Fifth FTW PhD Symposium

Faculty of Engineering, Ghent University

1st December 2004

Aspect Oriented Design and Legacy Software

Helping Old Music to a Fresh Beat

Kris DE SCHUTTER

Promoter(s): Prof. Dr. Ir. Ghislain Hoffman

Abstract— **Today's industry is fraught with problems facing legacy software: outdated code, patched systems, and poor, incomplete documentation. In general knowledge and expertise regularly disappear. The consequences, however, may be dire unless that same —and often mission-critical— software keeps up with the times. It has been proposed that the advances made by the Aspect Oriented paradigm may help alleviate the woes of long-term software development and maintenance. Unfortunately, Aspect Orientation (AO) has yet to be truly applied outside its comfortable home of Object Orientation. In this paper we show some tentative steps in bringing AO to the realm of ageing languages using a framework based on an unlikely mix of grammars, XML, and declarative meta-programming. We have already validated this framework to some extent in the context of two old mastodonts: ANSI-C and COBOL.**

Keywords— **legacy software, aspect orientation, evolution, code instrumentation**

I. Context

THERE exists a well known principle in evolution called the *Red Queen Equilibrium*, which owes its name to Lewis Carroll's famous book "*Alice through the looking glass*". In it Carroll has the Red Queen remark: 'here, you see, it takes all the running you can do, to keep in the same place.'

This principle reflects that in order to keep up with an evolving environment you have to keep investing energy to maintain your place in the foodchain. It does not matter whether one finds himself in the animal kingdom or in the competitive world of modern industry. One either keeps up or becomes extinct.

Over the course of the last decade the digital world has seen some great leaps in its pace, most notably through the popularization of the internet. This has opened up new opportunities for businesses, even to the extent that every service provider is clamoring to get its products online.

Unfortunately, these services rely on software which was not necessarily conceived with such an interactive environment in mind. In fact, we find that most of it was —and often still is— written in COBOL and targeted to a mainframe environment [1]. Worse still, very little knowledge about the internals of such applications survives. The reason is no surprise:

- a general lack of up-to-date documentation,
- developers moved on to other projects, or even left the firm,
- fixing systems by patching them has obscured the code, and
- why fix it if it is not broken ?

II. Aspect Orientation

IT is in this context that we find ourselves musing about Aspect Orientation. But what exactly is it?

AO is a relatively new paradigm, which has grown from the limitations of Object Orientation [2]. OO takes an object-centric view to software development, where a programmer describes objects, how they should behave and in what ways they should interact with other objects.[1]

The trouble with OO is very simple: *some concepts can not be cleanly captured in an object.* Consider for instance logging —which has become *the* hallmark of AO—. All objects and actions needing to be logged have to actively participate in order to achieve this goal. Any implementation of logging will therefore be spread out over the entire application, which makes it hard to implement, and harder still to maintain.

AO proposes the concept of *aspects* to solve this dilemma. Aspects allow us to *quantify* which events in the flow of a program interest us (through so-called *pointcuts*), and what we would have happen at those points (through *advice*). Hence we can 'describe' what logging means to an application and have the *aspect-weaver* (a compiler for aspects) do the rest.

III. What AO means to legacy applications

CONSIDER being faced with an application you are hesitant to touch; you may not understand enough of the application even while it is mission critical to your company. Changing it may be an unacceptable risk.

Consider then possessing a technology which allows you to impact an application from the outside, in a way that the original application is left unharmed. That's right: Aspect Orientation can do this for you.

Aspects have the property that they can be 'switched off' in an application simply by not including them. This is not just a neat trick: *obliviousness* is a key property of any good AO language. It implies that your code does not have to be aware of the presence of aspects.

This is what AO means to legacy applications: assisting in the evolution of such applications, but also being able to instrument them in order to retrieve lost knowledge, all the while knowing that anything we do can easily be undone again.

IV. What legacy applications may mean to AO

IF the previous promise sounds too good to be true then know that it actually is for the time present. AO has been born out of needs present in OO, and has since its inception rarely left the nest. The end result is that if you feel like playing with AO you either have to do it in one of the recent OO languages —Java being the most prominent subject[2]— or you have to build your own weaver.

The goal of our research is therefore to enable one to do the latter as easily as possible. What's more, we hope that by seeing

K. De Schutter is with the Department of Information Technology, Ghent University (UGent), Gent, Belgium. E-mail: Kris.DeSchutter@UGent.be.

[1] While typically achieved by constructing a hierarchy of classes, we should point out that the problems presented here are not limited to class-based OO.

[2] AspectJ, http://eclipse.org/aspectj/

how AO lives and breathes inside legacy environments we may get a better understanding of the true nature of AO and its limits.

V. The framework

TO all of these ends we have set up a framework of tools based on a mix of grammars, XML and declarative meta-programming: *Yerna Lindale*[3].

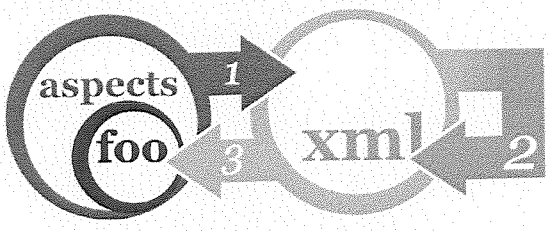

Fig. 1. The process of weaving.

Let us then explore this recipe for bringing AO into a legacy language. We refer to figure 1 for a high level view of the process.

A. Extend the grammar to allow for aspects.

The proces starts off by extending the grammar of a language —e.g. 'foo' in figure 1— in such a way that it accepts aspectual descriptions. This results in a superset of the original language, which we generally name '*AspectFoo*'.

B. Generate a parser/translator to XML.

Having established an extended grammar we can easily generate a working parser from it using the *Grammar Deployment Kit*[4], originally developed at the Vrije Universiteit Amsterdam. This parser is extended by the framework to allow it to dump the parsed sources to an XML format (figure 1, transition 1); the exact format being a function of the original language.

The choice for XML has been made for several reasons, the most important being that XML representations are much easier to handle directly than the original sources.

C. Set up analysis and transformation.

Using a logic language[5] we can now start to reason on the XML structure in order to detect all possible *joinpoints* (what AO generally calls events in the program flow). We will also query for all advices and their associated pointcuts.

Once all data has been gathered we can start weaving: for every pair of joinpoint and matching advice we transform the xml in such a way that the intent of the advice is fulfilled at that joinpoint —this is the hard part—. In the end we output the result back to file (figure 1, transition 2).

D. Translate XML back to the original language.

This last step is pretty straightforward, and is taken care of by the framework (figure 1, final transition). With the transformation process having woven away all our aspect definitions we end up with a source file conforming to the orginal unextended language. It no longer carries any advice or pointcuts within it, and can thus be compiled into an executable version.

[3]Quenya (High Elvish) for '*old music*' —this for the fans of the works by J. R. R. Tolkien (among them the author of this paper, obviously).

[4]http://gdk.sourceforge.net/

[5]PrologCafe, http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/

VI. Summing up

TAKE both source-to-xml and xml-to-source transformations. Add to that the ability of weaving inside xml, and by extension you find you have been weaving within the original source.

Using this framework we have already built a succesful AspectC [3] (example in figure 2), and are now in the process of setting up a basic AspectCobol (teaser in figure 3) as well.

```
int advice on (.*_i) && ! on (print.*) {
  int i = 0;
  printf ("before %s\n", this_joinpoint()->name);
  i = proceed ();
  printf ("after %s\n", this_joinpoint()->name);
  return i;
}
```

Fig. 2. Example of an advice in AspectC.

XML transformations need however not be limited to the *weaving* of aspects. They may also be considered for such tasks as the extraction of business rules from code [4], the assessment of code-quality, cleaning up of code, etc.

```
TRACING-ADVICE SECTION.
  USE AROUND JOIN-POINT
     ( IS READ-STATEMENT
       OR IS WRITE-STATEMENT
     ) AND NOT WITHIN "TRACING".
LOGGING-PARAGRAPH.
   DISPLAY "BEFORE A FILE MANIPULATION"
   PROCEED
   DISPLAY "AFTER A FILE MANIPULATION".
```

Fig. 3. Example of an advice in AspectCobol.

VII. Conclusions

WE have shown that a declarative approach to the weaving of legacy applications based on an intermediate XML representation is a very flexible and very manageable solution. The main concepts in AO —quantification and obliviousness— remain valid within the context of non-OO languages. The differences seem to be limited to the *kind* of events in the flow of an application an aspect may be applied to.

Acknowledgments

Prof. Hoffman and prof. Tromp for making this PhD possible. Also everyone at ARRIBA for setting the stage in which this PhD takes place, and for the challenges they bring to it.

References

[1] Isabel Michiels, Herman Tromp, et al. *Identifying Problems in Legacy Software: Preliminary Findings of the ARRIBA Project.* ELISA workshop at ICSM, 2003.

[2] Gregor Kiczales, John Lamping, et al. *Aspect-Oriented Programming*, In proceedings of ECOOP, 1997.

[3] Stijn Van Wonterghem. *Aspect-Oriëntatie bij procedurale programmeertalen, zoals C.* Master's thesis, Gent 2004. *(In Dutch.)*

[4] Isabel Michiels, Kris De Schutter. *Using Dynamic Aspects to Extract Business Rules from Legacy Code.* Dynamic Aspects workshop at AOSD, 2004.