns, as compared to the other two dinosaurs. This is because Stegosaurs grow fastest in the map: their eggs are laid fastest (only 10 turns needed), and they grow from baby into adult fastest as well (only 30 turns needed), as compared to the other two dinosaur types. As a result, shortening the duration of the Stegosaur corpse in the game map would prevent overly high chances for Allosaurs to find food, which indirectly prevents Allosaurs from mating and giving birth to more Allosaurs too often, thus giving an ecological balance in space.

In this game, the only way for Brachiosaurs and Allosaurs to die is when they fail to find any food supply, consequently become unconscious and then die after a fixed number of turns of unconsciousness (as compared to Stegosaurs, which could die due to hunger, or being attacked by Allosaurs or a Player with the laser gun). Having said that, Brachiosaur will turn into a corpse after 15 turns of unconsciousness while Allosaur needs 20 turns. We are also aware that eating a Brachiosaur corpse will fill up the Allosuar's food level to its maximum, while eating an Allosaur corpse will increase its food level by 50 only. Due to this, we should remain Brachiosaur corpse for a shorter number of turns. This is to prevent an Allosaur from always choosing to eat Brachiosaur corpses, as compared to Allosaur corpses, leading Allosaur corpses to be less useful (not bringing positive effect) in the game. So, with this proposed number of turns, we are able to ensure that the system is well-maintained.

However, it's also important that neither corpses of the three dinosaurs are left for too short periods of time. As corpses are one of the food supplies to Allosaurs which

are carnivores, this provides more (but not overly excessive) opportunities for Allosaurs to eat, leading the chances of them being unconscious (which will possibly lead to extinction) to be greatly reduced.

Egg & EggType Enum Class

In the Egg class, if it reaches a specific count, then the egg of the respective dinosaur will hatch and turn into a baby dinosaur. An appropriate message will be displayed on the console to inform the player that a dinosaur just hatched at the particular position on the game map.

Since an Egg instance could be either Stegosaur/Brachiosaur/Allosaur's egg, enumeration is also used here. An Enum class named EggType is created with: ***STEGOSAUR*****,** ***BRACHIOSAUR*****,*****ALLOSAUR*****.** With that, we can access the egg instance of Stegosaur simply by **x.hasCapability(EggType.STEGOSAUR)**.

Also, we've decided that Stegosaur eggs would hatch after 40 turns, while Brachiosaur eggs 30. Allosaur eggs will hatch after 50 turns, as given in the assignment specs.

MealKit & MealKitType Enum Class

Since a MealKit instance could be either Vegetarian or Carnivore, enumeration is also used here. An Enum class named EggType is created with:***VEGETARIAN*****,** ***CARNIVORE*****.** With that, we can access the MealKit instance by **x.hasCapability(MealKit.VEGETARIAN/CARNIVORE)**.

Consequently, the MealKitType Enum class will be used to add these Capabilities to the MealKit instance.

DinosaurGameMap & DinosaurLocation Class

We realized the mistake of using the given gameMap as we might provide our own implementation in the gameMap. Hence, to increase completeness of the game, add extra features (if any), and prevent making any edits that might crash the engine/system, we created new classes named **DinosaurGameMap** and **DinosaurLocation.**

Action *(actions package)*

All classes in this package inherits the Action class.

AttackAction class

AttackAction is created to handle situations where an Actor attacks another Actor; execute() method is overridden to have its own implementations. If an actor successfully attacks the dinosaur and the dinosaur's hit points reach less than/equal to 0, the dinosaur will die and turn into a corpse. Therefore, to increase code reusability, a call will be made to DieAction.execute() method to process the required changes. We've also added a line of code to heal the actor by the amount of damage done to the attacked target, to cater for the requirement of 'Allosaur increasing their food level by 20, each time it attacks a Stegosaur'. Moreover, by overriding the menuDescription() method, messages will be displayed on the console, informing the player which dinosaur was killed.

BreedAction class

BreedAction is implemented to determine which dinosaur should be the one that is pregnant. Having said that, we again override the execute() method to update the pregnancy count accordingly. This action will be called in each dinosaur's playTurn() method if the dinosaur is well-fed, not pregnant and found a mating partner near them. Besides, we also overridden the menuDescription() method to display a message stating which dinosaur is getting pregnant.

DieAction class

DieAction is created to handle cases where a dinosaur is dead, either its food level remains as 0(unconscious) for a certain number of turns, or it is killed. This action will be called in each dinosaur's playTurn() method. Here, we override the execute() method to perform required processing. To illustrate, if a Stegosaur dies from hunger, its display character on the map will be changed from 'd' to ')' which indicates a stegosaur's corpse. Subsequently, this stegosaur will be removed from the game map and its corpse will be added into the list of items on the current location.

EatAction class

EatAction has 2 constructors. This is because a dinosaur is able to eat by searching for a food source on its own in the game map or being fed by a player. Different constructors will be used to tackle situations mentioned: the constructor that takes in a List of Items indicate that the dinosaur eats by searching for food on its own; the constructor that takes only an Item, indicates that it'll be fed by the Player. This eat action will be called in each dinosaur's playTurn() method if they found a food source near them, or in the FeedAction class when Player decides to feed it. Once again, we overridden the execute() method to perform different eating actions. Since eating different items will increase the hit points differently, if-else checking is implemented here. By doing so, we can handle different cases nicely and display appropriate messages accordingly. For example, if a stegosaur ate from bushes or fruits on the ground, its hit points would increase by 10. The item eaten will then be removed from the game map and a message: "**StegosaurX ate a fruit on the bush or a fruit laying on ground under a tree.**" will be displayed to inform the player. However for cases where the dinosaur is fed by Player, the fed item will be removed from the Player's inventory, but this is handled in the FeedAction class.

FeedAction class

This class is used by the Player instance, to handle its feeding action and also the fed dinosaur's eating action (a call to EatAction). FeedAction's constructor accepts an Actor instance, which is the to be fed dinosaur, as input. The execution of this action would first display what the Player has in his/her inventory, then prompt a user input to indicate which food item the Player would like to feed. After that, it will check if the item is suitable to be fed to the dinosaur, eg herbivore dinosaurs can only eat fruits and vegetarian meal kit, and if it is, this action will remove that item from the Player's inventory, and call an EatAction for the dinosaur, with the food item as input parameter. After a successful feed, a message saying the Player fed which dinosaur, along with the output from the EatAction, would be displayed.
In addition, we've used a **do..while** loop, and a **try..catch** block in this class, so that exceptions that are thrown when there is invalid input, would be caught and not make the game crash, instead make the game keep on running until the user enters a suitable input.

LayEggAction class

An action class for a dinosaur to lay eggs. This class would look at the dinosaur's type (via its display character), and determine which egg to be laid in that dinosaur's location. The way to differentiate the type of dinosaur egg is explained here, using Enums. This class would also set the dinosaur's pregnancy status to be false, and its pregnant count back to 0, after laying an egg. Lastly, a message of the dinosaur name and the location of the egg laid is displayed in the output.

PurchaseAction class

PurchaseAction is called when the Player wants to buy items from a vending machine, whenever the Player is adjacent to a vending machine. When executed, firstly the Player's eco points would be shown, followed by the vending machine menu. Then, upon the user entering its option on which item he wishes to purchase, if the player has enough eco points to purchase the item, its eco points would be deducted that item's price amount, and a new instance of that item would be added into the Player's inventory. Again at last, the display would output a message indicating what item the Player purchased (if any).
Similar to FeedAction, we've used do..while, try..catch here as well to catch exceptions and keep the game running.

SearchFruitAction class

SearchFruitAction handles the action of Player searching and picking fruit on a bush/tree in the same square. Each time this action is called, a String input parameter is required to the constructor, indicating which plant type (a bush or a tree) is being searched for fruit. Searching for fruit has a 60% chance of failing, and we used Math.random() to handle it. An output message of whether the Player successfully searched and picked the fruit is displayed after each execution.

SearchNearestFoodBehaviour class *(game package)*

This behaviour class is a class that finds the nearest food source and moves the dinosaur actor one step closer to that food source. Its getAction method would return a MoveActorAction if a suitable food source is found for the actor, else it would return a null. This getAction method is also called from each of the 3 dinosaur classes (Stegosaur, Brachiosaur, Allosaur) in their playTurn.

Vending Machine class *(game package)*

A vending machine instance is placed on the map in (12,5). It extends the Item class, but is not portable. Its display character is '$'.

Laser Gun class *(game package)*

Extends WeaponItem class. A laser gun deals 70 damage, is portable, and displayed with a '~' character. Used by Player to attack Stegosaurs.