Informatics 1: Object Oriented Programming

Assignment 3 - Report

<S2084333>

April 11th 2021

1. Basic

- Areas

The different areas are split into two abstract classes which implements IArea interface: Habitat and NonHabitat. Class Habitat includes Cage, Aquarium, and Enclosure. Class Habitat includes Entrance and Picnic Area. Subclasses of class Habitat are required to input the maximum occupancy on initialization.

In this way of inheritance, it allows for habitat-specific methods to be added into their own abstract class like getting the maximum number of animals within the same habitat (getMax() abstract method). It also makes identifying whether an IArea object is a habitat easier in the addAnimal method instead of checking the type of an IArea object for each area type.

- Animals

Each animal class is inheriting the Animal abstract class. The animals have getHabitat() method which returns the correct habitat (area class name) of the animal as a string. This allows easy comparison when given an area and checking whether an animal fits within this area.

- Area IDs, IArea objects, and animals within

The unique Area IDs are generated by incrementing a counter starting from 0(The ID of the initial Entrance of the zoo), and a new area would have ID 1 and so on.

Removing an area takes away the ID and causes a gap left in between, but to keep keys incrementally consistent, removing one ID results in shifting every key after it backwards by one, which gets inefficient if a particularly long HashMap with lots of IDs is used for the zoo.

Data related to the Area IDs of IArea objects, the IArea objects themselves, and the animals inside areas are stored in a single HashMap “areas” that has the following type signature:

HashMap<Integer, HashMap<IArea, ArrayList<Animal>>>

which corresponds to:

{ID:{IArea object: [animals inside]}}

If the area is not a habitat, the size of the animal arraylist would be 0. This structure allows easy setting & getting of the referred data. Adding or removing an area is the same as adding or removing an ID which is easier to manage, the animal arraylist can be quickly referred and organized.

The areaInitializer method is created to pair a newly added area with an empty arraylist, thus, adding the area itself can be done with a single line:

areas.put(id, areaInitializer(area));

- Adding animals

Adding an animal to an area demonstrates the purpose of the choice of class inheritance and data structure of the program:

If the input area ID doesn't exist (if the areas HashMap's keyset contains this ID), which the program will return -1 if encountered.

If the input area ID doesn't correspond to a habitat. (if the area corresponding to the ID is an instance of NonHabitat class)

Similarly, if the area is an instance of Habitat class, the program would then check if the habitat's full or if some animal is not compatible.

If the input area ID's corresponding area doesn't fit the animal's habitat. (if the name of the area's class the same as the returned string of the getHabitat() method)

Since the data of every IArea object can be easily can quickly referred to using only the area ID because the way area classes inherited IArea interface, it makes the logic behind checking different possible errors more intuitive and natural.

2. Intermediate

- Zoo model and area connections

There isn't a definite model of the zoo. In the program's model, each area can be connected with any other existing area which allows complete freedom while creating the connections between areas. This allows a multitude of structures to be created by the user: a line, a loop, a tree, or a net.

The area model implements IArea interface with three methods: getAdjacentAreas(), addAdjacentArea(int areaId), and removeAdjacentArea(int areaId). Without the add and remove methods, the adjacent areas that is stored in the object has to be public and can be modified freely to meet the functionality, which defeats the purpose of a getter for the adjacent areas. They also makes the code more self-documenting as well as modularizing methods to declutter code.

The adjacent areas of an IArea object is stored directly into an arraylist of that IArea object. Getting the adjacent areas and adding or removing an adjacent area can be done with the interface methods. It has the following benefits:

Firstly, the connected areas can be found directly using the area ID, and made looping over a series of area IDs to find connected areas IDs of every ID simpler. Secondly, the connected areas is stored into a specific IArea object, which prohibits other areas to be connected with this object or vice versa without calling the method again with different conditions.

Lastly, by storing the connected areas in the object themselves, when only looking at a single IArea object, the connected areas of this object is still accessible, while some other options might not.

- Alternative representation

The zoo structure can also be represented with a HashMap e.g. HashMap<Integer, ArrayList<Integer>> where the key are the IDs and their values are arraylists of areas they are connected to. (example signature: {0:[1, 2], 1:[3], …})

In this implementation, although the connected areas are still easy to access and modify, but it has the following flaws:

- It is separated from the IArea objects themselves which makes area connections not accessible when only having the IArea object.
- It defeats the purpose of getAdjacentAreas() method, as the connections and the IArea objects are completely separated beside their IDs.
- It adds another class variable and needs to be initialized and separately treated when adding, removing areas and testing if a path is viable or finding unreachable areas.

Thus, although this alternative representation is viable, it is not the most practical considering the way IArea interface and its subclasses implemented.

3. Advanced

- Price representation

The price is represented using a HashMap with the following structure:

{“Pounds”: int x, “Pence”: int y}

It has a clear distinguishment between Pounds and Pence using string as their key, using only Integer type prevents precision issues caused by numeric operations with floating point numbers, easily accessible and modifiable, and each of them can be separately operated but saved in the same object.

It also makes comparing values and calculating the returning change simpler with only addition and subtraction, although unit conversion between Pounds and Pence is an additional factor that needed to be considered.

- Ticket machine algorithm

There is a method for converting an ICashCount object into {“Pounds”: int x, “Pence”: int y} structure (converted form) and it is a vital part in the algorithm.

Buying a ticket goes through the following process:

1. **Calculating change for the purchase**

Firstly, the changeAmount(ICashCount inserted) method handles calculating the amount of change and return it in its converted form.

This is done by converting ICashCount inserted to converted structure, then subtract entrance fee’s Pounds and Pence from the converted inserted amount, and change the unit if Pence is smaller than 0 to ensure Pence always stays bigger or equals to 0. It leads to 3 outcomes:

- *numbers of Pounds and Pence are both equals to 0*

This means the inserted cash is exactly the same as the entrance fee. A zero ICashCount object will be returned with zeroChange() method and the cash will be added to the machine. **The ticket purchase is complete.**

- *number of Pounds is smaller than 0 and Pence is greater than 0*

This shows the inserted cash is not sufficient to purchase a ticket, which the original ICashCount object will be returned and no cash added to the machine. **The ticket purchase is complete.**

- *The rest of cases where the total value of change is greater than 0 (e.g. 1Pound 70p, 0Pound 50p, 3Pounds 0p, etc.)*

This shows the machine will need to return some amount of change, thus, the cash will be first added to the machine before finding which combination uses the highest possible face value of notes and coins.

2. **Generate a “best fit” for the change**

The “fitting” would be done as follows:

Firstly, the Pounds section will be fitted with the fitPounds methods. It takes the amount of change in the converted form changeAmount method, and compare the Pounds value of the change to the face value of Pounds notes and coins:

1. While the number of Pounds notes in the machine with face value X is not zero AND the face value is smaller than the amount of Pounds change:

remove one such note from the machine, and increment a counter corresponding to such note by 1 in a list of counters representing the number of note used in the returning change.
Repeat this step for each notes with different face value from the highest to the lowest.

2. If there are any Pounds left that are not successfully fitted, convert them to Pence(multiply by 100) and add it to the number of Pence that needs to be fitted.

In this way, the highest possible face value of Pounds notes will be used to fit the amount of Pounds of the change. The leftover that are not fitted will be converted to Pence and fitted similarly as the Pounds in method fitPence. Lastly, fitChange() method uses the counters representing number of notes and coins and convert them into an ICashCount object.

3. Check the best fit

The output from fitChange method has 2 outcomes:

1. The best fit has a lower converted value than the calculated change

In this case, the notes and coins in the machine is not able to return the exact amount of change to return to the buyer, thus, the initial cash added into the machine will be removed, and the inserted cash will be returned. **The ticket purchase is complete.**

2. The best fit has the same converted value as the calculated change

In this case, the machine is able to find suitable notes and coins to return the change to the customer, it will return the fitted change. **The ticket purchase is complete.**