These are the slides of a presentation I gave at the Workshop on Linking Aspect Technology and Evolution in 2005. The notes of the presentation are included. For more detailed information please read the paper “Experiences with Identifying Aspects in Smalltalk using ‘Unique Methods’” which can be downloaded from my website at http://prog.vub.ac.be/~kgybels/.

© 2005, Kris Gybels and Andy Kellens. All rights reserved.
The code base we investigated is that of a pretty much default VisualWorks system image as it comes with the distribution of VisualWorks Smalltalk. Some of the code packages already loaded into the system image by default are those for the collection class framework, the graphics framework, the framework for interfacing with databases etc. Together with a few additional packages we loaded for the experiment, the system contained all-in-all about 3400 classes implementing a total of about 66000 methods. All of this code has been implemented without use of Aspect-Oriented Programming technology, so our questions of course then were: are there any hidden cross-cutting concerns in there and how do we find these?
For finding cross-cutting concerns, we think it’s important to take a look at how programmers without AOP at their disposal would have implemented concerns which we now know to be typically crosscutting. Of course we know that their resulting implementation is often characterized by code duplication and scattered code, but it is interesting to look at the different degrees with which these characteristics occur and the impact this may have on the detection techniques we should use. This seems especially so for the degree of code duplication.
Here's one example of a manually woven cross-cutting concern: the code in red here is an implementation of synchronisation. It consists of a pattern of a few lines of code that is repeated across the methods, you could imagine this was manually woven perhaps by simply copy/pasting the code. It’s no surprise then that the use of existing techniques for detecting “copy/pasted” code duplication have already been explored for aspect mining.

Another example, and a quite popular one in AOP research, is that of implementations of the observer design pattern’s notification concern. Here the way it is done in Smalltalk is shown: the code in red calls the method “changed:” to send out notifications. Each of these invocations of “changed:” is just one line of code however, and not a more complex pattern that occurs for example at the start and end of methods and that looks like copy/pasted code. So something we’d like to put up for discussion here is whether techniques based on detecting copy/pasted code are suitable for detecting such cross-cutting concerns or whether other techniques are needed.
An observation we hypothesized could be helpful for detecting this second kind of cross-cutting concerns is that the methods like “changed:” or “logged:” are typically centrally defined methods for messages with only one implementation. These are kind of like the reverse of an advice, and of course it makes sense to apply a “Turn Method into Advice” refactoring on them. Andy and I are also particularly interested in this refactoring, and let me advertise for a moment that we have another workshop paper here at AOSD that deals with some issues that arise in performing and automating it. As another sidenote and something to put up for discussion here at the workshop is the question of whether cross-cutting concerns characterized by high levels of code duplication should not be more suitably refactored to high-level aspect languages like COOL for the synchronisation concern rather than low-level ones.
Now back to our presentation topic: so based on that observation we went looking for unique methods in our Smalltalk code base. With unique method we mean these centrally defined single-implementation methods, with the added restriction that they should also not have a return value: it doesn’t seem to make much sense to try and apply a “turn method into advice” refactoring on a method that does return a value. So we filtered out all the unique methods in the code base which turned up 6248 unique methods. This is still quite a large number and a lot of these methods were likely not implementations of cross-cutting concerns but rather auxiliary side-effect methods or accessor methods. So to further filter the list down, we took a very rough approximation of scattering into account by approximating it with the number of invocations to the method. Limiting the list to those unique methods with more than 5 invocations resulted in only 228 methods, and with more than 10 in 124.
The next step after this filtering in our approach was simply to go through the list of filtered unique methods and look for those that could be potential aspects. The ones we found turned out to be mostly examples of state change notification concerns, like “updateAfterDo:” and “broadcastPendingSelectionChange”. A few others are different like “markAsDirty” which is an implementation of cache invalidation. What is perhaps more interesting though than just this list of methods is a question we have on whether this identification process can be further automated. We identified these methods now simply mostly by having our attention caught by these keywords such as “mark”, “update” and “broadcast” in the names of the methods. We did of course also further investigate the method bodies of these “eye catchers” to confirm that they are really potential aspects.
So the approach we took worked quite well for detecting some aspects, but one thing that did surprise us is that the “changed:” method in Smalltalk for sending out notifications was not detected as a potential aspect, the reason is that it turns out that this method is not unique. Apparently, two classes override the implementation of “changed:” from the root class Object as a kind of hack for making objects observe themselves. There might be other methods like these which are not entirely unique, for example implementations of logging with different variations. We currently also don’t have a comparison available with aspect detection in a Smalltalk code base using techniques based on detecting code duplication.
• Experience report with detecting aspects in Smalltalk
  – Mostly update notification aspects found
• Interesting questions for discussion:
  – Are “Unique Methods” a good technique to identify aspects which are implemented as calls to a central entity?
  – Are other techniques (code duplication) suitable as well? Or are the two complementary?
  – Using techniques for ‘interpreting’ keywords in names? (”mark”, “update”, …)