Inf1B - OOP

Assignment 3 Report

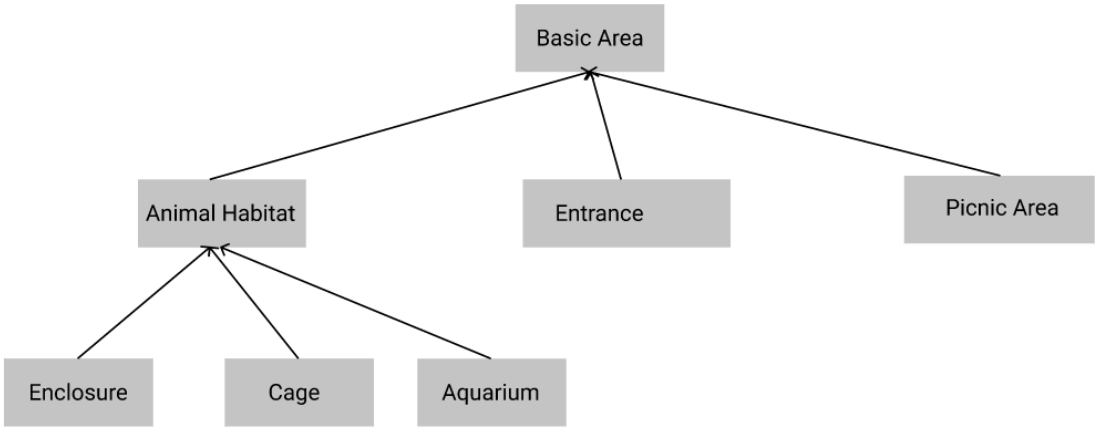
s2134605

DoS: 14th April, 2021

Beginner

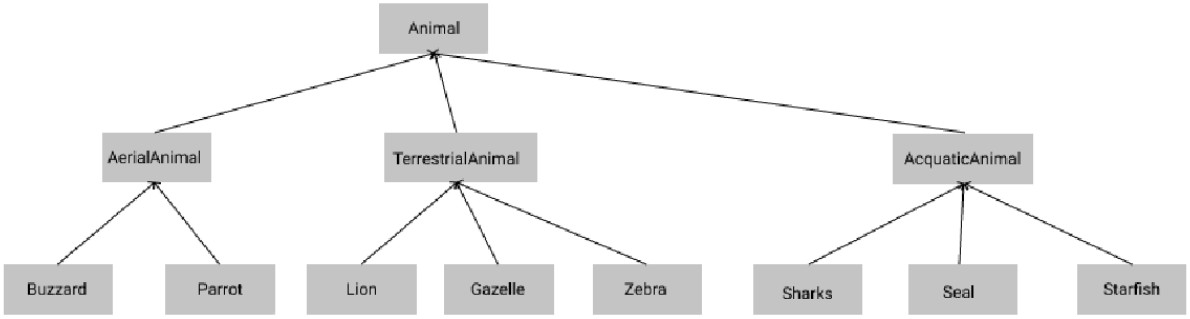
How Inheritance is used in the Project

Area ↓



*note that all classes implement the IArea interface

Animal ↓



In each of the domains (Area and Animal), each abstract class stores default methods and variables whose implementation is the same across all classes it is extended by and abstract methods for all methods which differ in their implementation

across sub classes. For example, BasicArea contains the ArrayList adjacentAreas, which every area needs to have, and AnimalHabitat has the animalsHere ArrayList, containing the list of all animals in that area, which needs to be either Enclosure, Cage or Aquarium. This avoids repetition and adds structure to the inheritance.

Enclosure, Cage and Aquarium extending AnimalHabitat coupled with the use of inheritance in the Animal domain (by having AerialAnimal, TerrestrialAnimal, AquaticAnimal, and an overarching Animal abstract class) helps in the implementation of the addAnimal method, especially where the program needs to check if the type of animal being added is compatible with the type of area and add an error if not, like so:

```
if ((animal instanceof AerialAnimal) && !(area instanceof Cage)) {

   errors.add(2);
```

rather than

```
if ((animal instanceof Buzzard || animal instanceof Parrot) && !(area
instanceof Cage)) {

   errors.add(2);
```

AND

```
if (areadetails.get(area) instanceof AnimalHabitat) {

...

}
```

instead of:

```
if (areadetails.get(area) instanceof Cage || areadetails.get(area) instanceof
Enclosure || areadetails.get(area) instanceof Aquarium) {

...

}
```

Beginner: Other Design Decisions

How the animals are stored

A given area's animals are stored in private Animal[] Array animalsHere of a length animalHabitatAreaCapacity, passed in by the client upon calling the constructor as an argument. Inside the constructor, setanimalHabitatAreaCapacity is called to set the capacity for that area. The advantage of using an Array is that it is immutable securing against more animals being added than the capacity allows.

Testing

Before tackling the implementation of method, I decided upon the logic using pen and paper, coming quite close to actual code in the process. While it certainly did not eliminate the need for testing, it did decrease the amount that had to be carried out.

What testing was done was performed by using the JUnit tests on CodeGrade and by creating a zoo in the main method, and performing operations on the zoo, like adding/removing areas/animals.

Intermediate

How the zoo's areas and connections were modelled

<u>How the areas are stored</u>

Areas are stored in a HashMap<Integer, Area> called *areadetails.* The HashMap data structure was chosen because:

- Hashtable stores key,value pairs :
    - Handy for storing the areaID (key) and the Area (value) in one data structure.
- Hashtable class contains only unique elements.
    - Useful since each area must have a unique areaID.
- Hashtable can't have any null values.
    - Useful since every area must have an ID associated with it and none of the IDs should be null.
- is Mutable
    - Allows the user to add and delete areas.

<u>How the areas can be manipulated</u>

- Adding Areas: addArea method must be called
- Removing Areas: removeArea method must be called
- Retrieving Areas: getArea method must be called
- Retrieving list of all keys (areaIDs): areadetails.keySet()
- Retrieving list of all values (Areas): areadetails.values()

<u>Alternatives for storing areas is either by using</u>

- 2 ArrayLists, one for areas, one for IDs

    - Not chosen because you need to access two separate ArrayLists for manipulating areas, cluttering and obfuscating the code.
- HashMap
    - Not chosen because it allows multiple null values and one null key. The Hashtable in this aspect adds another level of security, not allowing the addition of any null keys/values.

How the connections are stored

Since every Area needs a list of connections, they are declared in the BasicArea abstract class. Since connections can be added or removed, increasing or decreasing the length of the ArrayList, they are stored as an ArrayList<Integer> adjacentAreas.

How the connections can be manipulated

Connections can be added by calling the connectAreas method. Making a connection between area X and area Y essentially just means that:

1. X will be added to the ArrayList adjacentareas of Y
2. Y will be added to the ArrayList adjacentareas of Y

2 areas can be connected by calling the connectAreas(area1,area2)

Alternatives for storing connections

By using a normal array, we would be confined to a fixed length of connections. A conceptual way of thinking about connections

is to think of all the areas arranged in a circle and lines connecting different areas the paths/connections.

Problems with existing code

There are certain problems with the existing code that prevent a complete coverage of the specified functionality, in particular addAnimal(), visit(), connectAreas() and findUnreachableAreas(). Fortunately all cover the specified functionality to some extent although their coverage is not complete.

The addAnimal method has problems with code <3>, i.e. When the habitat is full, and for some reason, adds that as an error even when it's not the case.

The visit,connectAreas,findUnreachableAreas methods fail numerous JUnit tests on Codegrade, despite numerous modifications they continue to do so. The exact causes of the problems have not been identified.

Problems encountered

The only real problem encountered was the lack of any description of codegrade tests, so that it was often very difficult to make out what the tests were testing. Furthermore, since it was not possible to debug using codegrade and writing new test scenarios using main took a lot of time, debugging the code was quite difficult. Of Course in a professional setting a part of the development cycle, whether Agile or Plan Driven, and a part of the team would have been assigned to the task of writing tests, however, in this project this did account for all of the difficulties encountered

Advanced

How were money/prices represented

- Ticket prices were represented in two int fields, one for pounds, one of pence.
- The cash supply of the zoo is represented as a CashCount object, which stores the values for each denomination in a separate variable for each denomination. Getter and setter methods are provided for each denomination. An alternate way of storing these would be the use of a HashMap. This method was not chosen for the simple reason that a HashMap utilizes much more memory than the same amount of variables. Money in the algorithm is represented either as a CashCount or in pence or pounds and pence separately depending on the method in question.

The Algorithm & Logic

1. The money inserted (initially as a CashCount object) is converted to int pounds and int pence using the *ICashCountToPoundsAndPence*[1] method, and a new CashCount is created by casting the argument from ICashCount to CashCount.
2. Scenario Identification begins
    a. Is the amount of money inserted exactly equal to the ticket price? If so, add the money to the zoo's cashSupply and return an empty CashCount as change
    b. Is the amount of money inserted less than the amount of money needed to purchase a ticket? If so, simply return the cash inserted.
    c. Is too much money inserted? If so:
        i. Calculate change and store as the Pounds Change and the Pence Change.

ii. If the change is more than the amount of money in the zoo's cash supply, and hence can't be paid back, return the cashCount inserted.
iii. If the change is indeed payable, meaning the amount of money in the zoo's cash supply is sufficient to cover the change,
    1. pass the ticket price as the argument to the changeCalculatorAndOptimizer(3) and add that money to the cash supply.
    2. Pass the Pounds change and the Pence change to the changeCalculatorAndOptimizer and return that value.

1. ICashCountToPoundsAndPence: Converts a given cashCount into an equivalent amount of pounds and pence. Then, if the amount of pence is more than 100, calls on excessPenceToPoundsLogic to convert the excess into equivalent amount of pounds, and adds that amount to pounds. The final amount of pounds and pence is added to an Integer[] which is returned, with the Pounds as the first argument and the Pence as the second.
2. excessPenceToPoundsLogic: Given a number of Pounds and a number of Pence (both as Integers), converts any excess amount of pence (if > 100) into pounds.
    a. Sample function call with arguments:
        i. excessPenceToPoundsLogic(pounds: 15, pence: 350)
        ii. returns: [pounds: 18, pence: 50]
3. changeCalculatorAndOptimizer: Given a number of pounds and pence, converts them to equivalent in pence, then proceeds to calculate the optimized change (not returning everything in 10p coins) and returns the optimized change as a cashCount.

<u>Possible Improvements</u>

Currently the ticket price added to the zoo's cashSupply does not match the denominations entered. Essentially the program is not able to subtract the ticket price in an optimized way. One way to implement this is:

Pounds & Pence

For denomination in CashCount:

1. Check if the money inserted supply contains any of this denomination
    a. If not, move on
    b. If so, check how many times the denominations goes into the pounds part of the ticket price
2. Check if the money inserted cash supply at least that amount of that denomination
    a. If so, subtract how many times the denominations goes into the pounds part of the ticket price from the amount of that denomination we have
    b. If Not, subtract all of that denomination, reducing it to 0
3. Is the Ticket price in pounds 0?
    a. If not, move on to next denomination
    b. If Yes, move on to the Pence part of the Ticket Price