Informatics 1: Object Oriented Programming

Assignment 3 - Report

S2029893

Basic – Design decisions

Inheritance was used in three packages (animals, areas, and zoo). In package animals, Buzzard, Gazelle, Lion, Parrot, Seal, Shark, Starfish, and Zebra are subclasses of Animal. In package areas, I created a new interface IHabitat. Aquarium, Cage, and Enclosure implement IHabitat and IArea. Entrance and PicnicArea are subclasses implement IArea. In package zoo, Zoo implements IZoo.

In package animals, I used key word “extends” to inherit those subclasses of animal which share the same traits from superclass Animal. In package areas, to distinguish those areas which can be habitat to animals from other areas which can’t, I created a new interface IHabitat which contains some methods only can be used for habitats like “isHabitat()” which indicate whether the input animal belongs to this habitat, “isFull()” which indicate whether this habitat is at its maximum capacity, “addAnimal()” which add the input animal to this habitat. Therefore, I used key word “implements” to make those area classes implement IArea or both IArea and IHabitat. In package zoo, I used key word “implements” to make zoo implement IZoo.

By using inheritance, we can easily construct these classes with same traits which provide us code re-usability and make the code with high cohesion. For example, Buzzard, Gazelle, Lion, Parrot, Seal, Shark, Starfish, and Zebra share the same traits since they are all animals. In addition, inheritance allow us to override the method. For example, the inner logic and return value varies in each subclass with the method isCompatible() in abstract class Animal, therefore, we can use inheritance to change the implement of each method for different animals. Moreover, as those classes of habitats inherit all traits of area and habitat, the use of interface allows us to let them implement both IArea and IHabitat, while they can’t inherit two super-classes at once.

Intermediate – Modelling the zoo’s areas and connections

I created an instance of HashMap “areaHashMap” to store area ID and area. Since each area in the zoo has a unique ID, therefore, the use of HashMap can let us easily store and retrieve an area which is the value of HashMap, and retrieve the area and remove the area by using area ID which is the key of HashMap.

To implement the connections between areas, I created an ArrayList “adjacentArea” for each area to store the ID of adjacent area. Therefore, the methods used for getting adjacent area, adding adjacent area, and removing adjacent area only need to operate on the ArrayList to implement the demanded functionality. For example, use method “add()” to add the adjacent area and “remove()” to remove the adjacent area. Furthermore, to connect one area to another area, we only need to add the area ID of the first area to the ArrayList “adjacentArea” of the second area.

Intermediate – Alternative model

We can also use a 2D ArrayList to store area ID and area. This ArrayList should have two columns and n rows, where n is the number of areas in the zoo. The first index of column represents area ID, and the second index of column represents area. To store and remove the value, we use method “add()”, and “remove()”. Despite this alternative model has the same functionality with the model I used, however, the operations of this model need more code to implement. Also, we can use method like “keyset()” to get the keys contained in the map and “clear()” to remove all the mappings from the HashMap, which is much more convenient than the operations of 2D ArrayList.

Intermediate – Issues encountered

Issues with findUnreachableAreas()

I have created a private HashSet called reachableAreas and a private void method called recurseFrom(int number) in the class Zoo. The HashSet is used for storing the reachable areas of entrance, and the method is designed to use recursion to recurse the adjacent areas start from entrance and finally add all the reachable areas of entrance to the HashSet "reachableAreas". The method findUnreachableAreas() should call method recurseFrom(0) and compare the HashMap "areaHashMap" which contains all the areas in zoo to return the areas in the zoo but not contained in "reachableArea".

However, when the method findUnreachableArea() called method recurseFrom(), it will rise ConcurrentModificationException. I think that is caused when an object has been concurrently modified by a different thread, but I don't know how to solve this problem.

Advanced – Money representation in *ICashCount*

First, I used key word "implements" to make class CashCount to implement ICashCount. Second, just like I used HashMap to represent the adjacent areas, I used HashMap to represent money. HashMap can easily store the different types of cash and their amounts as its keys and values. Once we need to adjust the amount of cash, we can use method "put()" to set the amount of specified type of cash, and once we need to know the amount of specified type of cash, we can use method "get". That is how the method "setNrNotes" (setNrCoins) and method "getNrNotes" (getNrCoins) were implemented. To initialise the class CashCount, I used a constructor, it initialized the HashMap "cash" by assigning every type of cash as keyset with value 0. This makes sure each instance of CashCount has the basic structure of cash and we can then operate on it.

Advanced – Key ideas behind the chosen algorithm

Since using floats or doubles to represent currency will cause some precision issues, I used BigDecimal to represent currency and the maths operation I used for any value related to currency is under the method of BigDecimal. To set the entrance fee, I created a private instance of BigDecimal called "entranceFee" in the zoo class. We can use method setEntranceFee to assign the value of entranceFee by adding the value of pounds with the value of pence divided by 100. To set the cash supply, I created an instance of ICashCount called cashSupply and initialised it in the method "setCashSupply()". Therefore, I used the method "getNrNotes_" to retrieve the amount of specified type of cash in the inputted instance of ICashCount and set the corresponding value for cashSupply using the method "setNrNotes_". The method getCashSupply() was implemented by returning the instance of ICashCount "cashSupply".

To implement the method payEntranceFee(), I created a supporting method called getCashValue() which allow us to calculate the total value of cash in an instance of ICashCount. Inside the method payEntranceFee(), I first clone the current cash supply to another instance of ICashCount to set the cash supply to the original one if the ticket machine failed to return the change, thus, the method adds the amount of money from inserted cash to cash supply. Therefore, we compare the value of cash inserted to the entrance fee, the cash machine returns the cash inserted and set the cash supply to the original one if its value is smaller than the entrance fee, otherwise, the cash machine needs to try for return the exchange. After that, we initialise the instance of ICashCount "cashReturned" which was used for storing the cash returned by the cash machine and calculate the value of exchange (I multiplied this value by 100 to do the following integer division), and we do the integer division of exchange with 2000 (which represents 20 pounds). If the value calculated is greater than zero and less than the amount of 20 pounds the cash supply has, we set this value as the value of 20 pounds in the cashReturned. Also, the amount of 20 pounds in the cash supply was subtracted by this value and the value of exchange was subtracted by this value multiply with 2000. We repeat the process above to check whether the cash machine need to return the cash for 10 pounds, 5 pounds… and finally, if the value of exchange becomes zero, we return the instance of ICashCount "cashReturned", otherwise, the cash machine is unable to return the change, therefore, we return the inserted cash and set the cash supply to the original one.