

Aspect-Oriented Programming

Case Study: System Management Application

Kai Böllert
Elzacher Str. 10
71034 Böblingen, Germany
Tel. +49-7031-277246
kaib@acm.org

Abstract

This position statement presents a problem description and first results of a thesis I will present to the Fachhochschule Flensburg, Germany, in partial fulfillment of the requirements for the degree of *Diplom-Wirtschaftsinformatiker (FH)*¹. I am writing this thesis in cooperation with Hewlett-Packard GmbH, Böblingen, Germany.

1 Introduction

Development of server applications is by no means an easy task. Several requirements such as multi-user capability, central or distributed data repository, fault tolerance and graphical user interface must be thoroughly analyzed, designed and implemented.

The system management application the thesis refers to is no different. However, despite using the development platform VisualWorks, a modern implementation of the object-oriented programming language Smalltalk, a look at the source code reveals the known code tangling phenomenon as a result of various aspects that cross-cut subsystem boundaries.

In the thesis two of these aspects – typical for most server applications – will be analyzed:

- synchronization of processes who concurrently access shared objects and
- tracing the program's control flow.

Based on the results of the analysis phase two aspect languages will evolve that will be used by application developers when implementing future versions of the software product. In addition, the necessary foundation for aspect-oriented programming must be implemented in VisualWorks. The thesis will end with an evaluation of the benefits of using AOP in the system management application. The evaluation will consist of code reviews, code metrics and performance measurements.

The remaining sections of this paper highlight areas of completed work from the point of view of Smalltalk, the programming language used.

¹ *Wirtschaftsinformatik* is a combination of Computer Science and Business Science

2 Analyzing Aspects

2.1 Synchronization Aspect

Much analysis of the synchronization aspect was already done by Cristina Lopes in [Lop97] and resulted in the coordination aspect language COOL. Basically, COOL allows the definition and configuration of coordinators which perform the synchronization for instances of specified classes on method level. Each method can have synchronization properties like self exclusion, mutual exclusion with other methods, a guard condition and entry as well as exit statements.

2.2 Tracing Aspect

Few software products today are delivered without errors. When customers report unexpected software behavior, it is usually the support team's task to deal with those malfunctions. This task can be divided into three steps:

1. locate the subsystem in which the problem may reside,
2. find out what is going on/wrong in this area, and
3. fix the problem.

Each of these steps has its own inherent difficulties, but what makes finding bugs at the customer's site even more difficult is that server applications written in Smalltalk are normally deployed in headless mode, i.e. without components such as class browser and debugger that are available while running the application in the development environment. Hence, there is a need for debugging aids under these circumstances.

One aid is tracing the program's control flow. Traditionally, code for generating trace information is implemented with the main program code and surrounded by conditional compilation symbols, so that tracing may be left out at compile time to prevent a negative impact on performance. However, this additional code makes the original source code much more difficult to follow and, hence, maintain. Moreover, missing tracing conventions, e.g. a fixed output format, lead to unhelpful trace output, and without introducing trace levels the amount of output quickly reaches a critical mass.

Using aspect-oriented programming techniques, the code required for tracing can be easily separated from the component code. Like conditional compilation, aspects can be woven to the program on-demand, if necessary during runtime. Additionally, a declarative approach for defining what to trace is realizable, for instance – from the Smalltalk perspective – trace only invocations of methods in certain protocols or trace the values of specific instance variables. The declarative approach can also be used to automate assignment of trace information to trace levels.

3 Designing Aspect Languages

Naturally, developers are most familiar with the programming language they use for their current project. Thus, aspect languages that are syntactically closely related to this programming language will be easier to learn for developers and will achieve greater acceptance.

When designing aspect languages for Smalltalk, the designer has to be aware that developers like Smalltalk not only because of the rich class library but also because of the simple syntax that almost reads like natural language. Therefore, the specification of an easy-to-read method protocol (API) for programming aspects should be preferred over defining a whole new aspect language syntax or introducing reserved keywords into the Smalltalk syntax (see Figure 1).

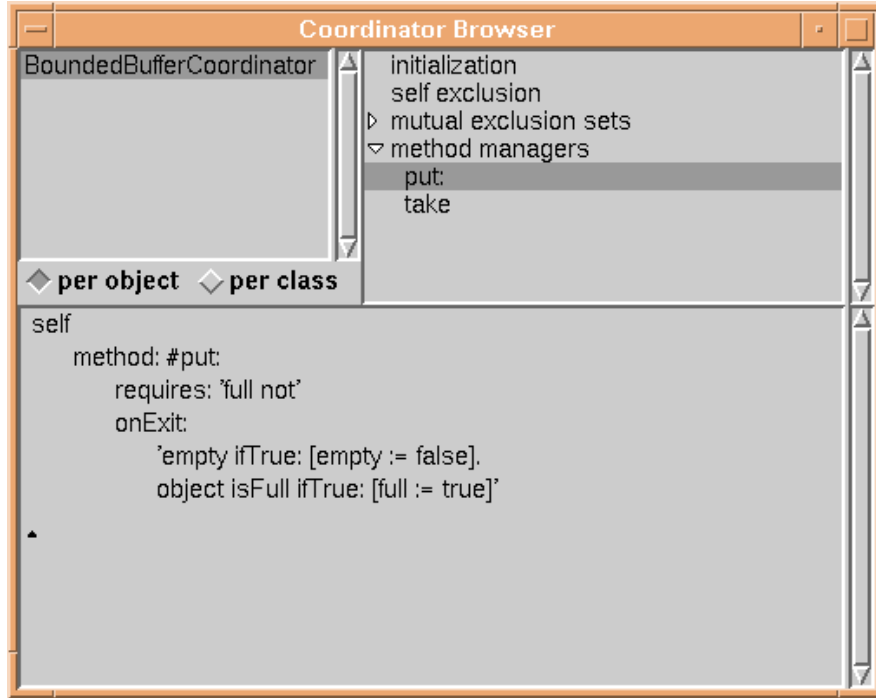


Figure 1: Coordinator Browser for defining and configuring coordinators by method protocol

4 Implementing an Aspect Weaver

In general, the task of an Aspect Weaver is to weave additional behavior in the form of aspects into classes. Instances of those woven classes will then behave differently.

By exploiting Smalltalk's broad reflection protocol which covers introspection as well as intercession it is possible to do the weaving without any change to the developer's source code. Thus, the developer does not have to debug through woven code but can instead concentrate on his own code. Furthermore, if he finds an error and corrects the code, the Aspect Weaver's work will remain in place untouched.

A possible way for implementing the above feature is to add the concept of lightweight classes to Smalltalk. A lightweight class inherits state and behavior from its superclass. While it adds no further state, it can override and thus adjust behavior if necessary. Normally, no instances of lightweight classes are created. Instead, lightweight classes are used to change the behavior of instances by changing the instances' class to the lightweight class. As mentioned before, this change in the behavior of an instance is exactly what an Aspect Weaver wants to achieve.

If the Aspect Weaver must weave an aspect into a class, it first creates a lightweight subclass of this class. This new subclass is called the woven class. Aspects know what methods will be affected by them and the Weaver overrides these methods in the woven class (see Figure 2). Such a woven method typically places a wrapper around the invocation of the original, inherited behavior. For example, if the woven aspect is a synchronization aspect, the wrapper can do the synchronization itself or can delegate the task to another object. The final part of the Weaver's work is to change the class of instances of the woven class' superclass to the woven class and to set up a mechanism, so that the same will happen for each newly created instance.

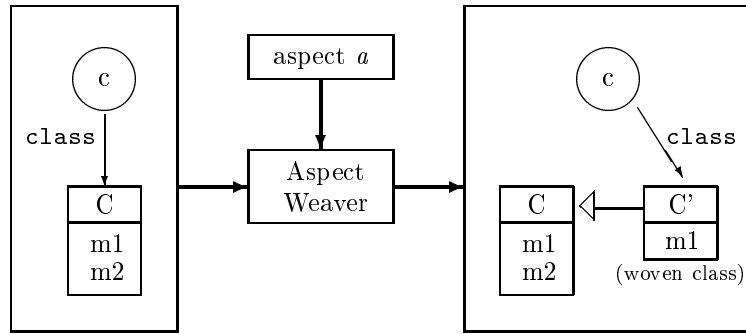


Figure 2: Aspect a is woven into method $m1$ of class C with instance c

For the previous synchronization example the situation for woven instances upon receiving a message is as follows: if the message needs to be synchronized, the method lookup will find the woven method and invokes it; the woven method performs the synchronization and calls the original method implementation (compare also Figure 3). In case the message needs not to be synchronized, the method lookup simply skips the woven class and invokes the implementation found in the developer's class.

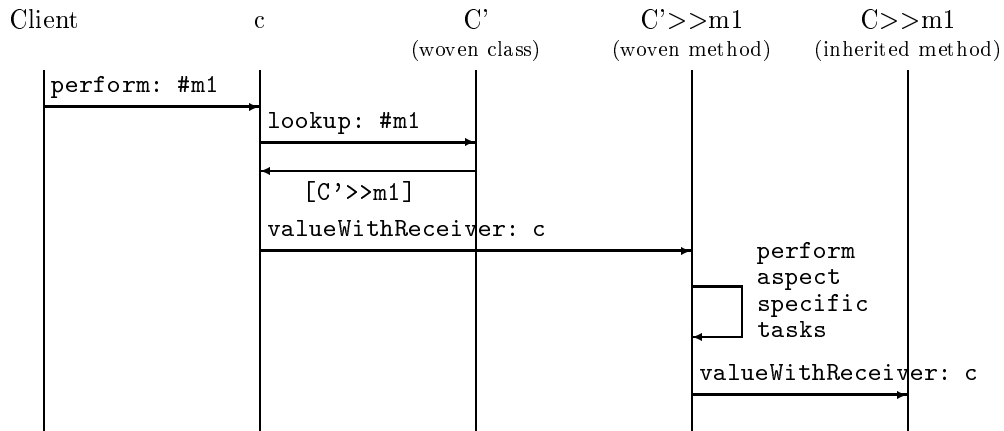


Figure 3: Sending message $m1$ to instance c

References

[Lop97] Cristina Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, 1997.