Package: actions

FeedAction(New)

- Extends Action

Attributes

- private Food: food
- private Dinosaur: target

Methods

- public FeedAction(Food food, Dinosaur dinosaur)
    - Sets food and target attributes
- public execute(Actor actor, GameMap map): String
    - Feed food item to dinosaur
    - Removes item from actor's inventory
    - Returns String displaying outcome
- public menuDescription(Actor actor): String
    - Description of action to display in the menu

How does this implementation follow good design principles?

Introducing an Action class for feeding Food objects to Dinosaurs follows the single-responsibility principle, as it ensures that the Player is able to feed the dinosaur without needing to know about the Dinosaur and increase dependencies.

How does this class interact with the rest of the system?

- Player uses FeedAction to feed Dinosaur a Food item

Alternative implementations considered

None. Initial idea was to not have this as a class.

Justification for changes from original design

- Created class

Separates the responsibility of the Player to feed the Dinosaur, which follows the single responsibility principle.

HarvestAction (New)

- Extends Action

Attributes

- private Ground: target

Methods

- public HarvestAction(Ground ground)
    - Sets target to ground
- public execute(Actor player, GameMap gamemap): String
    - Overrides method in Action abstract class
    - If target is a Dirt object with grass, calls harvestGrass() method to harvest the grass and turn it into hay
        - A Hay object will then be stored in the player's inventory
    - If target is a Tree object, calls harvestFruit() to try and pick a fruit
        - If successful, a Fruit object will be stored in the player's inventory
    - Returns String displaying outcome
- public menuDescription(Actor actor): String
    - Description of action to display in the menu

How does this implementation follow good design principles?

Introducing a separate class for the action of harvesting crops of any form adheres to the single-responsibility principle, as it ensures that classes representing the crops only need to implement the functionality of growing the crop itself.

How does this class interact with the rest of the system?

- Extends Action abstract class
- Allows a player to: try and pick fruit from a tree, and harvest grass for hay
- Instantiated in Player's playTurn method

Alternative implementations considered

None, as this class was created to revise our original design.

Justification for changes from original design

Originally, we planned to create a method in Player to implement all this functionality. However, we wanted to avoid having lengthy methods in Player for each relevant action. This design ensures that the Player class does not become bloated (i.e. too many responsibilities).

PurchaseAction (New)

- Extends Action class

Attributes

- private VendingMachine vendingMachine

Methods

- public execute(Actor actor, GameMap map): String
    - Calls vendingMachine.purchase() method to purchase an item
    - Once an item is successfully returned, it is added to the player's inventory
    - If unsuccessful, displays an error message and no item is purchased.
- public menuDescription(Actor actor): String
    - Description of action to display in the menu

How does this implementation follow good design principles?

Introducing an Action class for purchasing Items follows the single-responsibility principle, as it ensures that the Player is able to purchase items without needing to know about the VendingMachine and create dependencies.

How does this class interact with the rest of the system?

- Player uses PurchaseAction to purchase Item from VendingMachine

Alternative implementations considered

None. Initial idea was to let VendingMachine handle all the purchasing actions.

Justification for changes from original design

- Created class

Separates the responsibility of the Player to select an Item to purchase, which follows the single responsibility principle. Also removes the dependency of needing to know about the VendingMachine.

Package: behaviors

BreedingBehaviour

- Implements Behaviour interface

Attributes

- None

Methods

- public getAction(Actor actor, GameMap map): Action
    - Checks that actor is ‘eligible’ to find a mate (e.g. an adult and not already pregnant)
    - If there is a suitable mate within 3 tiles of actor, the method returns either a MateAction object (if the suitable mate is adjacent to actor), or an Action object from the FollowBehaviour class to move the actor closer to the suitable mate. If no such mate exists, returns null.
- private sameSpecies(Dinosaur dinosaur1, Dinosaur dinosaur2): Boolean
    - Checks that two given dinosaurs are of the same species
    - Called by the class’ getAction method
- private adjacentActor(GameMap map, Location location, Actor target): Boolean
    - Returns true if target is adjacent to location
    - Called by the class’ getAction method to decide if a MateAction object should be returned or not
- private suitableMate(Dinosaur dinosaur1, Dinosaur dinosaur2): Boolean
    - Checks if dinosaur2 is a suitable mate for dinosaur1 (i.e. same species, opposite sex and not already pregnant)

How does this implementation follow good design principles?

This adheres to the DRY principle, as both the Stegosaur and Allosaur class use the BreedingBehaviour class to implement breeding. This design choice also makes the system more modular, as breeding can be easily implemented for any number of new dinosaur species. The single-responsibility principle is also satisfied as the only reason this class should change in the future is if one wants to fundamentally change how actors breed with each other.

What does this class interact with in the system?

- Instantiated by the various subclasses (species) of Dinosaur

Alternative implementations considered

None, as this class was created to revise our original design.

Justification for changes from original design

We initially proposed for breeding functionality to be implemented within each subclass of Dinosaur. However, this meant that if any future modifications to breeding behaviour had to be made, one would have to go through every subclass of dinosaur and make this change which is not maintainable. Furthermore, this would also cause each subclass of Dinosaur to become overloaded; each Dinosaur subclass should only be responsible for *determining* the actions taken by a Dinosaur at each turn (rather than *implementing* it). Because of these reasons, a BreedingBehaviour class was needed to be created.

Package: actors

Player (Modified)

The only aspect of this class that was modified (addition of a few lines of code) is the playTurn method. This addition implements the requirement that a Player must be standing on grass or at the base of a tree in order to be able to harvest grass or fruit respectively. This functionality could not be achieved through the game engine.

Dinosaur (New)

- Abstract class
- Extends Actor class

Attributes

- private int: age
- private int: hungerLevel
- private int: daysUnconscious
- private int: daysUntilLay
- private Boolean: pregnant = false
- private String: sex
- private Behaviour: behaviour
- private Random: random = new Random()
- static final int: MATING_AGE = 30
- static final int: PREGNANCY_LENGTH = 10
- static final int: MIN_HUNGER = 0
- static final int: MAX_HUNGER = 100
- static final int: MAX_DAYS_UNCONSCIOUS = 20
- static final int: HUNGRY_THRESHOLD = 50

Methods

- public Dinosaur(String sex, String name, Character displayChar)
    - Default constructor for when game starts
        - Required so that there are two adult (age 30) Stegosaurus' of opposite sex at the start of the game
    - age set to 30 (by default)

    - hungerLevel set to 50 (by default)
    - daysUnconscious set to 0 (by default)
    - sex set by the argument passed by constructor
- public Dinosaur(String name, Character displayChar)
    - Constructor for when Egg hatches
    - age set to 0 (by default)
    - hungerLevel set to 10 (by default)
    - daysUnconscious set to zero (by default)
    - sex is set randomly between “Male” or “Female”
- protected getHungerLevel(): int
    - Returns hungerLevel of Dinosaur
- public getSex(): String
    - Returns String representing the Dinosaur’s sex
- protected isHungry(): boolean
    - Returns true if Dinosaur is hungry, false otherwise
- public increaseHunger(int hunger): void
    - Increases hungerLevel by specified amount
    - hungerLevel can never exceed MAX_HUNGER or go below MIN_HUNGER
- protected getDaysUnconscious(): int
    - Returns value of daysUnconscious attribute
- protected resetDaysUnconscious(): void
    - Sets value of daysUnconscious attribute to 0
    - Used whenever a Dinosaur has hungerLevel > 0
- protected incrementDaysUnconscious: void
    - Increments value of daysUnconscious attribute by +1
- private die(GameMap map): void
    - Creates a Corpse object and adds to the location of the Dinosaur
    - Removes Dinosaur from map
- public abstract layEgg(): Egg
    - To be overridden by subclasses of Dinosaur (different species lay different eggs)
- public isPregnant(): boolean
    - Returns value of pregnant attribute
- public isOppositeSex(Dinosaur target): boolean
    - Returns true if target is of opposite sex to this, false otherwise

- public mate(): void
    - Sets value of pregnant attribute to true and calls resetDaysUntilLay method
- protected noLongerPregnant(): void
    - Sets value of pregnant attribute to false
- protected resetDaysUntilLay(): void
    - Resets daysUntilLay attribute to its default value
- protected getDaysUntilLay(): int
    - Returns value of daysUntilLay attribute
- protected decrementDaysUntilLay: void
    - Reduces value of daysUntilLay attribute by 1
    - Used to keep track of how long a Dinosaur has been pregnant (and thus when it should lay an Egg)
- public isAdult(): boolean
    - Returns true of age of Dinosaur is >= MATING_AGE, false otherwise
- protected incrementAge(): void
    - Increments value of age attribute by +1
    - Used to age baby Dinosaurs so we know when they can start mating
- protected generalBehaviour(GameMap map, Location location): void
    - Called in the playTurn method of every Dinosaur subclass
    - Increments age by calling incrementAge method
    - Checks if Dinosaur is hungry (prints message if this is the case)
    - If the Dinosaur is pregnant, calls pregnantBehaviour method
- protected unconsciousBehaviour(GameMap map): Action
    - Called in the playTurn method of a Dinosaur subclass if that Dinosaur has hungerLevel equal to 0
    - Starts counter on days the Dinosaur has been unconscious
        - If this counter reaches MAX_DAYS_UNCONSCIOUS, dinosaur dies
    - Returns a DoNothingAction most of the time (since Dinosaurs that are starving cannot move)
- private pregnantBehaviour(GameMap map, Location location): void
    - Calls decrementDaysUntilLay method
    - Checks if the pregnancy period is over. If so, calls layEgg method and prints a message.
    - Called on pregnant Dinosaurs at every turn
- protected breedBehaviour(GameMap map): Action

    - Instantiates an instance of BreedingBehaviour and calls getAction method from the BreedingBehaviour class
    - Called on Dinosaurs that are not hungry, and therefore will be looking for a mate
- protected abstract lookForFoodBehaviour(GameMap map, Location location): Action
    - To be overridden by Dinosaur subclasses
    - Depending on the species, and whether or not they are a herbivore/carnivore, returns a suitable Action when the Dinosaur is hungry

How does this implementation follow good design principles?

Creating this as an abstract class follows the DRY principle, because this would allow creating subclasses that share attributes and methods between dinosaurs. Thus, we would not need to recreate the same methods for Stegosaurus and Allosaurus.

Furthermore, this design reduces dependencies by using constant static values (see 'Attributes' section). This abstract class also follows the Fail Fast principle, since we use MIN_HUNGER_LEVEL and MAX_HUNGER_LEVEL to ensure the dinosaur's hunger level does not go out of bounds. Furthermore, many of the methods in this class are declared private, as they are used as helper methods within the class. This maintains the readability of certain methods.

What does this class interact with in the system?

Stegosaurs and Allosaurs extend from this class.

Alternative implementations considered

Initially implemented dinosaurs that extends Actor without this abstract class, but this would violate the DRY principle (as all dinosaurs have common behaviour that can be grouped).

Stegosaur (Modified)

- Extends Dinosaur class

Attributes

- static final int: HUNGER_POINTS_FOR_GRAZE_GRASS = 5

Methods

- public playTurn(Actions actions, Action lastAction, GameMap map, Display display): Action
    - Calls generalBehaviour
    - Calls one of unconsciousBehaviour, breedBehaviour or lookForFoodBehaviour (depending on what the hungerLevel of the Stegosaur is)
    - Hunger level is decreased by 1 at each turn
    - If no Actions are available following this, then Stegosaur just wanders or does nothing
- public getAllowableActions(Actor otherActor, String direction, GameMap map): Actions
    - Overrides method in Actor class
    - Adds the following functionality: if a Player is standing adjacent to a Stegosaur, the Player can attack the Stegosaur or feed it (if the Player possesses a suitable Food item)
- private canEat(Item item): boolean
    - Returns true if Stegosaur can be fed the item, false otherwise
    - Stegosaur can only be fed Fruit, Hay and VegetarianMealKit
- public graze(Location location): void
    - Checks if on grass
    - If yes, then eats grass (calls removeGrass method) and hunger level increases by 5
    - Prints a message
- protected lookForFoodBehaviour(GameMap map, Location location): Action
    - Uses ScanSurrounds class to check if there is grass in sight
    - If so, uses the FollowBehaviour class to return an Action that moves the Stegosaur closer to the grass. If not, returns null.
- private layEgg(): StegosaurusEgg
    - Calls noLongerPregnant and resetDaysUntilLay methods
    - Instantiates and returns a StegosarusEgg

How does this implementation follow good design principles?

By extending the abstract class Dinosaur, this follows the Do not Repeat Yourself principle, because this Stegosaurus class inherits the methods belonging to the Dinosaur class.

What does this class interact with in the system?

- Stegosaurs graze on grass (which belong to Dirt)
- Player can feed Stegosaurs with Fruit, Hay, and VegetarianFoodKit
- Player can attack Stegosaurs
- Stegosaurs breeds with other Stegosaurs
- Allosaurs attack Stegosaurs (and eat their corpses)

Alternative implementations considered

Originally, the playTurn method was not broken up into helper methods (i.e. generalBehaviour, lookForFoodBehaviour, unconsciousBehaviour and breedBehaviour). This made the method difficult to read. Since Allosaurs exhibits most of the behaviour of Stegosaurs (except for the way in which they search for food), we decided instead to pull some of these helper methods up into the Dinosaur class (as to avoid reimplementing them twice).

Allosaurus (New)

- Extends Dinosaur class

Attributes

- static final int: HUNGER_POINTS_FROM_CORPSE = 50
- static final int: HUNGER_POINTS_FROM_STEGOSAURUS_EGG = 10;
- static final int: BITE_DAMAGE = 20;

Methods

- public playTurn(Actions actions, Action lastAction, GameMap map, Display display): Action
    - Calls generalBehaviour
    - Calls one of unconsciousBehaviour, breedBehaviour or lookForFoodBehaviour (depending on what the hungerLevel of the Allosaur is)
    - Hunger level is decreased by 1 at each turn
    - If no Actions are available following this, then Allosaur just wanders or does nothing
- public getAllowableActions(Actor otherActor, String direction, GameMap map): Actions
    - Overrides method in Actor class

    - Adds the following functionality: if a Player is standing adjacent to a Allosaur, the Player can attack the Allosaur or feed it (if the Player possesses a suitable Food item)
- private canEat(Item item): boolean
    - Returns true if Stegosaur can be fed the item, false otherwise
    - Allosaur can only be fed CarnivoreMealKit
- private eat(Location location): void
    - Used to check if Allosaur's Location possesses a Corpse or StegosaurEgg. If so, then the Allosaur eats the item (thus increasing their hunger level), and the item is removed from location. A message is then printed.
- protected lookForFoodBehaviour(GameMap map, Location location): Action
    - Uses ScanSurrounds class to check if there is a Corpse or StegosaurEgg in sight. If so, uses the FollowBehaviour class to return an Action that moves the Allosaur closer to the item.
    - If no such items exist, then ScanSurrounds is used again to check if there is a Stegosaur in sight. If so, uses the FollowBehaviour class to return an Action that moves the Allosaur closer to the Stegosaur. However, if the Allosaur is adjacent to a Stegosaur, then an AttackAction is returned.
    - If none of the above apply, returns null

How does this implementation follow good design principles?

By extending the abstract class Dinosaur, this follows the Do not Repeat Yourself principle, because this Allosaurus class inherits the methods related to Dinosaur.

What does this class interact with in the system?

- Player can feed Stegosaur with CarnivoreMealKit
- Allosaurs breeds with other Allosaurs
- Allosaurs attack Stegosaurs
- Allosaurs eat Corpses and StegosaurEggs
- Player can attack Allosaurs

Alternative implementations considered

Details for alternative implementations for Allosaurus is the same as the Stegosaurus class.

Package: ground

Dirt (Modified)

Attributes

- private Boolean: grass = false
- private Random: random = Random()
- static final int: POINTS_WHEN_HARVEST_OR_GROW_GRASS = 1
- static final int: GROW_CHANCE_AT_START = 2
- static final int: GROW_CHANCE_FOR_ADJACENT_TREE = 2
- static final int: GROW_CHANCE_FOR_TWO_GRASS = 10

Methods

- public hasGrass(): Boolean
    - Queries grass attribute of a Dirt object
- public tick(Location location): void
    - Executes only if grass is set to false
        - Calls chanceOfGrass() on whether the Dirt should grass
        - If chanceOfGrass returns true, then call growGrass()
- public allowableActions(Actor actor, Location location, String direction)
    - Instantiates Actions() class
    - Inserts HarvestAction()
- private chanceOfGrass(Location location): boolean
    - Calls ScanSurrounds.numberOfGrass() and ScanSurrounds.numberOfTree() methods to determine the chance of grass growing (scenarios with higher percentage considered first; if percentage is equal, prioritise the rule involving other terrain).
    - Then evaluates return boolean value accordingly.
- private growGrass(int probability): Boolean
    - When called, sets grass attribute to true
    - Change the displayChar attribute (to “^”)
    - Calls EcoPointsSystem.earn() method to earn points
- public harvestGrass(): Hay
    - When called, sets grass attribute to false
    - Change the displayChar attribute (to “.”)

    - Calls EcoPointsSystem.earn() method to earn points
    - Creates an instance of Hay and returns this (to the player)

- public removeGrass(): void
    - Change the displayChar attribute (to ".")
    - When called, sets grass attribute to false

How does this implementation follow good design principles?

The code follows good design principles of abstraction; the only thing that needs to be called is the tick() method to implement the functionality of growing grass. Again, the responsibility of checking the surroundings for a given Dirt object is delegated to the ScanSurrounds class.

How does this class interact with the rest of the system?

- After each turn, the game engine calls tick() on each Dirt object. The grass attribute may be set to true if it is currently false.

Alternative implementations considered

We considered creating a Grass class and instantiating Grass objects to exist in the same location as Dirt objects, but decided to implement a Dirt attribute instead for simplicity. With the existing game engine, you also can't store a Dirt and Grass object in the same location.

Justification for changes from original design

- Created more constant attributes to represent the chances of growing grass based on scenarios (adjacent tree, or two adjacent grass objects).

This was implemented because we did not realise that we needed these constant attributes. These were assigned as constant attributes of the Dirt class because having a 'magic number' will increase the number of dependencies which is a poor design choice.

- Changed chanceOfGrass() to evaluate the probability of growing grass itself and return the boolean value.

The way this method operated was changed since our original design caused a Connascence of Algorithm. This is because growGrass() would need to depend on the integer value returned by chanceOfGrass(), before growGrass() can evaluate whether grass will grow.

By moving the logic of evaluating whether grass will grow inside chanceOfGrass(), this will allow this method to return a boolean value instead. This will decrease the chances of bugs when adapting the code later (eg. passing an invalid integer output).

Tree (Modified)

Attributes

- private int: age = 0
- private final Random: random = Random()
- static final int: DROP_CHANCE = 5
- static final int: HARVEST_CHANCE = 40

Methods

- public tick(Location location): void
    - Calls increaseAge()
    - Calls dropFruit()
- public allowableActions(Actor actor, Location location, String direction)
    - Instantiates Actions() class
    - Inserts HarvestAction()
- private increaseAge(): void
    - Increases the tree's age
    - Based on tree's age, changes the displayChar
- private dropFruit(Location location): void
    - Using DROP_CHANCE and the Random class, calculates whether or not a Fruit object is initialised
    - If successful, call addItem() method in Location
- public harvestFruit(): Fruit
    - Uses Random class and calculates whether a Fruit can be found in a tree (40% success rate)
    - If successful, returns Fruit
    - If unsuccessful, print a suitable message

How does this implementation follow good design principles?

The code follows good design principles of abstraction; the only thing that needs to be called is the tick() method to implement the functionality of dropping fruit and removing dropped fruit that rots with the passing of time.

How does this class interact with the rest of the system?

- Fruits can be dropped onto the Location of the Tree
- After each turn, the game engine calls tick() on each Tree object. This can change the fruits attribute of Tree (in that a Fruit object may be added or removed if rotten).

Alternative implementations considered

Previously had an ArrayList<Fruit> attribute to keep track of the fruit that drops to the base of a tree (thus creating an association between the Tree and Fruit class). This was not necessary, because a dropped item can be stored in the Location class.

Justification for changes from original design

- Extracted code logic from tick() to create a method for increasing age

By creating a method for increasing the age of the tree, this would lead to having a more readable tick method and better maintainability overall, so that each method would contain only one responsibility.

ScanSurrounds (New)

This is not a subclass of Ground, but it has been included in this package because of its close relation to the Ground object, in that it helps in tasks such as determining the likelihood of grass growing on a square of dirt.

Attributes

- No attributes

Methods

- private static getLocationsWithin1(Location location): ArrayList<Location>
    - Returns ArrayList of all Location objects within one tile of location
- private static getGroundsWithin1(Location location): ArrayList<Ground>
    - Returns ArrayList of all Ground objects within one tile of location
- public static getLocationsWithin3(Location location): ArrayList<Location>

    - Returns ArrayList of all Location objects within three tiles of location
    - Calls getLocationsWithin1 as helper method
- public static adjacentTrees(Location location): int
    - Returns the number of Tree objects adjacent to location
    - Calls isTree as helper method
- private static isTree(Ground ground): boolean
    - Returns true if ground is Tree object using instanceof operator
- public static adjacentGrass(Location location): int
    - Returns the number of Dirt objects with grass attribute set to True (that are adjacent to location)
    - Calls isGrass as helper method
- private static isGrass(Ground ground): Boolean
    - Returns true if ground is Dirt object (determined using instanceof operator) with grass attribute set to true
- public static getStegosaur(Location location): Stegosaur
    - If there is a Stegosaur within three tiles of location, the Stegosaur is returned
    - Calls getLocationsWithin3 as helper method
- public static getGrass(Location location): Location
    - If there is a Dirt object with grass attribute set to True (within three tiles of location), the Location of this object is returned
    - Calls getLocationsWithin3 as helper method
- public static getLocationOfCorpse(Location location): Location
    - If there is a Corpse within three tiles of location, the Location of this object is returned
    - Calls getLocationsWithin3 as helper method
- public static getCorpse(Location location): Item
    - If a Corpse object is ‘stored’ in location, returns a reference to the object
- public static getLocationOfStegosaurEgg(Location location): Location
    - If there is a StegosaurEgg within three tiles of location, the Location of this object is returned
    - Calls getLocationsWithin3 as helper method
- public static getStegosaurEgg(Location location): Item
    - If a StegosaurEgg object is ‘stored’ in location, returns a reference to the object

How does this implementation follow good design principles?

ScanSurrounds is delegated (by various classes throughout the system) the task of querying the surroundings of a given tile. Though what you query for may may change, the way in which the taquerying is accomplished is largely the same. Having a dedicated class to do this means that individual classes/packages don't need to reimplement this functionality themselves. Thus, the DRY principle is satisfied.

How does this class interact with the rest of the system?

Used to assist classes with querying the surroundings of a given tile (e.g. to in turn determine chance of grass growing). This also includes obtaining references to or locations of a desired object (e.g. helps Stegosaurs search for grass when hungry)

Alternative implementations considered

Initially, this class was to be used only for implementing grass-growing.

Justification for changes from original design

We ended up adopting the 'alternative implementations considered' (stated in the previous version of this document), in that this class ended up being used to support functionality unrelated to grass-growing. The changes that were made to this class include more methods that assist in implementing the behaviour of Dinosaur subclasses. This was done for the reasons specified above.

VendingMachine (New)

A single VendingMachine object is to be added to the game map instantiated in the main() method of the Application class.

Attributes

- private String[]: items = { "Fruit", "Hay", "LaserGun", "StegosaurEgg", "AllosaurEgg", "VegetarianMealKit", "CarnivoreMealKit"}

Methods

- public purchaseItem(): Item
    - Prints the list of items for the user to select which item they want
    - If the item is spelled correctly and it exists in the items attribute, it should assign the new purchased item to a variable

    - The total number of eco-points is deducted from the cost of the purchased item
    - The new item is returned

How does this implementation follow good design principles?

Since this is a very small and isolated class, a discussion on design principles is not applicable.

What does this class interact with in the system?

- Players can purchase Item objects from VendingMachine

Alternative implementations considered

None.

Justification for changes from original design

- Changed items attribute to become an array of String

Initially we planned to have a HashMap<String, Item>, however this was not feasible as we cannot create a new instance of the Item by returning the Item value from the Hashmap. To solve this problem, we decided to create an array where purchasing will be handled by switch statements. While this is not very condensed code, this gave the program functional correctness, because now it is possible to select an item that exists in the vending machine.

- Changed purchase method logic to handle selecting an item

In conjunction with the key change above, this would mean that selecting an item would need to be handled by the VendingMachine's purchase() method. This would remove the Connascence of Algorithm, since passing in a String that depicts the Item desired is no longer needed. This means that selecting an item is now executed inside this method, without depending on another method passing a String.

At the same time, this change follows the concept of separation of concern, since selecting an item to purchase is no longer the Player's responsibility.

Package: portables

PortableItem (Modified)

Abstract class extending Item

Attributes

- private int: COST

Methods

- public getCost(): int
    - Returns value of COST attribute

How does this implementation follow good design principles?

By creating an abstract class for portable items, this creates a shared attribute and method between all portable items, because these can be bought from the vending machine. By doing so, we are applying the DRY principle.

How does this class interact with the rest of the system?

- Extends Item
- Abstract class for items that are portable

Alternative implementations considered

Initially implemented food items (that extend PortableItem) without this abstract class, but this would violate the DRY principle (as all food items have common behaviour that can be grouped).

Food (New)

Abstract class extending PortableItem.

Attributes

- private int: FILL

Methods

- public feed(Dinosaur dinosaur): void
    - Abstract method

How does this implementation follow good design principles?

Defines common behaviour (e.g. feed method) for all items that can be fed to actors in the game.

How does this class interact with the rest of the system?

- Extends PortableItem
- Acts as a “template” for items that can be eaten by dinosaurs

Alternative implementations considered

Initially implemented food items (that extend PortableItem) without this abstract class, but this would violate the DRY principle (as all food items have common behaviour that can be grouped).

Fruit (New)

- Extends Food abstract class
- Fruit objects in the player’s inventory will not rot because the tick() method is only called on fruit that is in a location

Attributes

- private int: age
    - Keep track of when the fruit rots away, once fallen onto floor
- static final int: FILL = 30
- static final int: COST = 30
- static final int: POINTS_WHEN_FRUIT_FED = 15
- static final int: ROT_AGE = 20

Methods

- public Fruit()
    - Set age to zero
- public getAge(): int
    - Returns the current age of the fruit
- public tick(Location location): void
    - Increments age attribute by +1
    - If age >= ROT_AGE, remove the item from location.items

- public feed(dinosaur:Dinosaur): void
    - Increases dinosaur's hunger
    - Calls EcoPointsSystem.earn() to earn points when fed

How does this implementation follow good design principles?

The class is isolated and self-contained, so it does not have any "knowledge" of other classes. Thus, discussion of design principles is not applicable.

How does this class interact with the rest of the system?

- Extends Food abstract class
- "Stored" as part of a Tree (when dropped on floor and not picked up) or Actor (when in inventory) object

Alternative implementations considered

None.

Justification for changes from original design

- Removed precondition check for feed() method only working on Stegosaur

This precondition check is no longer required, as calling the feed method is only possible through the new FeedAction class which can only be used by Stegosaurs. Hence, this precondition check is redundant and the code design becomes more maintainable, since this condition check will be handled by one singular class opposed to all the Food classes.

At the same time, this reduces technical debt, in the event this code needs to be changed again in the future.

Hay (New)

Extends Food abstract class.

Attributes

- private int: COST = 20
- private int: FILL = 20
- static final int: POINTS_WHEN_HAY_FED = 10

Methods

- public feed(dinosaur:Dinosaur): void

    - Increases dinosaur's hunger
    - Instantiates EcoPointsSystem and calls earn() method

How does this implementation follow good design principles?

The class is isolated and self-contained, in that it does not have any "knowledge" of other classes. Nor does it require a specific implementation; it just exists as another item that needs to be kept track of for the game. Thus, discussion of design principles is not applicable.

How does this class interact with the rest of the system?

- Extends Food abstract class
- "Stored" as part of Actor (when in inventory) object
- Hay is instantiated and added to inventory attribute of an Actor when an Actor decides to harvest grass

Alternative implementations considered

None.

Justification for changes from original design

- Removed precondition check for feed() method only working on Stegosaur

Similar to what was noted in Fruit, this precondition check is no longer required. Since the feed method is only possible through the new FeedAction class, which can only be used by Stegosaurs, it has become redundant.

MealKit (New)

Extends Food abstract class.

Attributes

- private int: FILL = 100

Methods

- public feed(dinosaur:Dinosaur): void
    - Abstract method. Code logic differs between its subclasses.

How does this implementation follow good design principles?

By creating this as an abstract class, this follows the Do not Repeat Yourself principle since subclasses can inherit shared attributes and methods. Furthermore, the feed() method as an abstract method follows the Do not Repeat Yourself method since this shared method would be inherited by subclasses.

What does this class interact with in the system?

- Because this is an abstract class, this can be extended by other MealKit subclasses
- Extends Food abstract class

Alternative implementations considered

Initially implemented meal kits (that extended PortableItem rather than Food) without this abstract class, but this would violate the DRY principle since all meal kits have common behaviour that can be grouped.

VegetarianMealKit (New)

Extends MealKit class.

Attributes

- private int: COST = 100

Methods

- public feed(dinosaur: Dinosaur): void
    - Increases dinosaur's hunger to FILL

How does this implementation follow good design principles?

By inheriting the MealKit abstract class, this follows the Do not Repeat Yourself principle since it will inherit the shared attributes and methods of MealKit. This also follows the Fail Fast principle because it checks the precondition that the dinosaur needs to be a Stegosaurus.

What does this class interact with in the system?

- VegetarianMealKit can be bought from VendingMachine by Player
- VegetarianMealKit can be fed to Stegosaurus by Player
- Extends MealKit class

Alternative implementations considered

None.

Justification for changes from original design

- Removed precondition check for feed() method only working on Stegosaur

Similar to what was noted in Fruit and Hay, this precondition check is no longer required. Since the feed method is only possible through the new FeedAction class, which can only be used by Stegosaurs, it has become redundant.

CarnivoreMealKit (New)

Extends MealKit class.

Attributes

- private int: COST = 500

Methods

- public feed(dinosaur: Dinosaur): void
    - Checks if dinosaur is Allosaurus, else throws error saying it can't eat this
    - Increases dinosaur's hunger to MAXIMUM_HUNGER_LEVEL

How does this implementation follow good design principles?

By inheriting the MealKit abstract class, this follows the Do not Repeat Yourself principle since it will inherit the shared attributes and methods of MealKit. This also follows the Fail Fast principle because it checks the precondition that the dinosaur needs to be an Allosaurus.

What does this class interact with in the system?

- CarnivoreMealKit can be bought from VendingMachine by Player
- CarnivoreMealKit can be fed to Allosaurus by Player
- Extends MealKit class

Alternative implementations considered

None.

Justification for changes from original design

- Removed precondition check for feed() method only working on Allosaur

This precondition check is no longer required. Since the feed method is only possible through the new FeedAction class, which can only be used by Allosaurs, it has become redundant. By doing this, the code has become more maintainable and reduces technical debt.

Egg (New)

- Abstract class
- Extends PortableItem

Attributes

- private int: timeAlive
- static final int: POINTS_WHEN_HATCH

Methods

- public tick(Location location): void
    - Increments timeAlive by +1
    - If timeAlive = 10, call hatch()
- public hatch(Location location): Dinosaur
    - Abstract method

How does this implementation follow good design principles?

By creating this as an abstract class, this follows the Do not Repeat Yourself principle since this creates a ‘template’ with shared attributes and methods for eggs. Therefore, any subclasses of Egg would inherit its attributes without having to rewrite in subclasses.

What does this class interact with in the system?

Since this is an abstract class, it does not interact with the system, but its subclasses (e.g. eggs for specified dinosaur breeds) will.

Alternative implementations considered

None.

StegosaurusEgg (New)

- Extends Egg class

Attributes

- private int: COST = 200
- static final int: POINTS_WHEN_HATCH = 100

Methods

- public hatch(): void
    - If timeAlive is equal to 10 turns, returns a Stegosaur object with age zero
    - Removes StegosaurEgg object from the system
    - Calls EcoPointsSystem.earn() and earns POINTS_WHEN_HATCH

How does this implementation follow good design principles?

By extending the Egg class, this inherits the shared attributes and methods of Egg. This means that attributes and methods don't have to be rewritten.

What does this class interact with in the system?

- StegosaurusEgg hatches into Stegosaurus
- Extends Egg class

Alternative implementations considered

None.

AllosaurusEgg (New)

- Extends Egg class

Attributes

- private int: COST = 1000
- static final int: POINTS_WHEN_HATCH = 1000

Methods

- public hatch(): void
    - If timeAlive is equal to 10 turns, returns a Allosaur object with age zero
    - Removes AllosaurEgg object from the system

    - Calls EcoPointsSystem.earn() and earns POINTS_WHEN_HATCH

How does this implementation follow good design principles?

By extending the Egg class, this inherits the shared attributes and methods of Egg. This means that attributes and methods don't have to be rewritten.

What does this class interact with in the system?

- AllosaurusEgg hatches into Allosaurus
- Extends Egg class

Alternative implementations considered

None.

LaserGun (New)

Extends WeaponItem.

Attributes

- private int: COST = 500

Methods

- public LaserGun()
    - Calls super()
- private getCost(): int
    - Returns value of COST attribute

How does this implementation follow good design principles?

The class is isolated and self-contained, in that it does not have any "knowledge" of other classes. Nor does it require a specific implementation; it just exists as a weapon to keep track of and potentially use. Thus, discussion of design principles is not applicable.

How does this class interact with the rest of the system?

- Extends WeaponItem
- When purchased from the vending machine, it is added to player's inventory
    - When instantiated, damage should be set to 100 so that it can kill a dinosaur instantly

Alternative implementations considered

None.

Package: game

Classes that do not belong to subpackages of game, but exist in game.

Corpse (New)

- Extends Item

Attributes

- None

Methods

- public Corpse()
    - Calls super()

How does this implementation follow good design principles?

By extending the Item class, this inherits the shared attributes and methods of Item. This means that attributes and methods don't have to be rewritten.

How does this class interact with the rest of the system?

- When a Dinosaur dies, it is removed from the current GameMap, and a Corpse object takes its place
- Allosaurs eat Corpses when hungry

Alternative implementations considered

None, as this class was created to revise our original design.

Justifications for changes from original design

Initially, corpses were not a separate class; when a Dinosaur died, a PortableItem would be instantiated to act as a corpse. This became an issue, as it meant that the Player could illogically pick up the corpse of a Dinosaur. Another issue was that it would be difficult for Allosaurs to distinguish between another PortableItem such as a Fruit. Giving corpses its own class (that extends Item) solves both of the above issues.

EcoPointsSystem

A single global EcoPointsSystem object will exist in the entire game. Players access this to spend points, and the engine will access this to earn points for the Player.

Attributes

- private static int: ecoPoints = 0

Methods

- public static getPoints(int points): int
    - Returns the eco-points that the Player can spend
- public static earn(int points): void
    - Called by other classes in the system for the Player to earn eco-points
    - Precondition: points must be positive integer
    - This method is only used to earn money
- public spend(int points): void
    - Called by VendingMachine to spend eco-points
    - Precondition: points must be positive integer
    - Postcondition: remaining ecoPoints cannot be lower than zero
    - This method is only used to spend money

How does this implementation follow good design principles?

This class follows the Reduce Dependencies principle since the Grass and Egg classes would not need to know about the Player class for the Player to earn points. The above preconditions and postconditions listed follows the Fail Fast principle, since this ensures that the game runs properly and debugging becomes easier.

What does this class interact with in the system?

- Eco-points are earned when Dirt grows grass
- Eco-points are earned when Player feeds Stegosaur with Hay or Fruit
- Eco-points are earned when Egg hatches
- Eco-points are spent by the Player for the VendingMachine

Alternative implementations considered

Initially thought of assigning an ecoPoints attribute to Player, but this made earning points when Dirt grows grass or Egg hatches very difficult. This is because there was no method in the game engine to return the Player at any point of the map.

Justification for changes from original design

- Converted all methods to be static

This allowed this class to have more functionality, since there is no point in instantiating a new EcoPointsSystem every time a method needs to be called. There is only one EcoPointsSystem ever created in the existence of one game, so it does not need to be instantiated. Furthermore, this allowed all classes to easily use this class without requiring to create dependencies.