Informatics 1: Object Oriented Programming

Assignment 3 –Modelling a Zoo

The University of Edinburgh
David Symons (dsymons@exseed.ed.ac.uk)

Overview

This assignment aims to give you some practical experience of working on a larger project with multiple classes. The basic section focuses on modelling a scenario using inheritance to create a hierarchy of classes. The intermediate tasks revolve around data representation and the advanced section deals with a search problem.

Marks will be awarded out of 100 and count for 30% of your overall grade for Inf1B. Completion time may vary depending on your experience and the grade you are aiming for. Of the 200 hours the course is designed to take, 20 have been allotted for this assignment. This is the estimated time for completing the basic and intermediate tasks. Marks above 69% are expected to be rare and require you to tackle the advanced section, which goes beyond the course in terms of content and time.

Before you start, please read the following sections below:

- *Good Scholarly Practice*
- *Marking Criteria*
- *Submission, late submission & extensions*

This assignment must be **submitted on CodeGrade by 16:00 on Friday April 8th**.

Basic: Building a zoo

The first task is to model a zoo based on the following description.

A zoo consists of multiple areas, each of which can be identified by a unique ID. There must be exactly one entrance area, which always has an ID of zero. Any number of picnic areas and animal habitats may be added to (or removed from) the zoo. Aquariums can accommodate seals, sharks and starfish, while lions, gazelles and zebra are kept in enclosures. Buzzards and parrots must be caged.

Each habitat is assigned a maximum capacity upon creation. This maximum number of animals may not be exceeded when populating the habitat. Attention must also be paid not to put incompatible animals into the same habitat. Zebra and gazelles can be in the same area, but neither can live with a lion. Sharks cannot be allowed near the seals, but starfish are compatible with both. Buzzards should be kept separate from parrots. Animals of the same type can always share a habitat, so there can be multiple lions in the same enclosure, multiple seals in the same aquarium, etc.

Your aim is to create and implement the required classes. This assignment is heavily focussed on design. The use of inheritance and a well chosen class hierarchy is absolutely essential! Just getting the program to pass functional tests will not give you many points!
The following steps guide you towards a clean design that significantly reduces code duplication.

Step 1: Download the provided code

Starter code is provided in the form of *src.zip*, which can be downloaded from LEARN (under Assessment → Assignment 3→ src.zip). Create a new Java project in your favourite IDE and replace the *src* folder with the one provided.

Step 2: Create the animal classes

Create a class for each animal *(Lion.java*, *Zebra.java*, etc.) inside the *animals* package. All of these must be related to the provided, abstract class *Animal.java*. As you can see from that class, every animal has a *getNickname()* method, which returns the name given to the animal upon creation. For example, you could create two parrots called Polly and one called Billy (nicknames are not unique). Think about where it is best to implement this method and where the name should be stored. Now, implement the *isCompatibleWith()* method for each animal according to the above restrictions on sharing a habitat. The method need not cater for animals that cannot live together anyway.

Step 3: Create the different areas

Create: *Entrance.java*, *PicnicArea.java*, *Aquarium.java*, *Cage.java* and *Enclosure.java* inside the *areas* package. These classes must all conform to the *IArea* interface, but you may find it useful to create some intermediate classes in the hierarchy. For now, *getAdjacentAreas()* can return null.

Step 4: Create the zoo

Create *Zoo.java* in the *zoo* package and have it implement the provided interface *IZoo*. This will require adding all unimplemented methods to your code. The ones that are not needed (yet) should return null, zero or false. This is to ensure your code compiles.

Step 5: Add new areas to the zoo

Implement the *addArea*, *removeArea* and *getArea* methods in *Zoo.java* according to the descriptions in the *IZoo* interface. You will need to decide how to store and retrieve the areas that are added. Depending on your design you may (or may not) need to change other classes. You must choose some way of generating integer IDs to uniquely identify each area that is added. How you do this is up to you. An area you have already added cannot be added again before first being removed, but you can add multiple areas of the same type by using different instances.
Finally, ensure the zoo always has exactly one entrance area with ID zero that cannot be removed. You do not need to worry about the possibility of multiple zoos being instantiated.

Step 6: Add animals

Now implement the *addAnimal* method. This method returns a byte code that indicates if the operation was successful or (if not) why it failed. Error codes are provided in *Codes.java*. When multiple reasons apply, the error with lowest value should be returned (as documented in *Codes.java*). The *addAnimal* method should take advantage of your other classes by delegating some of the responsibility. You may find that some changes to your current design and class hierarchy are required to do so.

Step 7: Report on your design decisions

Explain **where**, **how** and **why** you made use of inheritance. This is about explaining your **design**, not your code! Include this in the *Basic* section of your report. Guideline length is 300-400 words.

Note on reporting issues

Should you have difficulties completing any of the steps, you can explain your ideas in your report to get some of the credit. Please do so under separate sub-heading e.g. “Issues with step 5”. You can also report problems with partially working code, such as when it fails only for certain inputs. These sections are not required if your code is fully functional. Since the problems can vary greatly, a fixed word limit does not make sense here, but you should try to be as concise as possible.

Intermediate: All around the zoo

Due to an ongoing pandemic, the zoo has to adopt a one-way system for visitors moving between areas. Currently, all paths have been blocked off and it is your job to reconnect them and allow visitors to go between the different areas. The following methods need to be implemented.

connectAreas(int fromAreaId, int toAreaId)
After this method is called, visitors are allowed to go from the area with the *fromAreaId* to the area with the *toAreaId*. Going the other way is not possible (unless you call the method again with the arguments swapped). If *fromAreaId == toAreaId*, the call has no effect.

Example:
You have added one aquarium, which was given an ID of 1 and the entrance area has an ID of 0. It is not possible to go from the entrance to the aquarium until you call *connectAreas(0, 1)*. It is not automatically possible to go from the aquarium back to the entrance, but calling *connectAreas(1, 0)* allows this.

You now also need to implement the *getAdjacentAreas* method of *Iarea.java* (the other methods in the interface will be of use). This returns all areas connected to the current area by previous calls to *connectAreas*. Only include areas that visitors are allowed to go to directly (in one "hop") from the current area without breaking the rules of the one-way system.

The *getAdjacentAreas* method must be implemented before you proceed with any further tasks! It is quite possible that you will need to change your design to do this, but it is essential. You will not get marks for the following tasks unless you have done this!

isPathAllowed(ArrayList<Integer> areaIds)
Returns true if (and only if) visitors are allowed to visit the areas in the order given by the passed in list. The path starts with the area ID at index 0 in the list. The first area in the list need not be the entrance or even reachable form the entrance, but all areas must be part of the zoo.

visit(ArrayList<Integer> areaIdsVisited)
A visitor passes through the rooms in the order they are listed in *areaIdsVisited*. They write down the names of all animals they encounter in the order they see them. Within each area, the visitor always happens to spot the animals in the order they were added to the habitat. If the same habitat is visited multiple times, they write down the names again. The list of names is returned by the method. The visitor may start taking notes from any area in the zoo (need not start at the entrance). However, if they disobey the one-way system while taking notes, the notes are confiscated and null is returned.

findUnreachableAreas()
This method returns the set of all IDs belonging to areas that are part of the zoo, but cannot be visited when starting from the entrance area and obeying the one-way system.

Report
The intermediate section of your report should cover the following:
1. Explain how you modelled the zoo's areas and connections. (Guideline 300 words)
2. Describe an alternative (data-)representation and explain why you did not choose it. (Guideline 200 words)

As in the basic section, you can (optionally) report issues and ideas in the report for partial marks.

Advanced: Ticket machine

The zoo offers ticket machines at which visitors can pay the entrance fee in cash. The machines need to be programmed to return the correct change. Your task is to implement the *payEntranceFee* method of *IZoo.java* along with a few other (mostly trivial) methods described here:

setEntranceFee(int pounds, int pence)

As the name suggests, this is used to set the price of a ticket in pounds and pence. Note that you should not use floats or doubles when dealing with currency. This is due to precision issues with those types. After some numeric operations, a result that should e.g. be exactly £1 can be £0.999. Occasionally, the zoo has an open day where tickets are free (visitors still need a ticket though).

setCashSupply(ICashCount coins)

This method is used to stock the ticket machine with cash that it can return as change. This will require implementing the *ICashCount* interface, which is used to represent an amount of cash in terms of the numbers of the following notes and coins: £20, £10, £5, £2, £1, 50p, 20p and 10p. The machine does not accept smaller coins or notes larger than £20. As its name indicates, this method sets the cash supply (rather than adding to it).

getCashSupply()

Simply returns the *ICashCount* instance, which should reflect the number of notes and coins currently in the ticket machine. These counts may increase as a result of cash being inserted or decrease when change is returned.

payEntranceFee(ICashCount cashInserted)

The challenging part is this method, which takes as its parameter the cash that is inserted and returns an *ICashCount* instance representing the correct change. The contract of the method is as follows:

- If the inserted cash amounts to less than the price of a ticket (as set via *setEntranceFee*), the machine returns the cash that was inserted. It must return the exact same number of each note and coin (not just any equivalent amount of cash). This implies that you can (but need not) return the instance of *ICashCount* that was passed in.
- If too much money is inserted and the machine cannot return the exact change (due to running out of the required notes or coins), it returns the inserted cash. As above, you must return the same number of each note and coin. The inserted cash is available to the machine for making up the change.
- If the inserted cash pays for the ticket exactly, no change is given. This is indicated by returning an *ICashCount* instance that contains zero of each denomination. The inserted cash is added to the machine's cash supply.
- If too much money is inserted and the machine is able to give exact change, it must do so. So as not to annoy the visitor by paying out the entire change in 10p coins, the machine will always prioritise large denominations to return the least number of notes and coins possible (given its current supply). The cash inserted by the user can be used to make up the change.

Example: A user pays for a £17.80 ticket with a £20 note. Coins adding up to £2.20 must be returned as change if at all possible. If the machine has a £2 coin and a 20 pence piece, then those must be given. If, however, the machine is out of 20p coins it returns two 10p coins instead. If it does not have those either, it returns the £20 note (no sale).

Note: In some cases it is easier to find the correct change than in others. Marks are awarded based on the types of cases your algorithm can handle. I would recommend starting with a simple approach and then improving it if you have time.

Testing: Adhering exactly to the specification is part of the challenge. The best way to make sure your algorithm is working is to write a few tests (the CodeGrade tests for this task are hidden). This is optional though. There are no marks for writing tests.

Report

The advanced section of your report should cover the following:

1. Explain how you chose to represent money in your implementation of *ICashCount* and why. (Guideline 100-150 words)
2. Explain the key ideas behind your algorithm. Give an overview of how it works, rather than explaining it line by line. (Guideline 300-500 words)

Again, you can report issues and ideas in the report for partial credit.

Restrictions

The following restrictions apply to this assignment:

- Do not use any functional language constructs such as lambdas or streams.
  - Most functional constructs are methods that take a function (or predicate) as an argument, like *map*, *filter*, *reduce* or *removeIf*.
  - The arrow operator (→) is another indicator.
- Do not use any third party libraries.
  - All imports starting with *java.util* are allowed.
  - No other libraries allowed.

Good Scholarly Practice

Please remember the good scholarly practice requirements of the University regarding work for credit. See: https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct
This also has links to the relevant University pages.

Marking Criteria

Marks will be assigned in accordance with the University's Common Marking Scheme (CMS).
See: https://web.inf.ed.ac.uk/infweb/student-services/ito/students/common-marking-scheme

For a pass grade (up to 49%) it is sufficient to tackle the *Basic* section. This includes both code and report aspects, with the latter playing a significant role. Code quality is also taken into account, but to a lesser degree than in higher grade bands. The same holds true for the presentation of the report.

For a 2nd or low 1st class (up to 79%) you must first complete the the basic report and almost all basic coding features to a high standard. Only then does it make sense to address the *Intermediate* tasks (code and report). Quality and presentation become more important. There should be evidence of a conscious effort to produce good code and a clear, informative and structured report.

The advanced section is for those aiming at an A1 or A2 (up to 100%). At this point, all basic tasks must be well polished and almost all intermediate tasks should well solved and documented. Grades in this range require excellent code quality and a clear, well presented report that demonstrates real understanding and insight.

The report is an important part of this assignment. It can give (or lose) you a lot of marks. Even if you only have some ideas for how to approach a task, they are worth writing down.

Report

The assessment of the report includes the following aspects, which will be further discussed at the Wednesday session introducing this assignment.

- Content
  - Answer questions directly and to the point.
  - Do not write down everything you know about the topic, stick to what was asked.
  - Go beyond giving the correct answer by explaining why it is correct (shows understanding).
- Structure
  - Break the content down into well labelled sections (see report template).
  - If you are making an argument, ensure you organise your points into a logical sequence.
  - Try to make the text flow by avoiding frequent or abrupt changes of topic.
- Writing style
  - Should be clear and concise.
  - Make the report as short as possible without sacrificing content.
  - Avoid repetition: Make your point once and make it well, but avoid re-stating it.

Guideline lengths

Markers must read the stated number of words for each section, but can stop reading after that and only give you points for what you managed to express in the given words count. There is no explicit penalty for writing more, but it is obviously in your interest to write concisely. If you are well under the guideline length, you are likely missing some content.

Submission

This assignment is **due at 16:00 on Friday April 8th**. Submission and resubmission rules are as in the previous assignments i.e. multiple submissions are possible until your deadline, after which the first submission is marked.
Submission is via CodeGrade. Go to LEARN→ Assessment→Assignment 3 → CodeGrade.
When the interface opens, you can click on “New tab” (on the bottom left) to enlarge the view.

Please practice submitting well before the deadline. New submissions will overwrite previous ones (until the deadline) and you can submit as often as you like. You will receive automated feedback on your progress and can adapt your solution to achieve a higher score.

Keep in mind that the auto grader only tests for functional correctness and robustness. While these scores contribute to your grade, your marker will also consider your report and all other aspects listed in the *Marking Criteria* section.

For automated testing to work, your code must compile. This is why it is essential that you provide dummy implementations for all methods, even if you do not intend to solve the corresponding tasks. All you have to do is return null, zero or false in the method body.

CodeGrade also insists that you upload all required files, including the report. If you want to test your code before you have written the report, just upload a blank file named Report.pdf (don’t forget to replace this with your real report later).

To upload, create a .zip file containing your src folder and your report at the top level – like this: