Informatics 1: Object Oriented Programming

Assignment 3 - Report

S2106632

April 2021

Basic :

- **Create the animal classes**

This is completed mainly by using inheritance. Since each of the animals stands for a particular type of Animal, I made them the subclasses of the abstract class Animal. Since every animal has to have a nick name, for the sake of being able to be called, it is set to be protected. GetNickName() should be overridden in each class to be a getter for nickName. IsCompatibleWith() should be declared individually since different species have different compatibilities.

- **Create the different areas**

I added each of the area and made them all implement the IArea as the instruction says. I also split them into two types: Habitat Areas(Aquariums, Cages, Enclosures), Non-Habitat Areas(Picnic Areas, Entrance), names are pretty much self-explained. And this constructs the hierarchy for the IAreas.

As the overall instruction says, each habitat must also have their maximum capacity upon creation, so I add variable maxCapacity in the Habitat Areas class to reduce the code duplication by utilizing the inheritance, and in the constructor of each subclass, they take in the capacity parameter to set it. The hierarchy of the classes mainly helps reduce the code duplication and make the structure clearer.

- **Create the zoo**

This step requires me to make a concrete class Zoo which implements the interface IZoo.

- **Add new areas to the zoo**

As I moved on, it requires to store the areas with unique ID. Since the IDs are unique, which is like the keys in hash map. So I decided to store the areas using hash maps. Remove area and get area can be easily implemented by the methods provided by hash maps. For generating IDs, I brought in a new integer variable id. Every time an area is added, the id will increase by one. For the addArea method, since there must be one and only one entrance and id of it is always zero, I put the entrance in as the object zoo is created, and there will be no chance to put another entrance in. After checking, it simply adds it using hash map methods put to add it in.

- **Add animals**

This method requires to check if the addition fulfills the rule and the error codes are given. Since we want to add animals to the areas, we first need to construct an array list to store the animals added; then, we want the smallest code to be reported first, we have to start checking cases with smallest codes.

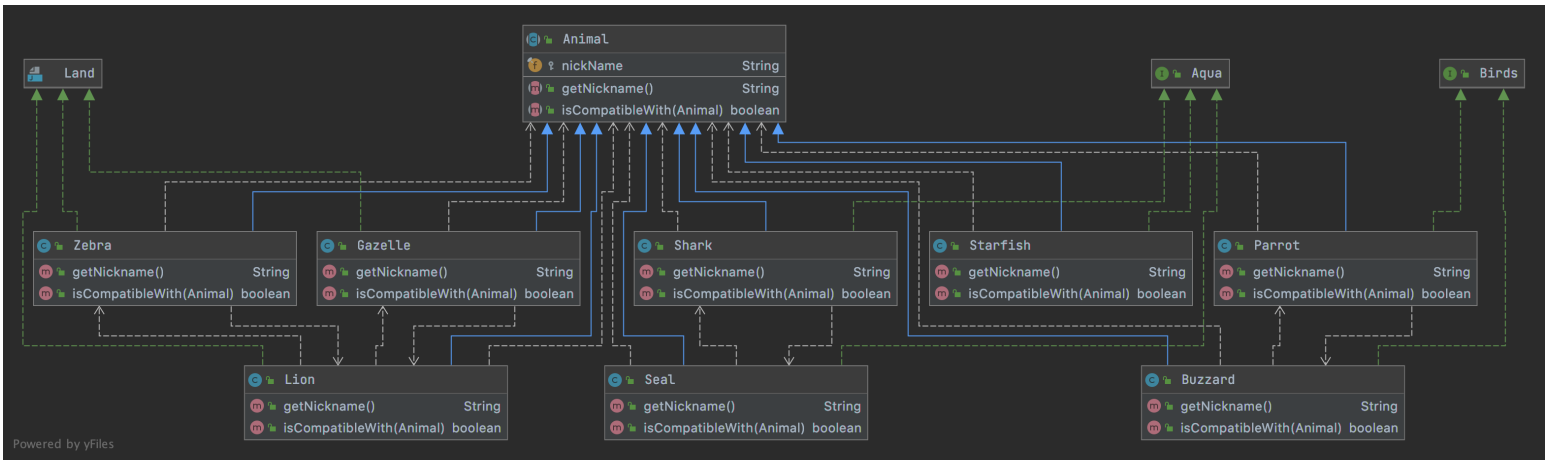
First is checking if it is a habitat area, and it's implements by instance of again. Since we have checked if the area is a habitat, it would be safe to cast area to HabitatArea after this step.

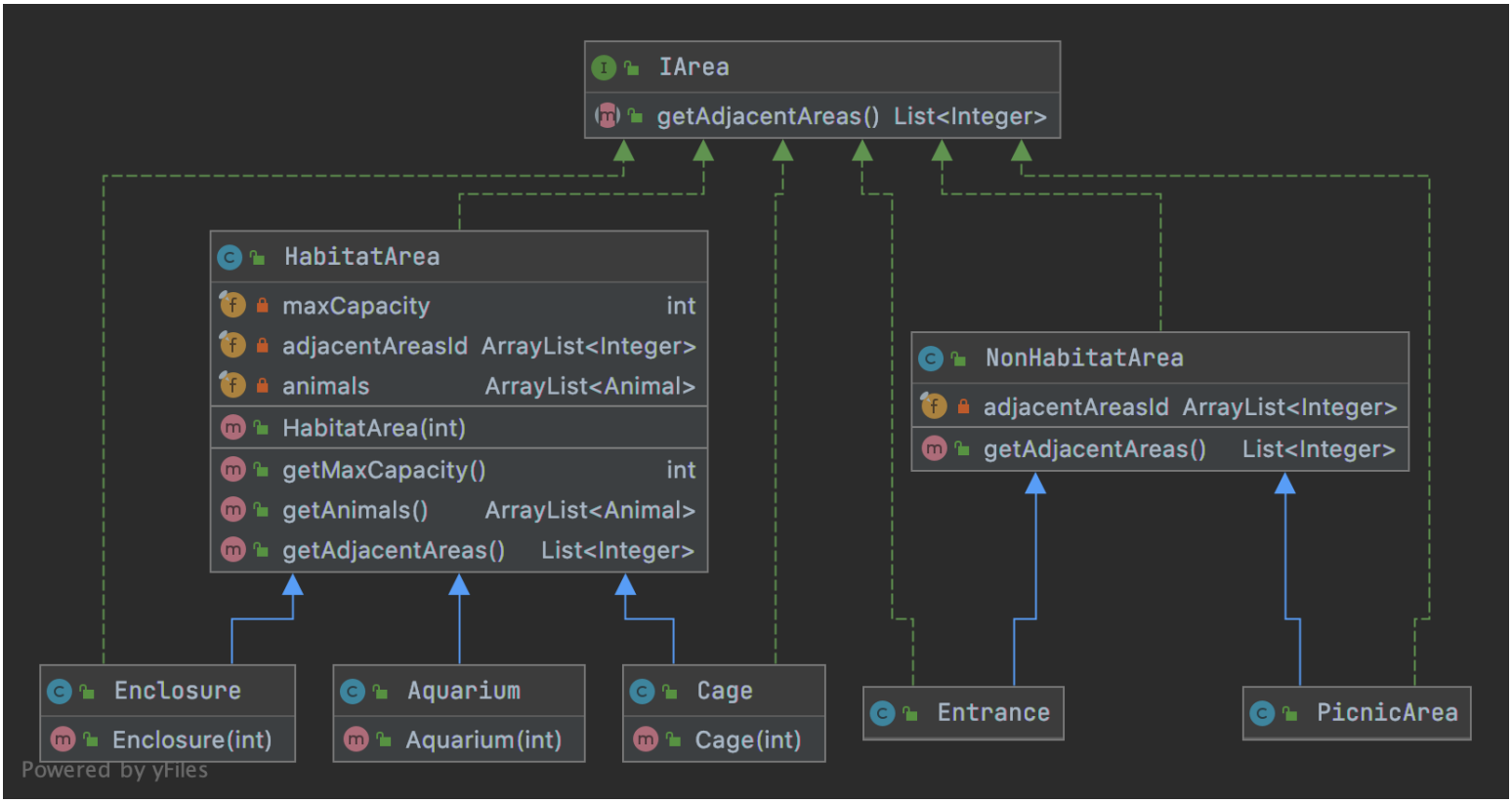
Second is wrong habitat, since animals can be split in land, birds, and aqua by habitats, I add intermediate Interfaces(Aqua, Birds, and Land) to help make it clear and we just need to check if animal types conform to their habitats.

Third is when the habitat given is full, then it's impossible to add it in, so we only need to check if the the size of the list stored the animals is already equal to the max capacity of the habitats given.

The fourth step is if there are incompatible animals in the given habitats already. I just cast it to habitat area and call getAnimals which all habitat areas have and then call isCompatibleWith for each animal.

Since all the possible failing cases has been checked, the left case is just successful addition





Intermediate:

- **How the zoo's areas and connections are modeled.**

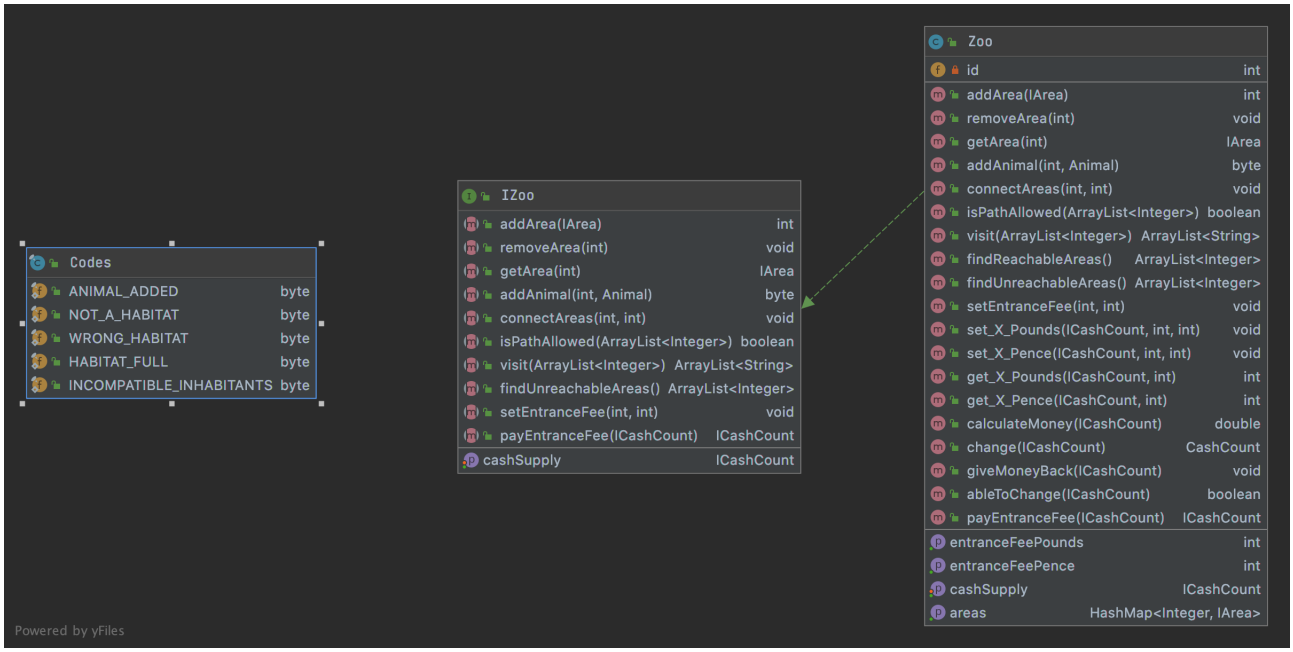
I modeled zoo with the hierarchy that all areas implement the interface IArea; then there are habitat areas and non-habitat areas, which are created to help reduce code duplication and make the structure clearer. Under the two classes, all the areas are either habitats or not, so they either inherit habitat area or non-habitat area. For the connections in-between, they are represented as Array Lists which are easy to add or remove objects(connections).

Zoo's areas are either habitat areas or non-habitat areas. Therefore, all the areas should either inherit Habitat or non-habitat. To implement the connect areas, I create a field for both habitats and non-habitats. The list is used to store the areas connected with some area. e.g if someone called connectArea(a, b), then b will be in the array list of a.

IsPathAllowed is implemented by checking if the adjacent areas' IDs list of the previous one contains the next one. For a list with n elements, if from 1st to (n-1)th element, they all pass the check, then the path should be allowed. This utilizes the method of array lists.

The method visit requires to output the nicknames of all animals have been seen. I first created a new array list which is used to store output data. Since we already have the lists storing animals in each area, what I next did is first check if the path is allowed, and if so, it checks if it is a habitat area. And if it is, then cast it to habitat area and add all animals' nicknames in it to the output list.

Then for the find unreachable areas, I tried to first find all reachable areas, and then for the reachable areas list, find the complement for it in all areas.



- **Alternative representation and drawbacks**

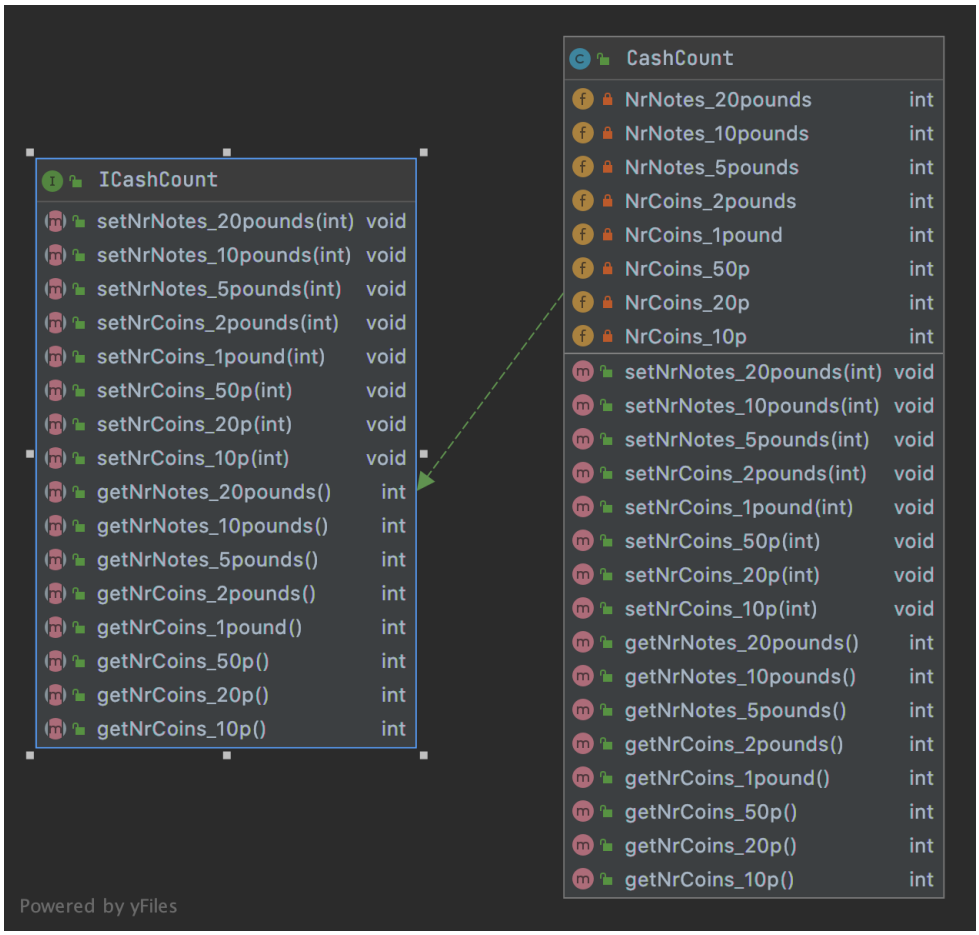
One of the alternatives that I can choose for the representation of areas is not to construct the hierarchy, mainly the two intermediate classes(habitat areas/ non-habitat areas). The cost of this way of implementing is that, I have to sacrifice the code quality like the Structure issues, e.g. code duplication and a well split implementation. One of the examples of code duplicity is the method addAnimal. For the fourth part of the method, I called method getAnimals just by casting the area to habitat area instead of the 3 subclasses, since the method is implemented in the super class, I don't need to define it in the subclasses any more, which significantly reduce the code duplicity.

For the representation of the connections of areas, I used Array Lists to stored the areas that an area can reach. One of the alternatives I can choose here is Linked Lists which can also store the objects like IArea. The reason why I chose Array Lists is the demands. At most of the time what I do to the list adjacent areas is to read the whole list, in which aspect array lists are weigh faster than Linked Lists, since Linked Lists have to go through all the lists to get the element we want, which could be time-consuming and not efficient.

Advanced

- **Representation of money and reasons**

I create a class CashCount to represent the money which implements the interface ICashCount. In this class, field variables are the numbers of each of 8 denominations we have. To know the real value of these money, I wrote a method calculateMoney, which would takes in a ICashCount object, since every class in the ICashCount would implements the methods, using getters and setters provided to calculate would be the best way and it would bothers other classes may added to the ICashCount later. I represent the money like this because it would be the best way that will be compatible with the possible classes to be added latter on and it would be the most general way.



- **Explain the key ideas behind your algorithm**

In this part, I used JUnit to help me construct the test. Just as the lab sessions we have done, I try to use exhaustive cases to measure the correctness of the implementations, which is try to build several cases that could be representative of one type of inputs. To show how the outcomes could be and when it is under certain conditions, the output of the implementations should be conformal.

Here the main method used is assertEquals. This method is used to compare the expected result and the actual result.

Firstly, since I created a concrete class CashCount, it should be able to compile and be constructed, so I try to call the constructor of it to confirm it works. For the setEntranceFee, they are field variables in the class Zoo, so after the setter has been called, the input value should conform to the getter method, which should give the value of the entrance fee.

Next, I wrote a test for setter and getter together. The idea here is that if the setter works perfectly fine, all the denomination should be same as coins are; also, if the getter works correctly, the getter method for the denomination could work and they generate the same integer value.

Finally, for the payEntranceFee, it is quite a general and big method; in other words, it does multiple functionality at the same time. Thus, it makes the test more complicated since multiple different cases are required for different scenarios.

- First case is the inserted money is not enough to pay the ticket and it will return the inserted money. Therefore, I set the fee to 4.50 and pay 2.00. I expect what I insert to be the output of the method, and the cash supply should not change, and thus it adheres the specification tightly.

- Second condition is enough money but no exact change, and it should return the inserted money as well. With the same cash supply and entrance fee, I paid 5 pounds and the change should be 50 pence, but there's no pence in the supply. Hence, I would expect the paid money to be returned and cash supply should not change either.

- Third situation is that exact money is paid and no change should be returned. Since the money is enough and paid, inserted money should add to the cash supply; therefore, the number 2-pound coins should increases by 2, 50-pence by 1. I again uses the getters to be the actual value. As no change should be returned, the output object should be 0 in each denomination.

- The final situation is too much money is inserted and the machine should give changes back. This time I insert 10 pounds, and therefore 5.50 pounds should be returned. Following the greedy method, it should be one 5-pound and one 50-pence that is returned according to the current cash supply. Since 10 pounds are paid, the cash supply should change accordingly.

As all four cases have been tested to work finely, the test is proven to be adhered to the specifications tightly enough.