Tutorial 2 Solution Guidelines: Inf2C-SE 2014/2015

Thursday 9th October, 2014

Below I have given some notes to the questions posed in the second tutorial. Similarly to the first tutorial there are no absolute right or wrong answers. Although there is less subjectivity than in the first tutorial.

One thing to avoid is to mistake a class diagram for a blue-print for their object-oriented code. It is important for such students to realise that they are not attempting to design the implementation classes, but describe the *design* of the implementation. In particular if all we wanted was a list of the classes in the implementation, then why not simply write down that description in the implementation language. We could even write software to extract a UML class diagram from source code.

However, implementation will have more classes and more detailed interfaces than will a class diagram. In short the class diagram is an intermediate point between the unavoidably ambiguous natural language of the use-cases and requirements specifications and the unambiguous implementation source code. Here you are making the specification of the solution *more* concrete, without as much commitment as a full implementation. The goal is to clean-up mistakes in the requirements before implementation is begun. Because of this, the class diagram should still be understandable to someone who is not a software developer.

1 Thermostat

This is a relatively simple system. Most of the classes will only have a single instance, with the exception of a class to model a room. I would expect a good solution to have the following classes:

- Room, with a temperature attribute. The student may also model this as a sensor, or even a room with a sensor attribute and a separate sensor class. Personally I find this a bit too detailed at this stage. A good compromise would be a room with a 'sensor-temperature' attribute.
- Boiler, this will need associated methods for turning on and turning off.
- Thermostat, this has the associated method to either turn on (the entire heating system) and another method to set the desired temperature.

In addition we would expect there to be some relations. In particular the rooms are connected to the boiler and thermostat in some way. What these relations are named is not important. But for example the Boiler may 'heat' each room and the thermostat may 'poll' each room. Finally a boiler is controlled by a thermostat.

Concerning multiplicity, there should be exactly one thermostat and one boiler but each can be connected to one or more rooms. Exactly one thermostat controls exactly one boiler.

A good point here is that the class diagram may concern itself with allowing the user to set the temperature in whichever room they are in. This is still setting one global temperature for the property rather than a different temperature in each room. In this case, you still have only one thermostat which still controls only one boiler. But the desired temperature setting on the thermostat can be controlled by multiple *remote controls*.

For example solutions to this exercise, please see the lecture notes: `http://www.inf.ed.ac.uk/teaching/courses/inf2c-se/Lectures/Lectures-2014/lecture05-classDiags.pdf`

2 Elevator

Because of the cut down nature of this problem there is only one user. Hence the solution should be careful not to have multiple users. Of course a real solution to a real elevator problem would have multiple users, but that is not what is being asked here.

An important part of specification is to detail what the system will *not* do. So explicitly having only a single user (at a time), details well that the system cannot cope with multiple users. At that point, the customer may well require a change.

This system, then requires three classes:

- The elevator itself, with a current floor and destination floor which may be null. You may also include an 'occupied' attribute.
- A user, with a current floor and a desired floor.
- The call buttons on each floor.

Again there can be variations on this. The student may decide to model each floor with each floor having a call button. A really good solution will include a generic button class which is sub-classed for the call buttons and the elevator buttons. The advantage of this is that each button has to have the functionality of displaying to the user that it has been pressed and that the system is currently processing that request (and of course additionally some way for the system to remove the highlighting from the button once it believes that request to have been satisfied).

Figure 1 depicts a possible class diagram for the simple elevator system.

3 Shopping List App

There is quite a bit more scope for creativity here. Any solution will need some class for at least:

- A recipe
- An item to be included in a shopping trip
- A shopping list, consisting of items which need to have associated check buttons, such that the user can cross items off the list as they are added to the basket.

I think a good solution will extend the items in a recipe for use as items on a shopping list. The extension includes the functionality to check the item off on the list.

Figure 2 depicts a possible solution to the shopping list application. Note that the relationship between the shopping list and a recipe is only one of referring to. Adding a recipe to a shopping list merely results in all of the products in a recipe being added as items to the current

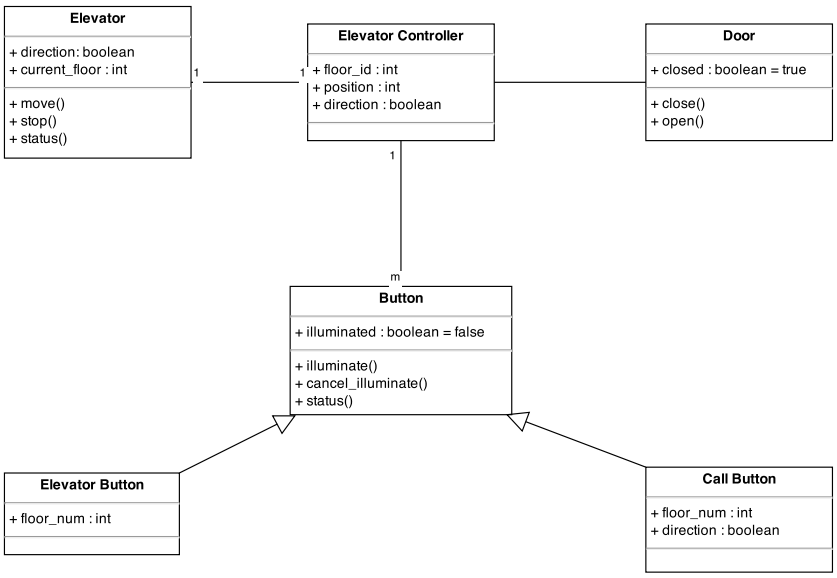

Figure 1: A possible solution for the elevator problem. Note that here the elevator to elevator controller is depicted as a 1-to-1 mapping as is specified by the problem. In a more realistic setting with multiple elevators this could be a many-to-one relation, with one elevator controller, controlling many elevators.

shopping list. Removing a recipe can be done in a similar fashion by first looking at what is in the recipe and then removing all of those items from the current shopping list. Exceptions would be made for the cases in which the user has already removed a recipe-item manually.

Also note that the solution given would mean that a user could simply reduce the quantity of an item if they are able to buy some of them but not all. For example if the recipe calls for 12 eggs and for some reason the first shop only has 6 eggs then the user can reduce the quantity required rather than checking the item.

4 Sequence Diagram

This section is mostly intended to expose students to drawing a sequence diagram. Most sequence diagrams are relatively straightforward. You are drawing the sequence of actions for a *single* use-case, usually without any alternative paths. Figure 3 depicts a sequence diagram for the elevator system. The particular use-case is that of the user calling the elevator to the user's current floor. Here we have assumed that the doors close method can be called even if the doors are closed.

5 Advanced - Negative Commenting

This depends hugely upon what class diagrams have been produced for the first parts. It might be that your class diagrams are already quite concise, and hence removing each of the classes does indeed have a drastic effect on the system.

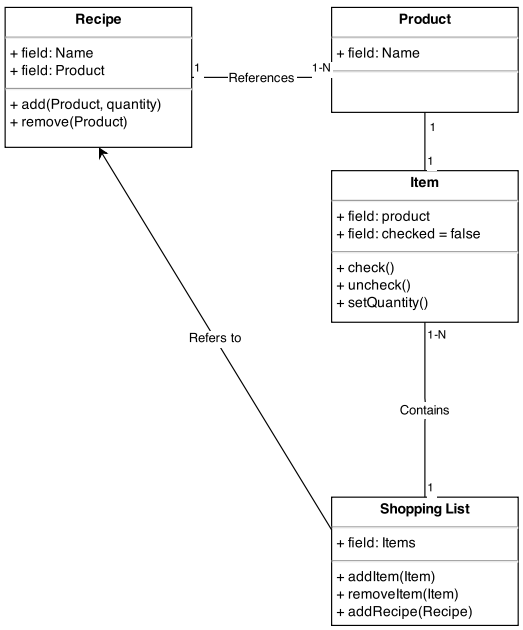

Figure 2: A possible class diagram for the shopping list application. This solution is kept simple.

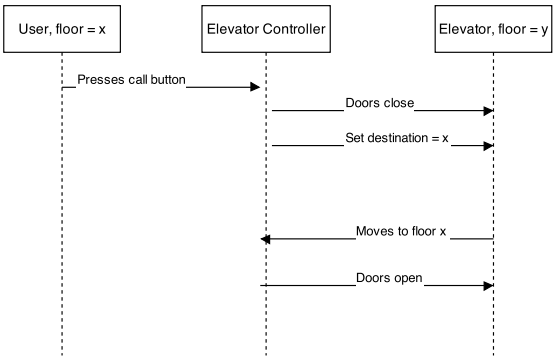

Figure 3: A possible solution to drawing the sequence diagram for a single use-case for the elevator system. In this case the use-case is that of calling the elevator to a floor the user is on.

In general *negative commenting* on your class diagrams is a useful exercise particularly when first learning to use class diagrams. Hopefully, as you use class diagrams more often it becomes a question you ask of yourself before you *add* a class to the diagram. Hence going over the document at the end is less necessary. But it could rarely hurt.