Informatics 1: Object-Oriented Programming

Assignment 3 - Report

<s2196231>

Basic – Design decisions

I used the abstract class of Area.java to model all the zoo areas, I implemented all three methods from the interface in this abstract class because these methods are common among all types of areas, implementing them here and making all areas its inheritance allows them to share using one piece of code in this class, therefore avoiding code duplication and providing a clear structure. Under the Area.java abstract class, I added another abstract class Habitat.java. In this class, I modeled all the common functionalities of animal habitats for the same reason as previously. I also defined an abstract method correctHabitat(), which is implemented separately in concrete subclasses of three different types of habitats as each kind allows different kinds of animals and their codes cannot be simply shared.

Intermediate – Modelling the zoo's areas and connections

All the zoo areas are stored in form of a HashMap with integer keys within Zoo.java class. While adding new areas to the zoo, after checking if the area has already been added or a second entrance (not allowed), then put it into the HashMap with a newly generated id key. Its integer key id is generated by finding the next smallest natural number not occupied by pre-existing areas. This way makes the code clean and readable, as well as the HashMap library, is pretty much designed for this kind of implementation, it is a good idea to use the well-tested and optimized solution in practice.
Inside Area.java, I used adjacency lists to store the ids of areas being connected to the inside of each area object. To connect one to another area, just add the id of the area to be connected to onto the adjacency list inside of the starting area. The interarea connection is represented in the form of internal adjacency lists because it is easy to implement and straightforward, as there is no need for displaying the grand view of zoo area connections or preference differences among paths so this method will be sufficient for this situation.
To check if a path is allowed, it first checks if all the areas on the path list are on the list of ids of current zoo areas. Then for each area on the list except the last, check if the following area is on its list of adjacent areas. If all of them passed, means the path is allowed.
As for checking unreachable areas, it follows the depth-first search order, visiting and recording all the areas an area is connected to and itself. Starting from the entrance it goes through all the paths one by one recording all the unvisited areas until there is none left, at the end after deducting them from the collection of the entirety of zoo areas, the remaining will be the unreachable areas.

Intermediate – Alternative model

An alternative for storing area ids can be an integer variable inside of area classes. I did not use this because the HashMap library is made for this kind of implementation, it works well and makes the code compact and readable. Also, IArea.java does not allow any modification so a lot of changes must be made to accommodate storing id inside individual area classes.
An alternative for representing the connections between areas is with an adjacency matrix, storing adjacency data all together inside of the zoo object instead of separate area objects. If there are weighting differences or the need for overall connection representation, the adjacency matrix method will be advantageous against the adjacency list method but that is not the case in this instance. As well as there is a getAdjacentArea() method defined in the given not-modifiable IArea.java interface, it is more reasonable to store adjacency relationship data inside each area object so each area object does not have to access Zoo.java, thus reducing overall importations and improving running speed.

Advanced – Money representation in *ICashCount*

As instructed by the given information, money is to be stored in form of CashCount.java objects. Inside the CashCount class, I used private int variables to store all the numbers of notes and coins. As there were already getters and setters defined in its interface class so it would only be reasonable if all the variables are not directly accessible publicly. As well as for security and robustness reasons, it is common practice to set up this way with private variables and public getters and setters. I chose int to store their values because int has well over enough capacity for money inside a zoo's ticket machine and Integers does not have the precision issue as floats and doubles as stated by the given instruction. An alternative representation will be using HashMap and storing each cash value and the amount together but doing this will increase the resource used on frequent calls and it does not reduce code duplication or improves readability.

Advanced – Key ideas behind the chosen algorithm

For the payEntranceFee() method, I chose the greedy algorithm, because works the fastest, and for the zoo ticket machine application it is good enough to handle most the situations. The likely hood of a ticket machine running out of changes would be very low as common ticket machines will be checked regularly collecting the income of the period and refilled. Also, usually, there are multiple ticket machines and service counters where visitors can also purchase their tickets, meaning even if one ticket machine stopped working the entirety of the zoo will be in trouble. Although there are other algorithms that can handle 100% of the situations like dynamic programming, it would significantly increase its time and space complexity, plus all the situations greedy algorithm cannot handle while it can is when the machine is running low on cash supply and using dynamic programming will only be able to sell like one more ticket and it too will stop working. As well as it is not significantly better at finding the fewest number of changes, dynamic programming does can provide the most optimal number of changes but comparing to the greedy algorithm, the difference will only be like one less coin and I do not think that will be much of a difference for the customers while what does annoys the customers is having to queue up and wait for hours in front of tickets machines just to get in the zoo. The sacrifice on performance will be too high on the limited hardware on ticket machines, speed will be a lot more important above all in this situation as the faster each machine runs, the fewer machines the zoo needs to purchase thus not only improving customer experience but also saves the cost. Overall, the drawbacks outweigh the improvements, I think the greedy algorithm is more suitable for this application.

It first checks the total value of the inserted cash, if not enough to pay the ticket it gives it back, if their values exactly match the entrance fee, it adds them onto the cash supply and returns zero cash. Then if none of the above, it creates a temporary cash bank of the current cash supply combined with the cash inputted this time and tries to find a change with this bank. If changes cannot be found, the inserted cash gets to be returned. For the change finding process, to achieve prioritizing large denominations and returning the least number of notes and coins possible, it starts finding change from notes with a large value in this instance notes of 20 pounds, the remaining amount of change to be found goes through floor division by the value of the selected cash type, and the resulting number of notes or coins gets added onto a provisional change CashCount object, and the remainder after this amount is being deducted from the total amount of change to be found goes through the same process with the second-largest notes or coins until finished for all the different cash types. In the end, if there is zero remaining change to be found, meaning this change has been found. Then the provisional change object gets outputted after deducting them from the bank. Afterward, the temporary bank becomes the new cash supply, marking the end of one cycle of entrance fee-paying. I used temporary objects in this process to ensure the current cash pool does not get mistakenly changed and simplify the return process as the original objects do not get contaminated if, in the end, the change-finding process failed. To return the least number of notes and coins, I decided to start from the notes with the largest value and work my way downwards because if the sum is constant, individuals with larger values will make the number count smaller.