The Limits to Factoring

David Ungar
Sun Microsystems Laboratories

Yes, we DO need different modularizations at different times. However, we may never see support for this capability in development environments in widespread use because it brings up several tough challenges:

1. Factoring is not enough.
2. We need ambigous modularization.
3. Classes are counter-productive.

1. Factoring is not enough

Following the tradition first set forth so clearly by David Parnas, we assume that the most important things to modularize are those design decisions that may be subject to change in the future.
However, as any one imperative statement in a program may be influenced by multiple design decisions, it is impossible to factor the program so that each piece contains code that is impacted by a single design decision.
For example, the statement: ``contents[i++] = x'' in the body of a Stack.push routine embodies at least three design decisions: that the stack is represented by a vector, that the index grows with the number of items pushed, and that (taken together with the pop code) the Stack is LIFO rather than FIFO.

Although one could try to create a system with multiple factorings, the result would be that a single piece of code would have to reside in more than one bucket, and the result would be a confusion of causality leakage. When making a change, one could easily end up making other inadvertent changes.

Perhaps it would be better to adopt a factoring + labelling scheme, in which code would be factored into unique buckets, but could also be labelled in such a way as to impose a hypertext-like structure atop it. For an analogy from file systems, consider the MacOS file system, which keeps each file in exactly one folder, but provides labels as a way to orthogonally group files. With this approach, one could easily find all the code that was related to a particular design decision, and one would be forced to be aware of the structure of the system as well.

The idea that factoring will not solve this problem may be difficult for practitioners.

2. We need ambiguous modularization.

The most powerful forms of communication present the recipient with ambiguity for him or her to sort out.
Humor, theatre, visual art, music all offer both cognitive and affective impact in many dimensions. The listener gets to jump from aspect to aspect

WITHOUT making any gestures at all--merely by attending to different aspects. This fluidity allows the focus of attention to be steered by precognitive as well as cognitive processes. As programmers, we do this all the time with our programs, too. Variable names, line-formatting, program structure, flow-of-control layout decisions are made carefully and intuitively to communicate as much as possible in many dimensions WITHOUT requiring the reader to perform a gesture. Yet all the mechanisms for multiple factorings, or grouping via labels would seem to lack this fluidity. How can get as much fluidity for large-scale program structure as the Golden Gate bridge has for its architectural structure?

3. Classes are counter-productive

Classes force code to be factored according to the instance variables referenced and the static types of the receivers. They encourage programmers to think about a single factoring and a single hierarchy, prima facie antagonists to the kind of multidimensional structures we are attempting to foster in this workshop. I think that prototypes are a step in the right direction, but they have their own problems, too. What's a language-designer to do?

Each of these challenges is tough enough by itself, taken together we will have our work cut out for us.