Experiences with Identifying Aspects in Smalltalk Using 'Unique Methods'

Kris Gybels* and Andy Kellens†
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
Belgium
{kris.gybels, andy.kellens}@vub.ac.be

January 9, 2005

Abstract

Now that Aspect-Oriented Software Development has matured, the techniques developed in this field may be used to cleanly modularize the crosscutting concerns in legacy applications. Due to the complexity and size of these applications it is important that the identification of crosscutting concerns and the transformation into aspects is automated as much as possible. In this paper we present a simple heuristic named *Unique Methods* which can be used to detect crosscutting concerns in an application. We demonstrate the use of this technique on the code of an entire VisualWorks Smalltalk image and discuss the benefits and disadvantages of our approach.

1 Introduction

With Aspect-Oriented Software Development (AOSD) becoming more and more mature, there is a growing interest in application of its techniques to the field of re-engineering legacy applications. Even if such an application was modularized as cleanly as possible using only pre-AOP techniques, a number of concerns may remain scattered throughout the application. Using AOSD techniques, these crosscutting concerns can be factored out into aspects, thus improving the further maintainability of the application. This transformation of pre-AOP code into AOP code involves two steps: identifying crosscutting concerns in the source code (aspect mining) and transforming these concerns into aspects while preserving the application's behaviour (aspect refactoring). Because of the typical size and complexity of legacy applications, it is imperative for the success of aspect-oriented re-engineering that automated support is provided for performing the transformation. Our focus for this paper is a particular heuristic which can be used in supporting the aspect mining step.

In order to find heuristics for aspect mining, we need to look at how a pre-AOP developer would deal with concerns we now consider to be typical aspects. For some concerns such as synchronization of threads, it has already been observed that they are often dealt with using just copy/paste reuse, leading to scattered code duplication. Some existing techniques for supporting aspect mining have hence been based on heuristics for finding such scattered code duplication using pattern matching techniques [6].

For other concerns, such as logging and state-change updating, we can observe that the situation is usually less severe: crosscutting concerns can also have been implemented as a single method that is called from a wide-spread number of places in the code. These implementations are thus not

*Research assistant of the Fund for Scientific Research - Flanders, Belgium (F.W.O.)

†Author funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)"

```
public class BoundedBuffer {
    /* definition of semaphore variables and
    constructor to initialise semaphores */

     public void put(Object o) {
       putSem.P();
       putExclusion.P();
       array[putPtr] = o;
       putPtr = (putPtr + 1) % array.length;
       putExclusion.V();
       takeSem.V();
     }

     /* definition of method get, similar to
     to put */
   }
```

Figure 1: Example of tangled code for implementation of synchronisation concern in Java without using built-in support for the concern (Example adapted from PhD dissertation of Cristina Lopes [9]).

characterized by multi-statement snippets of code spread around in the implementation, and pattern matching techniques may be less suited to identifying these as aspects.

This paper is structured as follows: in section 2 we present the observation which lead to the heuristic of *unique methods*. In section 3 we define the *unique methods* heuristic. We applied our heuristic to an entire Smalltalk image. The results of this experiment are described in section 4 and extensibly discussed in section 5. Before concluding the paper in section 7, we present some related work (section 6).

2 "Manual" Aspect Weaving

In pre-AOP languages, developers would have found it difficult to avoid scattered and tangled implementations of certain concerns. Nevertheless, developers could have dealt with such concerns in a number of different ways, essentially by "manually" weaving them. One kind of concern for which this has been extensively studied is the concern of synchronisation of concurrent processes [9]. Figure 1 gives a typical example of an implementation based on explicit semaphores. What characterizes this implementation is that each of the methods of the BoundedBuffer class starts and ends with a similar pattern of manipulating the same semaphores, which looks very much like it has been simply copy/pasted.

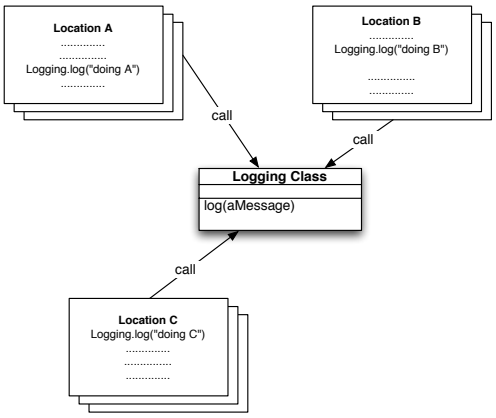

Figure 2: Logging as a central class providing logging functionality

```
moveTo: newx and: newy
  x := newx.
  y := newy.
  self changed: #x.
  self changed: #y.

shiftSidewaysTo: newx
  x := newx.
  self changed: #x.
```

Figure 3: Example of tangled code for implementation of update concern in Smalltalk.

In other cases developers need not have been desperate enough to revert to copy/pasting. A typical example in this case is a logging concern which can be encapsulated well enough using classical techniques to at least avoid tangling, though not scattering. A typical implementation for logging is illustrated by figure 2: a singleton class is defined which encapsulates the management of the logging file, the logging of only those messages which are of a certain 'verbosity level' etc. The 'log' method is then called from a number of places in the code, where it is just a single-line statement thus not being really tangled, nor a

good example of copy/paste code, though still scattered. Another example of this kind of concern is the implementation of the update notification concern in Smalltalk for the observer protocol: a single method 'changed:' is defined on the root class Object, which is called whenever update notifications are to be sent as illustrated in figure 3. Sends of the 'changed:' message are scattered throughout the entire Smalltalk code base, but each message itself is only a single statement and not part of a larger pattern as with the synchronisation concern implementation.

3 Identifying Aspects

In the previous section we have observed that certain typical implementations of 'manually woven' concerns are characterized by both high degrees of tangling and code duplication and high degrees of scattering. It is hence no surprise that a number of aspect identification techniques are based on exploiting existing techniques for code duplication .

But one question we have for discussion at the workshop is: are these techniques also suitable for detecting 'manually woven' concerns which are more (and only) characterized by a high degree of scattering? As these consist of single lines of code, techniques for detecting copy/pasted duplication of larger blocks of code may not be suitable.

Another observation that can be made is that the concerns of logging and updating were implemented using a single central entity, particularly a *unique method*:

Unique method:
"A method without a return value which implements a message implemented by no other method."

Thus another point we would like to put up for discussion at the workshop is that we hypothesize that 'unique methods' can be used as a simple heuristic to identify candidate 'manually woven' concerns of the logging and update type. We have done some experiments in Smalltalk in order to confirm this hypothesis, though with mixed results as reported in the next section.

The 'unique methods' heuristic also ties in with other research we are performing on automated support for defining crosscuts for the refactoring step of aspect mining [5]. The goal is to have a refactoring tool which allows one to select a method and turn all calls to it into an advice, while also producing a crosscut for that advice which is not simply an enumeration of all the places in the code where all the original calls occurred. Limiting ourselves in that research to refactoring unique methods avoids problems with polymorphism, and allows us to concentrate on the crosscut generation step. We conjecture that existing refactoring techniques can be used in combination with the unique-method-to-advice refactoring to turn non-unique methods and highly-tangled concern implementations into unique methods first.

Note that in the definition of unique methods we exclude methods with a return value[1] since for methods with a return value it doesn't make sense to apply a method-to-advice refactoring, they are clearly part of the base functionality of the code.

Class	Selector
CodeComponent	#startLoad
Locale	#currentPolicy:
Menu	#addItem:value:
ScheduledWindow	#updateEvent:
UIFinderVW2	#showClasses:
ComposedText	#centered
UIBuilder	#wrapWith:
Text	#emphasizeAllWith:
Cursor	#show
Image	#pixelsDo:

Table 1: A few examples of unique methods

4 Unique Methods in Practice

We took the following approach in performing an 'aspect identification' experiment based on 'unique methods' on a Smalltalk image: we selected all unique methods in a standard VisualWorks Smalltalk image containing 3400 classes which together implement a total of about 66,000 methods. We found 6248 unique methods. We further filtered

[1] In Smalltalk all methods return a value which is by default 'self'; we exclude those methods that do not specify an explicit return.

Class	Selector	Calls
Parcel	#markAsDirty	23
ParagraphEditor	#resetTypeIn	19
UIPainterController	#broadcast PendingSelectionChange	18
CodeRegenerator	#pushPC	15
AbstractChangeList	#updateSelection:	15
PundleModel	#updateAfterDo:	10

Table 2: Classes,selectors and number of times called of identified aspects

these down by pruning methods which are part of the 'accessing' protocol (these are clearly not aspects) and taking scattering into account. A good approximation heuristic for scattering is the number of times the method is called. While this does not take into account the actual spread of the calls throughout the code, it can be used to weed out at least auxiliary private methods[2]. We then manually browsed through this list of methods to identify methods for which it would make sense to apply a method-to-advice refactoring, thus those methods implementing an aspect.

In the experiment we limited ourselves to unique methods which were called at least five times. Computing this list took about 2 minutes[3] and resulted in 228 methods, a number small enough to allow us to manually inspect and identify possible aspect candidates in a reasonable amount of time.

Table 1 shows a number of unique methods which remained after filtering. When just looking at the selectors of these methods, the inclusion of the words "update" and "event" make `#updateEvent:` stand out as a very likely aspect candidate. Methods such as `#addItem:value:` on class `Menu` seem less likely to implement aspects, as it more clearly refers to adding items on a menu bar. We similarly went through the list of the other methods of which many are clearly accessors, mutators or in other ways part of the base functionality of the classes. We wound up identifying a remainder of about 16 candidate aspects.

Table 2 gives a few examples of aspects we identified. The majority of the aspects we found, like for instance `#updateAfterDo:`, `#broadcastPendingSelectionChange` and `#updateSelection:` implement a *state change updating* mechanism. The method `#markAsDirty` is an example of a *cache invalidation* aspect. Surprisingly in Smalltalk, we encountered a *memory management* aspect in the library for connecting with C programs, the method `#beGarbageCollectable`.

Despite our success in identifying a number of aspects, our unique methods list did to our surprise not include the `#changed:` method which implements the state change notification aspect. It turns out there are two other methods which override the central `#changed:` method of the Object class, in both cases this appears to be an implementation 'hack' to make the objects observe themselves, essentially implementing an 'after' advice on a number of methods.

As most aspect candidates we identified are called ten times or more, we could have further filtered down the list to unique methods that are called more than 10 times, which leaves only 124 methods to be inspected. However, this would leave at least 4 aspect candidates which are called no more than 10 times undetected.

5 Discussion

The 'unique methods' heuristic we presented above is a light-weight approach for supporting the identification of aspect candidates in legacy code: it limits the developer's attention to a relativily small number of methods which are more likely to be implementations of aspects based on the observation that the 'unique methods' pattern is commonly applied in manually woven aspect implementations.

[2]Note that Smalltalk does not have any language construct for declaring methods to be private, though this status is commonly indicated by classifying them in a 'private' protocol

[3]On a PowerBook G4 667 Mhz

The heuristic obviously has its drawbacks as clearly evidenced by the `#changed:` method case. We expect there might be other cases of unique methods which are not *entirely* unique: it is to be expected that in the case of a 'logging aspect' for example, there might be two alternative logging classes, one for logging to disk and one for logging to screen each with their own `#log:` method. It is not clear however how the heuristic can be changed to also detect these cases while not reporting every single framework method as an aspect candidate.

Another obvious drawback of the approach is that it is very light-weight and still involves the developer guessing at whether a unique method would be better implemented as an advice. As indicated in the previous section, we did this mostly on the basis of certain keywords such as 'mark', 'update' etc. occuring in the names of methods. Thus another topic we'd like to put up for discussion at the workshop is whether it would be worthwhile to incorporate dictionaries or ontological techniques in aspect identification techniques.

As there is currently no other report available on applying an aspect identification technique to the entire code base of a Smalltalk image , we can give little comparison as to the relative success of our heuristic. A comparison with techniques based on detecting code duplication is necessary to see whether the two are complementary as we hypothesize.

6 Related Work

While the field of aspect identification is fairly young, a number of approaches are already under investigation. One group of approaches relies purely on tools which aid a developer in manually browsing the code while looking for crosscutting concerns. Examples of such approaches are Concern Graphs [11], Aspect Browser [4], Aspect Mining Tool [6] and JQuery [7]. Another group of approaches aim to automatically detect aspects. A number of these are based on the idea of detecting code duplication: Tourwé and Mens [14] apply the technique of Formal Concept Analysis, Bruntink [2] proposes the use of *Clone Class metrics* and Shepherd et al. [12] use PDG-based clone detection. Other approaches that aim for automation, as is our own, are based on the idea of detecting scattering: Marin et al. [10] perform fan-in analysis while Krinke [8] proposes the technique of control-flow-graph-based aspect mining. Other techniques like Greevy [3], Breu [1] and Tonella [13] consider dynamic information in order to identify candidate aspects. In practice full automatic detection of aspects is impossible, and the second group of approaches is used to complement the first by pointing out potential aspects or "seeds" from where to start browsing [10].

We've essentially used fan-in analysis [10] as our approximate heuristic for measuring scattering. A difference with the work of Marin et al. is that we've applied the analysis only to unique methods, as our interest is mostly in identifying candidates for applying a method-to-advice refactoring [5], while Marin et al. use it for detecting good "seeds" from where to start browsing which can help find larger aspects such as support for undo operations in a graphical editor which requires more complex refactorings to extract.

7 Conclusion

In this paper we presented a light-weight heuristic for identifying aspect candidates in legacy code. We applied our heuristic to an entire Smalltalk image and were able to detect a number of aspects. The heuristic is based on the observation that in pre-AOP days certain crosscutting concerns were implemented by 'manually weaving' aspects as method calls to a central entity.

The goal of this paper is two-fold: apart from presenting an experience report about identifying aspects in Smalltalk we also want to point out our position and a number of interesting topics for discussion:

- Are 'unique methods' a good technique to identify aspects which are implemented as calls to a central entity?
- Are other techniques such as code duplication techniques usable for detecting aspects implemented as calls to a central entity? Or is our 'unique methods' heuristic complementary to these?
- Would it be interesting to use techniques from ontology research to help identify aspects?

Acknowledgments

We would like to thank Kim Mens (UCL) for proofreading this paper.

References

[1] S. Breu. Towards hybrid aspect mining: Static extensions to dynamic aspect mining. In *1st Workshop on Aspect Reverse Engineering*, 2004.

[2] M. Bruntink. Aspect mining using clone class metrics. In *1st Workshop on Aspect Reverse Engineering*, 2004.

[3] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. To appear in proceedings of the 9th European Conference on Software Maintenance and Reengineering.

[4] W. Griswold, Y. Kato, and J. Yuan. Aspect browser: Tool support for managing dispersed aspects. In *First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems - OOPSLA 99*, 1999.

[5] K. Gybels and A. Kellens. An experiment in using inductive logic programming to uncover pointcuts
an experiment in using inductive logic programming to uncover pointcuts. In *First European Interactive Workshop on Aspects in Software*, 2004.

[6] J. Hannemann. Overcoming the prevalent decomposition of legacy code. In *Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering 2001*, 2001.

[7] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *International Conference on Aspect Oriented Software Development 2003*, 2003.

[8] J. Krinke and S. Breu. Control-flow-graph-based aspect mining. In *1st Workshop on Aspect Reverse Engineering*, 2004.

[9] C. I. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, nov 1997.

[10] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Working Converence on Reverse Engineering (WCRE)*, 2004.

[11] M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *International Conference On Software Engineering 2002*, 2002.

[12] D. Shepherd, E. Gibson, and L. Pollock. Automated mining of desirable aspects. In *International Conference on Software Engineering Research and Practice*, 2004.

[13] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *11th IEEE Working Conference on Reverse Engineering*, 2004.

[14] T. Tourwé and K. Mens. Mining aspectual views using formal concept analysis. In *Source Code Analysis and Manipulation Workshop (SCAM)*, 2004.