Package: actions

FeedAction(New)

- Extends Action

How does this implementation follow good design principles?

Introducing an Action class for Player 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. This is different to the Dinosaur eating WildEdibles, because the code logic is different and this class is associated with the Player.

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

- Player uses FeedAction to feed Dinosaur a Food item

Justification for changes from previous design

- Created class

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

DieAction (New)

How does this implementation follow good design principles?

This class was needed since an Actor's playTurn() method requires an Action to be returned, whereas initially it returned a DoNothingAction. This was a problem because it was printing two statements in the program, one where the Dinosaur 'dies', and another where the Dinosaur 'does nothing'. By introducing an Action class for Dinosaurs dying, this resolves that issue and follows the single-responsibility principle.

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

- Dinosaurs use DieAction to die

EatAction (New)

How does this implementation follow good design principles?

Introducing an Action class for Dinosaurs to eat WildEdibles reduces duplicate code, as it encapsulates a common action of taking the Item from Location and eating it for the Dinosaur. This is different to the Player feeding the Dinosaur, since the code logic is different and the Player is not associated with this Action class.

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

- Dinosaurs use EatAction to eat Items from Location

DrinkAction (New)

How does this implementation follow good design principles?

Introducing an Action class for Dinosaurs to drink Water follows the encapsulation principle, because it groups the code logic behind drinking water in an Action subclass.

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

- Dinosaurs use DrinkAction to drink Water

HarvestAction (New)

- Extends Action

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?

- Allows a player to: try and pick fruit from a tree, and harvest grass for hay

Justification for changes from previous 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

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

Initial idea was to let VendingMachine handle all the purchasing actions.

Justification for changes from previous 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.

QuitPlayerAction

Extends Action

How does this implementation follow good design principles?

By extending Action, this prevents repeated code by inheriting the parent class's attributes and methods. This also introduces a single special Action for quitting the game, which follows the single-responsibility principle.

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

- Player can quit the game and return to the Application's menu

Package: actors

Player (Modified)

Further Clarifications on Attributes and Methods

- public playTurn(Actions actions, Action lastAction, GameMap map, Display display)
    - If Player is on top of a Dirt object with grass attribute, this allows Player to have HarvestAction. The reason why this is not initialised in Dirt.allowableActions() is because it would allow Player to harvest grass next to the block, but that contradicts the requirements.

How does this implementation follow good design principles?

The hasHarvestAction() method follows the abstraction principle, since a developer reading the playTurn() method would only need to know that the if statement's condition is met whether the Player's Location 'has a harvest action'. By extracting the complex condition logic out of the playTurn() method, this allows the method to be more readable, and therefore easier to maintain by other developers.

Dinosaur (New)

- Abstract class
- Extends Actor class

Further Clarifications on Attributes and Methods

- private isHungry(): boolean
    - Reduces duplicate code and reduces complexity in assessing whether the Dinosaur is hungry, in the playTurn() method
- protected canEat(Item item): boolean
    - Removes complex eating conditions from the getAllowableActions() method
    - Abstract method because it differs depending on type of DInosaur
- public boolean isOppositeSex(Dinosaur target): boolean
    - Removes complex conditions when assessing whether the Dinosaur can mate
- protected layEgg(): Egg

- Abstract method because a different type of Egg is returned depending on the Dinosaur

How does this implementation follow good design principles?

Creating this as an abstract class follows the DRY principle, because this would allow inheritance between dinosaurs that share attributes and methods. Thus, we would not need to duplicate repeated code.

Furthermore, this design reduces dependencies by using constant static values. 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.

Finally, extracting playTurn() logic into its own methods follows the abstraction principle, since the complex logic would be hidden away in its own methods and the playTurn() method becomes more readable. This playTurn() method was written in a way so that it can cover the general behaviour of Dinosaurs and can be inherited by other Dinosaurs to reduce duplicate code, then the other methods can be overridden to cover special behaviour which promotes Polymorphism.

What does this class interact with in the system?

- All types of Dinosaurs extend this class (Carnivore, Herbivore, Omnivore)

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).

Carnivore (New)

Extends Dinosaur class

How does this implementation follow good design principles?

By extending the Dinosaur class and creating a Carnivore abstract class, this allows all carnivore dinosaurs to inherit methods and attributes that are common between all carnivore dinosaurs. This will reduce duplicate code and be more maintainable, if this code were to be updated in the future.

What does this class interact with in the system?

- Inherits methods from Dinosaur
- Archaeopteryx and Allosaurs extend this class

Herbivore (New)

Extends Dinosaur class

How does this implementation follow good design principles?

Details for how this implementation follows good design principles is the same as the Carnivore class, except it allows all herbivore dinosaurs to inherit Herbivore.

What does this class interact with in the system?

- Inherits methods from Dinosaur
- Stegosaur extend this class

Omnivore (New)

Extends Dinosaur class

How does this implementation follow good design principles?

Details for how this implementation follows good design principles is the same as the Carnivore and Herbivore class, except it allows all omnivore dinosaurs to inherit Omnivore.

What does this class interact with in the system?

- Inherits methods from Dinosaur
- Agilisaur extend this class

Stegosaur (Modified)

- Extends Herbivore class

How does this implementation follow good design principles?

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

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 overwritten here. But this was moved to the Dinosaur parent class and was written in a way so that it covered all the functionality of the behaviour of a Dinosaur. This was so that this method can be inherited by all the Dinosaurs and reduce duplicate code, which promotes Polymorphism.

Allosaurus (New)

- Extends Carnivore class

How does this implementation follow good design principles?

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

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 Allosaur is the same as the Stegosaur class.

Agilisaurus (New)

Extends Omnivore class

How does this implementation follow good design principles?

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

What does this class interact with in the system?

- Using FollowBehaviour, if adjacent, can eat Corpse, Fruits, Grass, Hay and MealKits
- Allosaur can attack and kill Agilisaurs, but returns a small hungerLevel replenish
- AgilisaurEggs hatch into Agilisaurs
- Agilisaurs can breed with other Agilisaurs
- Players can attack Agilisaurs

Alternative implementations considered

Details for alternative implementation is the same as Allosaur and Stegosaur classes.

Archaeopteryx (New)

Extends Carnivore class

How does this implementation follow good design principles?

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

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

- Archaeopteryx can attack any subclass of Dinosaur except for its own
- Player can feed Archaeopteryx with CarnivoreMealKit
- Archaeopteryx' breed with other Archaeopteryx'
- Archaeopteryx eat Corpses and eggs of any subclass of Dinosaur except for its own
- Player can attack Archaeopteryx

Alternative implementations considered

Details for alternative implementation are the same as Agilisaur, Allosaur and Stegosaur classes.

Package: behaviours

BreedingBehaviour

- Implements Behaviour interface

Further Clarifications on Attributes and Methods

- The following methods reduce repeated code, and/or reduces the complexity of larger methods by extracting complicated condition checks.
    - private satisfiesMatingConditions(Dinosaur seekingDinosaur, Dinosaur targetDinosaur): boolean
    - private canBreed(Dinosaur dinosaur): boolean
    - private isSameSpecies(Dinosaur dinosaur1, Dinosaur dinosaur2): boolean
        - While it is bad practice to have ambiguous names like ‘dinosaur1’ and ‘dinosaur2’, however this method doesn’t care which dinosaur is seeking or is the target, therefore there is almost no difference between the two parameters. A variable name like ‘firstDinosaur’ was considered, but ‘dinosaur1’ was chosen because it was more concise and fits the job.
        - In a way, this follows the abstraction principle because a developer would just need to know to pass any Dinosaur, since there are no strict guidelines for the parameters.
    - private isOppositeSex(Dinosaur dinosaur1, Dinosaur dinosaur2): boolean
        - Reasons for ambiguous names like ‘dinosaur1’ and ‘dinosaur2’ have been explained in the method isSameSpecies() shown above.
    - private isAdjacent(Location location1, Location location2): boolean
        - Similar to isSameSpecies(), the two parameters do not care which Dinosaurs are on the Locations passed. There is almost no difference between the two parameters.
        - Similarly, this follows the abstraction principle because a developer just needs to pass any Location of a Dinosaur

How does this implementation follow good design principles?

This adheres to the DRY principle, as all types of Dinosaur uses BreedingBehaviour for breeding. Our methods are also extracted into its own methods to reduce complexity from the larger methods, therefore making the code more readable. The single-responsibility

principle is also satisfied as the only reason this class should change is if someone wants to change how actors breed with each other.

What does this class interact with in the system?

- All Dinosaurs use BreedingBehaviour to find a suitable mate

Justification for changes from previous design

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 bloated; and difficult to read. Because of these reasons, a BreedingBehaviour class was needed to be created.

FollowBehaviour (Modified)

Implements Behaviour.

Further Clarifications on Attributes and Methods

- public FollowBehaviour(Actor actor)
    - Allows Dinosaurs to follow a certain Actor
    - By overloading the constructor, Dinosaurs have flexibility in what they want to follow, which means more functionality. This ultimately means there is less repeated code, because there is no need to create a new class for Locations.
- public FollowBehaviour(Location location)
    - Allows Dinosaurs to follow a Location
    - Similar to the constructor above, overloading the constructor reduces repeated code since there is no need to create a new class for Actors.

How does this implementation follow good design principles?

By overloading the constructor so that the Dinosaurs are able to follow a certain Actor (either a suitable mate or dinosaur to attack) or a certain Location (grass, water or food item), this reduces repeated code since a new class for locating either doesn't have to be made.

What does this class interact with in the system?

- All Dinosaurs use FollowBehaviour to find potential Food/Water to replenish hunger or thirst level
- BreedingBehaviour uses FollowBehaviour to find suitable mate for Dinosaurs

Package: ground

Dirt (Modified)

How does this implementation follow good design principles?

The helper methods in this class reduce the complexity of the tick() method by extracting code logic into its own methods. This follows the single responsibility principle since each method would be delegated one task from the tick() method, this also makes code more maintainable and easier to read.

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 previous 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 previous 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)

How does this implementation follow good design principles?

Similar to Dirt, the helper methods reduce the complexity of the tick() method and are delegated one task from the tick() method. This follows the single responsibility principle and makes code easier to maintain.

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 previous 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.

Water (New)

How does this implementation follow good design principles?

By having this class extend the Ground class, the DRY principle is applied, since there is no need to rewrite all the attributes and methods that exist in the Ground class. Instead, these attributes and methods are inherited by the Water class.

The fact that we can implement the Water class with relative ease (simply by adding one more class without requiring any other changes to the rest of the system) demonstrates that the system adheres to the Open/Closed Principle.

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

- Dinosaurs drink from Water objects when they are thirsty
    - Like hunger, a message is displayed to the console when a Dinosaur is thirsty, or when a Dinosaur finds a Water to drink from
- The priorities of a Dinosaur are now as follows: curb hunger > curb thirst > breed
- Dinosaurs that cannot fly cannot traverse these tiles; dinosaurs that can fly can traverse these tiles
- Water objects are placed on the map at the very start and cannot be created during game execution

VendingMachine (New)

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

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

Justification for changes from previous design

- Changed items attribute to become an array of String

Initially planned to have a HashMap<String, Item>, however this was not feasible as we cannot create a new instance of the Item 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 has functional correctness, because now it is possible to select an item that exists in the vending machine. This class is an example that needs refactoring.

- Changed purchase method logic to handle selecting an item

In conjunction with the 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.

Package: items

PortableItem (Modified)

Abstract class extending Item

Further Clarifications on Attributes and Methods

- public getCost(): int
    - Won't need to duplicate getCost() for each subclass

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

WildEdibles (New)

This is an interface that is to be implemented by any classes in the item package that are considered "naturally attainable" by a Dinosaur (i.e. in the absence of intervention from the Player). For example, a Dinosaur can naturally come across an Egg (when two dinosaurs mate) or Corpse (when a Dinosaur dies).

Further Clarifications on Attributes and Methods

- getFill()
    - To be implemented How does this implementation follow good design principles?

How does this implementation follow good design principles?

In the execute method of EatAction (specifically the else block), the Item being targeted is cast to the WildEdibles interface. Note that we know that in the system's current state, the Item will either be an Egg or Corpse. However, Corpse and Egg classes have different parent classes, so the Item could not be cast to a class that accounts for Egg and Corpse. The interface therefore allows us to account for Egg and Corpse at the same time, without

having to specify the classes themselves. Thus, this implementation follows principles of abstraction (the EatAction class only deals with one interface rather than multiple classes).

What does this class interact with in the system?

- Method getFill() (the entire interface) is implemented by Egg and Corpse child classes

Food (New)

Abstract class extending PortableItem.

Further Clarifications on Attributes and Methods

- public feed(Dinosaur dinosaur): void
    - Abstract method common between all Items that can be fed to Dinosaurs

How does this implementation follow good design principles?

Defines common behaviour (e.g. feed method) for all items that can be fed to Dinosaurs.

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 (all Food item methods are encapsulated).

Fruit (New)

- Extends Food abstract class

How does this implementation follow good design principles?

By extending Food, this inherits attributes and methods that are common between Food items and therefore reduces repeated code.

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

Justification for changes from previous 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.

Hay (New)

Extends Food abstract class.

How does this implementation follow good design principles?

By extending Food, this inherits attributes and methods that are common between Food items and therefore reduces repeated code.

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

Justification for changes from previous 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.

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.

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.

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

Justification for changes from previous 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.

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.

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

Justification for changes from previous 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

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 it in subclasses.

What does this class interact with in the system?

- Extended by different types of Egg

StegosaurusEgg (New)

- Extends Egg class

How does this implementation follow good design principles?

By extending the Egg class, shared attributes and methods are inherited and not duplicated.

What does this class interact with in the system?

- StegosaurusEgg hatches into Stegosaurus

- Extends Egg class

AllosaurusEgg (New)

- Extends Egg class

How does this implementation follow good design principles?

By extending the Egg class, shared attributes and methods are inherited and not duplicated.

What does this class interact with in the system?

- AllosaurusEgg hatches into Allosaurus
- Extends Egg class

AgilisaurEgg (New)

Extends Egg class

How does this implementation follow good design principles?

By extending the Egg class, shared attributes and methods are inherited and not duplicated.

What does this class interact with in the system?

- Extends Egg class
- AgilisaurEgg hatches into Agilisaur

ArchaeopteryxEgg (New)

How does this implementation follow good design principles?

By extending the Egg class, shared attributes and methods are inherited and not duplicated.

What does this class interact with in the system?

- ArchaeopteryxEgg hatches into Archaeopteryx
- Extends Egg class

LaserGun (New)

Extends WeaponItem.

How does this implementation follow good design principles?

Inherits WeaponItem's methods and attributes which reduces duplicated code.

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

Corpse (New)

- Extends Item

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?

- Subclasses of Corpses inherit this class

Justifications for changes from previous design

Initially, corpses were not a separate class. When a Dinosaur died, a PortableItem would be instantiated to act as a corpse. This is a bug, because the Player could 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.

AgilisaurCorpse

Extends Corpse

How does this implementation follow good design principles?

By inheriting Corpse, this follows the DRY method and reduces repeated code.

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

- Inherits Corpse's attributes and methods

AllosaurCorpse

Extends Corpse

How does this implementation follow good design principles?

By inheriting Corpse, this follows the DRY method and reduces repeated code.

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

- Inherits Corpse's attributes and methods

ArchaeopteryxCorpse

Extends Corpse

How does this implementation follow good design principles?

By inheriting Corpse, this follows the DRY method and reduces repeated code.

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

- Inherits Corpse's attributes and methods

StegosaurCorpse

Extends Corpse

How does this implementation follow good design principles?

By inheriting Corpse, this follows the DRY method and reduces repeated code.

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

- Inherits Corpse's attributes and methods

Package: scanning

Scan (New)

Acts as the external class in the package that interacts with the rest of the system.

Further Clarifications on Attributes and Methods

The methods do not contain code logic and should only return the scanned objects. The code logic is stored in the other package-protected classes that are only accessed by Scan.

How does this implementation follow good design principles?

This class does not contain any code logic since it has (or will have) too many methods for scanning different objects. To increase readability, the code logic was extracted into separate classes to reduce the complexity of the class. The methods were separated based on the type of objects, this promotes encapsulation so that methods searching for similar types of objects (subclasses of the same parent) can be found in the same scanning class.

The Scan class is the only class in the scanning package to interact with the external system, so that all the scanning methods can be encapsulated and external methods can simply call this one class. Since the scanning methods are stored in package-protected classes, this promotes abstraction because a developer using this class would not need to know the internal logic of scanning and only know to pass a Location argument to return an object.

This class follows the Singleton design, which reduces dependencies from other classes creating an instance of Scan to use its methods. At the same time, this class follows the Composite design, which promotes both encapsulation and open/closed principle, so that more scanning methods can be implemented and organised easily without breaking existing code.

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

- All methods from ScanComponent subclasses are encapsulated here, so that external methods can simply use Scan for all scanning needs

Alternative implementations considered

Initially, all methods related to scanning were stored here and not separated into different classes. Because of this, the class became too large and difficult to read, even with

commenting. By separating the methods into their own respective classes, and converting the Scan class to simply query those classes, the Scan class became more simple to read and easier to extend.

ScanComponent

Abstract class that contains all the base methods related to scanning any type of object. This should be extended into subclasses to scan for different types of objects.

Further Clarifications on Attributes and Methods

- The following methods can be reused by subclasses (or this parent class) to reduce duplicate code and increase maintainability
    - protected static adjacentLocations(Location location): ArrayList<Location>
    - private static adjacentLocationsFromList(ArrayList<Location> locationsList): ArrayList<Location>
        - Prevents returning a list that stores the same Location multiple times
    - protected static adjacentLocationsIn3(Location currentLocation): ArrayList<Location>
- The following methods not only reduces duplicate code, but also removes the complexity of condition checking from subclass methods
    - protected static isGrass(Ground ground): boolean
    - protected static canSearchForEgg(Item item, Dinosaur dinosaur): boolean

How does this implementation follow good design principles?

By creating an abstract class that contains all the methods that can assist with scanning for Locations, Actors, Grounds and Items, this reduces duplicate code and acts as helper methods for subclass's methods to reduce the complexity of scanning for objects.

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

- Stores all the methods that assist with scanning
- ScanActors, ScanGrounds, ScanItems, ScanLocations extend this class

ScanActors

Extends ScanComponent

How does this implementation follow good design principles?

By extending ScanComponent, this class inherits all the helper methods that can assist with scanning for Actors, which reduces repeated code. Similarly to what was talked about in Scan, separating the scanning methods into classes that scan for the same type of object enhances readability and promotes encapsulation.

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

- External methods that want to scan for Actors can use this class

ScanGrounds

Extends ScanComponent

How does this implementation follow good design principles?

By extending ScanComponent, this class inherits all the helper methods that can assist with scanning for Grounds, which reduces repeated code. Similarly to what was talked about in Scan, separating the scanning methods into classes that scan for the same type of object enhances readability and promotes encapsulation.

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

- External methods that want to scan for Grounds can use this class

ScanItems

Extends ScanComponent

How does this implementation follow good design principles?

By extending ScanComponent, this class inherits all the helper methods that can assist with scanning for Items, which reduces repeated code. Similarly to what was talked about in Scan, separating the scanning methods into classes that scan for the same type of object enhances readability and promotes encapsulation.

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

- External methods that want to scan for Items can use this class

ScanLocations

Extends ScanComponent

How does this implementation follow good design principles?

By extending ScanComponent, this class inherits all the helper methods that can assist with scanning for Actors, which reduces repeated code. Similarly to what was talked about in Scan, separating the scanning methods into classes that scan for the same type of object enhances readability and promotes encapsulation.

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

- External methods that want to scan for Locations can use this class

Package: World

WorldFactory

Uses factory design pattern to return the desired World type

Further Clarifications on Attributes and Methods

- public static getWorld(String input): World
    - Based on user's selection, it will return the desired World
    - If this code will be extended in the future, an unknown worldName argument with a World that has not been implemented yet will return null.
- private static inputValidInt(String message): int
    - Purpose is to only accept a positive integer input, and continue looping until it does so. This is used for assigning the values for maxTurns and ecoPointsGoal for a ChallengeWorld
    - Since both maxTurns and ecoPointsGoal are required to be positive integer values, this allows this code logic to be reusable if another valid integer for World is needed in the future

How does this implementation follow good design principles?

By using a factory design pattern, we can create different types of game modes without needing to create an entire new codebase, which promotes Polymorphism. This means that implementing new Worlds will be easier, since integrating a new world (that implements a specific game mode) would simply be done through the WorldFactory.

Having a helper method called inputValidInt() to assist in accepting a positive integer value promotes abstraction. Since maxTurns and ecoPointsGoal does not care about the internal code of inputValidInt(), but it knows that it accepts a String parameter and returns an integer, then using the inputValidInt() method becomes really easy to use and extend.

Finally, by having the else code block in getWorld(), this will catch any unknown worldNames that are extended by a future developer. This promotes the Fail Fast principle, since the user interface would point out that their requested World doesn't exist.

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

- Returns a type of World based on user's gamemode selection in the Application

ChallengeWorld

Extends World

Further Clarifications on Attributes and Methods

- public run(): void
    - Difference in this method compared to the parent class is that it keeps track of the eco-points earned and turns remaining
        - Displays turns remaining and total eco-points for every turn
    - While this is considered repeating code, because the game engine is not allowed to be modified, we could not convert the World.run() method into an abstract method or reduce the number of lines in that method
    - We cannot use super() here, because it would be stuck in World.run()'s while loop
- private stillRunning(): boolean
    - Difference in this method is it returns false if the number of turns exceeded the maxTurns specified
- private endGameMessage(): String
    - Evaluates whether win or loss depending on number of eco-points earned
    - Prints String of number of turns and eco-points earned as well

How does this implementation follow good design principles?

By extending the World class, we can reuse its methods and attributes and therefore reduce duplicate code.

While ChallengeWorld.run() has some duplicate code from its parent class, because World.run() cannot be modified, the child class requires the duplicate code to display the eco-points and turns for every turn. If we could modify World.run(), then we would convert it into an abstract method, or reduce the method so that it can perform the execution before the while loop.

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

- Returned by WorldFactory based on user selection of their desired gamemode
- Literally the World for a Player who selects challenge gamemode

SandboxWorld

Extends World

How does this implementation follow good design principles?

While this class currently doesn't have many methods nor complex execution, by extending World, it allows new features of SandboxWorld to be implemented in the future. This makes SandboxWorld easier to maintain and extend, if needed in the future.

This also promotes abstraction, because we wouldn't need to rely on the internals of the World class if we were to use World for the sandbox game. If World was allowed to be modified, we would have created an abstract class or interface, so that we can promote object-orientation of inheriting similar attributes and abstract methods.

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

- Returned by WorldFactory based on user selection of their desired gamemode
- Literally the World for a Player who selects sandbox gamemode

Package: game

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

Application (Modified)

Further Clarifications on Attributes and Methods

- public static playWorld(World world): void
    - Adds a second game map (doesn't have a "building" with a VendingMachine, but has roughly equal numbers of Dirt, Tree and Water objects)
    - Second game map is added using private method addMapNorth
- private static addMapNorth(GameMap existingMap, GameMap mapToAdd): void
    - Allows the game designer to infinitely extend game maps in the northerly direction

How does this implementation follow good design principles?

The logic associated with adding a second game map to the north of the existing game map is extracted into a separate method in the Application class. Thus, should a game designer want to extend existing maps even more (e.g. add a third, fourth, etc. map), this can be done in one line (provided the game maps to be added are already initialised). Furthermore, the addition of this feature does not significantly impact the independence of this class; only one new dependency (towards Location) was introduced.

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

- Prompts the reader to select a game mode
- Sets the initial state of the game (with associated game maps and actors) and starts the running of the game

EcoPointsSystem

Follows the Singleton design pattern. 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.

Further Clarifications on Attributes and Methods

- private static int: ecoPoints = 0
- public static earn(int points): void

- public spend(int points): void

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 previous 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.