Harder class and sequence diagram exercises

September 18, 2014

Purpose

These exercises are intended to be a little more challenging than the basic ones. Do these if you have time once you are confident with the basic exercises. If you don't do them now, I suggest doing them later for revision. There may well not be time to discuss these in the tutorial, but if you would like feedback on your work, write it up and give it to me for marking (either by email or on paper when you see me). You may do this at any time during the course. I expect to be able to return marked work within a few days under normal circumstances – but of course that won't work if everyone hands in a pile of work a few days before the exam! If that does happen I'll consider arranging a group feedback session instead.

Exercises

1. Imagine you are leading the development of an application to provide university students with individualised timetables. Assume that the details of the situation are those you are familiar with: students registered for 120 points of courses in a year, some courses having lectures, some having tutorials, lectures and tutorials taking place in rooms at certain times on certain days, etc. Develop a conceptual class model for this situation. (That is, determine the domain classes that you will need and their conceptual relationships – do not (yet!) concern yourself with the timetabling functionality.) You should be able to use (at least) the following UML elements: class, attribute, association, generalization, aggregation.

2. Many interactions involve loops. Modify the `Party` example so that instead of a party involving a single entertainer it involves a collection of entertainers, whose costs have to be summed. First, think about how you could represent this in a sequence diagram and sketch possibilities. Then, look up sequence fragments in UML sequence diagrams and see how UML2 does represent this situation.

3. Consider a situation involving a callback. Show on a sequence diagram:

   (a) an object `a:A` sends a message `registerObserver(a)` to `b:B`, i.e. with a reference to itself as argument (assume `b:B` replies without sending any messages itself);

   (b) an actor sending `newValue(17)` to `b`;

   (c) `b`, as part of its response, sending message `notifyObservers()` to itself;

   (d) `b`, as part of its response to its own message `notifyObservers()`, sending message `notify()` to `a`;

   (e) `a`, as part of its response to `notify()`, sending message `getNewValue()` to `b`, which replies with 17.

   Put in nested activations and all return arrows, and check that what you've written makes sense (everything, except the initial message and the actor's message, should be caused by something). This is what happens in the Observer pattern, which we'll discuss later. Can you see what it's for, designwise?