Tutorial 3 Solution Guidelines: Inf2C-SE 2014/2015

Friday 17th October, 2014

1 Activity Diagrams

1. When and why would you use an activity diagram rather than a sequence diagram?

   Answer: When you are describing flow from one system to another. Another important distinction is that a sequence diagram typically describes the flow of exactly one use-case. Whereas an activity diagram may describe the flow of several use-cases, particularly those that interact with one another. In addition a sequence diagram typically *requires* a corresponding use-diagram but an activity diagram may be used independently.

2. What is the main way in which an activity diagram is differentiated from a flow chart?

   Answer: The main difference is that activity diagrams can show parallel and concurrent activities. Many display some skepticism towards their use describing them as glorified flow charts. They are relatively easy to understand however. A glorified flow chart should at least be useful in any situation that a flow chart is.

2 State Machines

1. States in a state machine diagram may refer to the states of a single object or a collection of objects. Unfortunately, when we have a collection of objects the number of *possible* states can increase dramatically. If you have n components each with m states, then the total number of *possible states* is m^n. More generally each component may have a different number of states, so if each component i has m_i states, and there are n components, then there are $\prod_{i=1}^{i=n} m_i$ total possible states. If we have just 8 components each with just 2 independent states we have 256 possible states. Why is this not normally a major problem for modelling?

   Answer: There are a few reasons. The first is that for a state machine you are only modelling the *significant* states. Hence, it is hopefully the case that for most components $m_i = 1$. Even, where $m_i > 1$, hopefully the states of each modelled component are not independent from each other. So for example, either all components are in some 'recovery' state or none are. Finally, if the states of components really are independent from each other, then their behaviour *should* be too. In this case, you can model them separately from each other (even if on the same diagram), such that you need only depict in the independent states of each component, rather than any combined state.

2. Suppose then that you are taking part in some project and a co-developer comes to you with a state machine diagram with a number of states far higher than you were expecting for the size of system you are developing. What might this indicate?

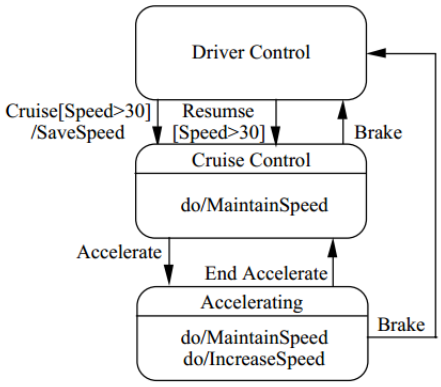

Figure 1: A state machine diagram from the cruise-control system.

Answer: The question is a little vague so I suspect there will be a variety of answers here. But the main concept here is that either there is something wrong with the model or something wrong with the design. So first check that the model is reasonable. A likely place it might not be is that the co-developer has modelled too many insignificant states. For example in the *thermostat* system, we do not require a state for each different temperature of each room. We do not even need a state for each room being above or below the desired temperature. We simply require two states, one in which at least one room is below the desired temperature and one in which there is no such room.

However, if you decide that the modeller has indeed done a good job, then it might be that the system you are designing has a poor design. The number of significant states should be manageable by a human reader, in particular by the user/customer. If you have a large number of states each with *qualitatively* different behaviour then you will have a difficult time verifying the correctness of each state. Even if you implement this correctly you may have a system which is difficult for the user.

Finally of course, the scenario includes a large amount of scope for interpretation. It includes the phrase "far higher than you were expecting" but this is of course a function of your expectations as well as of the model. It may well be that you are developing a complex system, in which case, perhaps you require many states with significantly/qualitatively different behaviour.

3. Draw a state machine for the Cruise Control System as described in the first and second courseworks.

   Answer: See the answer if Figure 1.

3 Design Patterns

1. Design patterns suggest that you copy an existing solution and modify it correspondingly for your specific application. One thing novice programmers are taught is that *"Copy-and-paste"* programming is to be avoided. Worst still, you may apply the same design pattern more than once in the same project. Isn't this a good argument to avoid the use of design patterns?

   Answer: Yes we try to avoid duplication, and yes the application is of a design pattern is necessarily the duplication of some logic. It may not be a direct copy-and-paste, the

duplication may be of the more abstract logic than of the actual concrete syntax used, but it is duplication nevertheless. However, the design pattern has arisen because, at least the authors, see no obvious way to write down the solution in a resuable/parameterised way. That is why the problem is a problem in the first place.

2. If you agree that the main advantage to the use of design patterns is as an aid to communication with other developers. Is there any value in using design patterns for an individual project? If you believe that you will only ever write programs that no one else will read (which is a sad thought), is there any benefit to *learning* design patterns?

   Answer: One obvious benefit is that you are able to draw on the thoughts of other expert software developers regarding the general advantages and disadvantages to the use of each particular design pattern use and apply those to your current situation.

3. Some programming languages have existing features that subsumes the need for particular design patterns. For example many dynamically typed languages (and some statically typed languages/variants) have a provision for *'MetaClasses'* and this largely obviates the need for the Factory pattern. Do you think that this means that the concept of a design pattern is not a useful one?

   Answer: Well, it may well be that a particular design pattern is subsumed by some language feature in another language. However, if the language you are currently using does not support that feature, then you need a plan for what you are doing *now*.

   The use of a particular design pattern may well be the best approach. You might consider switching implementation languages if that is an option, or you might consider proactively contributing some effort towards the introduction of that particular feature for the programming language that you are using. Though that will likely not affect your current course of action. It may be that such a feature is simply infeasible to be included into the particular language you have chosen.

   Finally, even if some particular design patterns are indeed subsumed by existing language features, this does not invalidate the *concept* of a design pattern. The concept can be thought of as capturing a recurring solution *for which* there is no current language feature. Whatever your opinion of the existing one, you would need a very strong opinion of the expressivity of abstraction of some programming to claim that it could never benefit from such a concept.