

Design Rationale

Group Name: KarHan

Group Members: Ng Wei Han, Ong Kar Kei

1. Grass and Fruit Implementation

1.1 Spawn Grass during beginning of the game

When **Application** class is first run, a **World** object will be created and it represents the entire game. This means that the initialization of actors and map will happen in the **Application** class. This also applies to spawning **Grass** objects during the beginning of the game. Therefore, this functionality is added into the **Application** class. How it works is that it will iterate each location of the map and check if it is a **Dirt** class. If the particular location is a **Dirt** object, there is a probability that the location will be replaced by a **Grass** object. Since **Grass** class is a child class of **Ground** class, the method *setGround(Ground ground)* in the **Location** class can be used to set the particular location with a certain **Ground** type.

1.2 Spawn Grass when Dirt is near Grass or Tree

Since **Dirt** class is a subclass of **Ground** class, **Dirt** objects can experience the time functionality in the game by overriding *tick(Location currentLocation)* from the parent class. How the grass spawning functionality works is that for each **Dirt** object, it will check for each location 2 boxes away from the **Dirt** object to see if there is a **Grass** object. If a **Grass** object exists, the method *calculateProbability(Int chances)* in the **Probability** object will be called. If the method returns True, a **Grass** object will replace the **Dirt** object. The similar functionality applies the same when a **Dirt** object is near a **Tree** object, whereby each **Dirt** object will check for each location 1 box away instead.

The new class **Probability** is created because in this game, there are many functionalities which require the system to generate randomly based on a certain percentage. Therefore, in order to avoid DRY principle and repeated codes, this new class is created so that it can provide the functionality to classes that wish to implement it. Furthermore, this design also obeys the design principle of "Classes should be responsible for their own properties". Other classes should not implement the probability functionalities because it is not their job. This ensures clear validity and the information needed is stored as close as possible to where it is needed.

1.3 Tree object can drop Fruit

Since **Tree** class is a subclass of **Ground** class, **Tree** objects can also experience time functionality in the game by overriding *tick(Location location)* from the parent class. This implementation of method overriding shows that this design provides an advantage of runtime polymorphism and provides specific implementation of a method declared in the parent class. The implementation works in such a way that for each turn played in the game,

Tree objects will use the method *calculateProbability(int chances)* in the **Probability** object to see if they can drop fruits. If a **Tree** object manages to drop **Fruit** object, the **Location** object will call the *addItem(Item item)* method. This is so that the **Fruit** can be picked up by the Player or eaten by Dinosaur. **Fruit** would also override the *tick(Location currentLocation)* method and it creates an int attribute called *age* which will increments by 1 when each turn. This is so that if a **Fruit** object stays on the ground for 20 turns consecutively and not in **Player**'s inventory, it would disappear.

Since there are many types of items, one of those being food items, a new abstract class **FoodItem** is created in which it inherits **Item** class. This allows the developers to accrue less technical debt in the future because the developers do not have to reclassify the items into different categories.

1.4 Harvesting Grass and Picking Up Fruit from Tree

In the design, two new classes in charge of the feature of harvesting grass are created which are **HarvestingGrassBehaviour** and **HarvestingGrassAction**. The **Player** class would have an array of **Behaviour** types which store the behaviours classes, which include **HarvestingGrassBehaviour** class. In **HarvestingGrassBehaviour** class, it would implement the **Behaviour** interface which then override its *getAction* method. The utilization of **Behaviour** interface gives an advantage such that the security of the implementation is achieved. In the implementation of the method, if the current location has a **Grass** object, **HarvestGrassAction** can be called. This gives an option for the **Player** to choose if they want to harvest the grass, which produces objects of a new class called **Hay** and stores them into their inventory. If the **HarvestGrassAction** is executed, the ground type of current location would be set to **Dirt** object again.

For the feature of picking fruit, two new classes are created which are **SearchFruitBehaviour** and **SearchFruitAction**. **SearchFruitBehaviour** object is also created and stored in the array of **Behaviour** field in the **Player** class. This implementation shows that a **Player** object can have multiple behaviours by using an array data type to store them, its behaviours are not hard coded. **SearchFruitBehaviour** object would return **SearchFruitAction** object if the current location of the **Player** is having a ground type of **Tree**. In the **SearchFruitAction** class, *calculateProbability(int chances)* of an **Probability** object would be used to determine if the **Player** has successfully picked the fruit and retrieved it into the inventory.

2. Hungry Dinosaur Implementation

2.1 Hunger implementations

To know if a Stegosaur is hungry, an Enumeration class named **DinosaurCapability** is made to record the current health status of Stegosaur using different Capabilities such as:

- ALIVE
- DEAD
- HUNGRY
- UNCONSCIOUS

When a Stegosaur object is created, the Capability ALIVE is added to the Stegosaur.

For the Stegosaur to feel hungry, an int attribute named **foodLevel** is first added to the Stegosaur Class. To decrease the foodLevel, a method that decreases the foodLevel of Stegosaur is implemented and is called in every playTurn of the Stegosaur. As a result, the foodLevel of Stegosaur will decrease by 1 for every round until it reaches 0 where the Dinosaurs become unconscious.

A new method that checks the status of the Stegosaur and adds or removes suitable DinosaurCapability to the Stegosaur is implemented in the Stegosaur Class and called in every play turn of the Stegosaur. The implementation is as following:

- When the foodLevel of Stegosaur drops below 30 and is above 0, a new Capability HUNGRY is added to the Stegosaur provided the Stegosaur did not have the Capability HUNGRY
- When the foodLevel of Stegosaur has increased above 30, the Capability HUNGRY is removed.
- When the foodLevel reaches 0, a new **Capability UNCONSCIOUS** is added to the Stegosaur.

When Stegosaur is unconscious, **DoNothingAction** is returned, Stegosaur cannot move or eat when it is unconscious. Also, since Stegosaur is supposed to die after 20 rounds of unconsciousness, a new class named **DeadActorAction** that inherits the Action class and overrides the *execute* method and *menuDescription* method is implemented to remove the Stegosaur from the map.

2.2 Eating and Feeding Implementations

**Assumption: Hay can be dropped can be eaten by Stegosaur from Ground*

To avoid the Stegosaur dying, two new classes named **EatFoodAction** that inherits the **Action** class and **EatFoodBehaviour** that implements the **Behaviour** interface is implemented to let the Stegosaur eat to increase its foodLevel. **EatFoodBehaviour** is added to the **Behaviours** array in the **Player** Class for the Player to execute when conditions are met. EatFoodBehaviour would return a **EatFoodAction** if the Stegosaur is currently standing on a grass by checking the location ground's type using the *getGround()* method or if the Stegosaur is standing on a fruit or a hay by checking the items that is laying on the ground at

the Stegosaur's location. This is based on the assumption that Stegosaur will eat the food whenever it is standing on the food regardless of being hungry or not.

Since Stegosaur's food can be of **Ground** or **FoodItem**, two constructors are needed for **EatFoodAction** where one constructor takes grass which is a Ground as the food to eat and another constructor takes an instance of **FoodItem** as the food to eat. **EatFoodAction** will check for which attribute (FoodItem or Ground) is not null and increase the foodLevel of the Stegosaur by the food's food points.

Food points of various foods for the Stegosaur are stored in a class named **FoodPoints**. Food points are stored in a *HashMap* as a value where the key is the food. This allows more food to be added to the *HashMap* in the future with convenience.

If Stegosaur is not standing on a food, it needs to move to a food source. Therefore, **MoveToFoodBehaviour** is implemented. **MoveToFoodBehaviour** implements the **Behaviour** Interface and overrides the *getAction()* method. This **Behaviour** will only be executed when the Stegosaur is hungry, that is when it has the **Capability HUNGRY**. It will check if there is any food next to the Stegosaur and find the nearest food source to the Stegosaur. **MoveToFoodBehaviour** will return a new instance of **MoveActorAction** with nearest food source **Location** and name of the **Exit** of the Stegosaur to the food as the arguments. Also, it is **assumed** that Stegosaur cannot move to the food source and eat the food in the same round. Stegosaur must first move to food source and eat the food in the next round.

For the Player to feed the Stegosaur, two new classes **FeedingAction** and **FeedingBehaviour** are implemented. **FeedingBehaviour** implements the Behaviour Interface and overrides its *getAction* method. **FeedingBehaviour** will first check if a Stegosaur is nearby the Player by checking if each **Exit** of the Player contains an Actor. If true, if Player's inventory contains a FoodItem Object, it will return a new instance of **FeedingAction** with the **FoodItem** Object and **Location** of Stegosaur as the arguments. **FeedingAction** will remove the **FoodItem** Object from the Player's Inventory and call **EatFoodAction** to enable the foodLevel of Stegosaur to increase accordingly.

The reason **EatFoodAction** is called in both **FeedingAction** and **EatFoodBehaviour** is to *reduce duplicated code*. This is because Stegosaur can eat food to increase its foodLevel through feeding by the Player or eating food on own choice. As the functionality is similar, there is no need to implement another class for increasing the foodLevel of Stegosaur when it is fed by Player. Thus, we have adhered to the "Don't Repeat Yourself" principle.

3. Breeding

3.1 Breeding Implementation

To indicate a Stegosaur is healthy enough to breed, a **Capability HEALTHY** is added to the **DinosaurCapability**. If a Stegosaur has foodLevel above 75 and hitPoints above 90, the **Capability HEALTHY** is added to the Stegosaur. To know the gender of the Stegosaur, a boolean named **male** is added as an attribute to the Stegosaur class and the user is required to indicate the gender when creating a Stegosaur, *true* if it is a male, *false* if it is a female. To indicate if a Stegosaur is pregnant, a boolean attribute named **pregnant** is added to the Stegosaur class and a setter and getter is created.

For the Stegosaur to breed, two new classes **BreedingAction** and **BreedingBehaviour** are introduced. **BreedingBehaviour** implements **Behaviour** Interface and overrides the *getAction()*. **BreedingBehaviour** will first check if the Stegosaur has the capability HEALTHY. If not, it will return null. If another opposite sex Stegosaur that is not pregnant is right next the Stegosaur, it will return a **BreedingAction** with the Stegosaur Actor and the Location of another opposite sex Stegosaur.

If the Stegosaur is not next to another opposite sex Stegosaur, **BreedingBehaviour** will check if there another opposite sex Stegosaur that is not pregnant two square away from the Stegosaur and if so, a new instance of **FollowBehaviour** with the opposite sex Stegosaur as the arguments. **FollowBehaviour** will be executed to find the nearest path to the opposite sex Stegosaur. Also, it was realised that **FollowBehaviour** and **EatFoodBehaviour** has some similar functionality that is, it both finds the nearest path to its target and returns it. To obey the **DRY** principle, a new Class named **CalculateDistance** is created to find the nearest path to a target and returns an **Exit**.

BreedingAction inherits the **Action** class and overrides the *execute* and *menuDescription* method. The probability of breeding successfully is set to be 20%. The *calculateProbability* method in **Probability** class is called to see if breeding is successful. If successful, **BreedingAction** checks which Stegosaur is a female Stegosaur and sets **pregnant** to be true. If not, it will return a String saying breeding has failed.

3.2 Dinosaur Egg and BabyDinosaur implementation

If a Stegosaur is pregnant, after 10 turns, it will lay an egg. Therefore, a class named **DinosaurEgg** that inherits the **Item** class is implemented. After 30 turns, the egg will hatch and a baby dinosaur should be added to the map. The gender of the babyDinosaur is chosen by using the method *calculateProbability* in **Probability** class. The baby dinosaur has an equal chance of being a male or a female.

A **BabyDinosaur** class that inherits the **Stegosaur** class is created. **BabyDinosaur** will grow up if it has a age of 30 and has **foodLevel** above 80 and **hitPoints** above 95. This condition is being checked for every play turn of the baby dinosaur in the playTurn method.

To let the baby Dinosaur grow up to be an Adult Dinosaur, a new class named **GrowUpAction** that inherits the **Action** class is implemented. **GrowUpAction** will first remove the babyDinosaur from the map and add an adult dinosaur (Stegosaur or Allosaur) to the map. The gender of the adult dinosaur will be the same as the gender of the baby dinosaur.

If the conditions for the babyDinosaur to grow up are met, **GrowUpAction** will be called with the **babyDinosaur** as the arguments.

The babyDinosaur has similar characteristics as an adult dinosaur except that it cannot breed.

It is assumed that the priority of Dinosaur behaviours is as following from highest to lowest:

1. **EatingBehaviour**
2. **MoveToFoodBehaviour**
3. **BreedingBehaviour**
4. **AttackBehaviour (for Allosaur)**
5. **WanderBehaviour**

4. Eco points and Purchasing

4.1 Introduction of **EcoPoint** in the game

A new class called **EcoPoint** is added into the system in which it contains an int attribute called *ecoPoint*. A new interface class called **EcoPointInterface** is created and the interface contains a **EcoPoint** object. Since the **EcoPoint** class contains a method called *addEcoPoint(int newPoint)*, other classes can implement the **EcoPointInterface** so that it can call the method and increase or decrease the existing eco points.

This feature is designed in such a way that only the classes which will increase the eco points will need to implement the **EcoPointInterface**. The implementation of the **EcoPointInterface** also allows classes to implement multiple interfaces as multiple inheritance is not supported in Java.

The classes which implements **EcoPointInterface** are:

- **BuyingAction** - Decreases eco points when items are bought in the vending machine
- **DinosaurEgg** - Increases eco points when a stegosaur or an allosaur hatches
- **FeedingAction** - Increases eco points when hay or fruit is fed to a dinosaur
- **Grass** - When grass grows each turn
- **HarvestGrassAction** - When hay is obtained by harvesting grass
- **Player** - Represents the total eco points a player is having right now

4.2 Purchasing feature

A new class called **BuyingBehaviour** which implements **Behaviour** interface is created. In this class, *getAction(Actor actor, GameMap map)* is overridden by only returning null value. This is because **Player** can buy many items, so there should be 1 or more than 1 actions to be returned. Therefore, a new method called *getMoreActions* is called in which it will return an **Actions** object instead. In the **Player** class, a conditional statement is added during the iteration of for loop in the *playTurn* method in such a way that if the **Behaviour** obtained is **BuyingBehaviour**, it will add all individual **Action** in the returned **Actions** object in the **Actions** object stored in **Player** class.

This design implementation utilizes previous features as much as possible as it is a hassle and bad design habit to design a new user interface just to allow **Player** to buy items from the **VendingMachine**.

4.3 Meal Kits and Laser Gun

VegetarianMealKit and **CarnivoreMealKit** are two new classes in which both inherit from **FoodItem** because they can be consumed by the **Dinosaur** objects. Since both meal kits have unique string representations, the hashmap attribute called *foodPoints* in the **FoodPoints** class will map String into Integer value.

LaserGun is a new class which inherits from **WeaponItem**. In order to activate the feature of **Player** whereby it can attack, **AttackBehaviour** class is created. In the **AttackBehaviour** class, if a **Player** is 1 box away from the **Dinosaur** objects, the **AttackAction** object will be returned. If a **Player** has a **LaserGun** object in the inventory, **Player** can attack other **Dinosaur** objects. An additional feature for **LaserGun** is also added in such a way that a **Player** can only carry and use **LaserGun** for 100 turns. This increases the difficulty of the game and brings in real-world logic of durability into the game.

5. Allosaurs

5.1 Dinosaur Abstract Class and Allosaurs

Since we have two different species of dinosaurs (Stegosaur and Allosaur) that have similar characteristics, a new abstract class named **Dinosaur** is implemented. The **Dinosaur** class inherits the **Actor** class and overrides its *getAllowableActions* and *playTurn* methods. A new class named **Allosaur** is created to represent the new species of dinosaurs that we have in our game. **Allosaur** class inherits the **Dinosaur** class and the **Stegosaur** class are modified to inherit the **Dinosaur** class. The methods previously implemented in the **Stegosaur** class are moved to **Dinosaur** class. The **Stegosaur** and **Allosaur** has their own constructor that is different from their super class and overrides the *playTurn* method. Both *playTurn* method in **Stegosaur** and **Allosaur** class calls the super method in the **Dinosaur** class. This can help us to reduce duplicated code in **Stegosaur** and **Allosaur** class which aligns with the **DRY** principle.

To record the different species of Dinosaur, a new *String* attribute named **species** is added to the **Dinosaur** class. The name of the species will be recorded when a **Dinosaur** is created. **Species** does not require input from the user when creating a **Stegosaur** or **Allosaur** Object as it has been set in the constructor of **Stegosaur** and **Allosaur**.

To avoid interbreeding between **Stegosaur** and **Allosaur**, **BreedingBehaviour** will check if the Dinosaurs are of the same species before returning **BreedingAction**.

Apart from that, **BabyDinosaur** is modified to inherit the **Dinosaur** class as well. Baby dinosaurs will be the same species as their parents.

5.2 Diets

As **Stegosaur** and **Allosaur** has different diets, new **Capability** named *HERBIVORE* and *CARNIVORE* is added to the **DinosaurCapability** class. When creating a **Stegosaur** object, it is automatically assumed that it is a herbivore and the **Capability** *HERBIVORE* is added to the **Stegosaur** Object. Likewise for **Allosaur** as a carnivore. **BabyDinosaur** will have either the **Capability** *HERBIVORE* or *CARNIVORE* added according to the species of its parents.

As **Allosaur** can consume dead corpse of **Stegosaur**, a new class named **Corpse** that inherits the **FoodItem** class is implemented. Also, since **DinosaurEgg** can be eaten by **Allosaur**, **DinosaurEgg** is modified to inherit the **FoodItem** class. **Corpse** and **DinosaurEgg** are added to the **FoodPoints** class along with its respective food points. At here, we assume that eating a **Stegosaur** Egg and eating a **Allosaur** Egg will give the same food points.

It is **assumed** that **Player** can feed **Allosaur** a *Vegetarian* food but there will be no increase in the **foodLevel** of **Allosaur** as **Allosaur** will not consume it. Also, **Player** can feed a **Stegosaur** *meat* but it will not consume it and thus, no increase in its **foodLevel**.

5.3 Allosaur Attacks Stegosaur

For **Allosaur** to attack **Stegosaur**, **AttackBehaviour** class that has been implemented previously has been added as a behaviour of **Allosaur**. Instead of modifying code in the `playTurn` method in the **Allosaur** class which will cause duplicated code as it now has different behaviours than **Stegosaur**, a new method named *AttackAbility* that returns a boolean that indicates whether an Actor has the ability to attack another Actor is implemented in the **Actor** Interface. **Player**, **BabyDinosaur**, **Stegosaur** and **Allosaur** needs to implement the method in each of their class. Since **Stegosaur** and **BabyDinosaur** cannot attack other **Actors**, *AttackAbility* will return false when it is called. **Player** and **Allosaur** are able to attack other Actors, thus, *AttackAbility* will return true when called.

The **AttackBehaviour** class is then modified to check if the **Actor** has the ability to attack. **AttackAction** will only be returned in the **AttackBehaviour** class if the **Actor** has the ability to attack.

AttackBehaviour is used by different **Actors** via *delegation* and is reused which reduces duplicated code in the system and possibly reduce work in the future if changes are to be made.

6. Quit Game

The user can choose to quit the game at every turn by inputting the hotkey. This is being done by the **removeActorAction** that was implemented previously. This helps to reduce duplicated code and improves the usability of classes.

