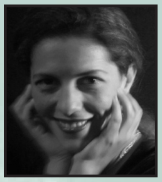
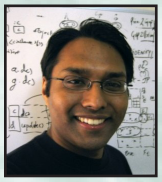
XE (eXtreme Editor) - Tool Support for Evolution in Aspect-Oriented Programming

Wiwat Ruengmee
University of California, Irvine
wruengme@ics.uci.edu

David F. Redmiles
University of California, Irvine
redmiles@ics.uci.edu

Roberto Silveira Silva Filho
University of California, Irvine
rsilvafi@ics.uci.edu


Cristina Videira Lopes
University of California, Irvine
lopes@ics.uci.edu


Sushil Krishna Bajracharya
University of California, Irvine
sbajrach@ics.uci.edu

June 2008

ISR Technical Report # UCI-ISR-08-1

XE (eXtreme Editor) - Tool Support for Evolution in Aspect-Oriented Programming

Wiwat Ruengmee, Roberto Silveira Silva Filho,
Sushil Krishna Bajracharya, David F. Redmiles, Cristina Videira Lopes
{wruengme, rsilvafi, sbajrach, redmiles, lopes}@ics.uci.edu

Department of Informatics
Donald Bren School of Information and Computer Sciences, University of California
Irvine, CA, 92697 USA
ISR Technical Report # UCI-ISR-08-1
June 2008

Abstract. In spite of the modularization benefits supported by the Aspect-Oriented programming paradigm, different usability issues have hindered its adoption. The decoupling between aspect definitions and base code, and the compile-time weaving mechanism adopted by different AOP languages, require developers to manage the consistency between base code and aspect code themselves. These mechanisms create opportunities for errors related to aspect weaving invisibility and non-local control characteristics of AOP languages. In short, AOP developers lack adequate support for: 1) visualizing and identifying the exact points in the code where aspects are woven; 2) preventing aspect-base code inconsistencies, and 3) evolving aspect-oriented code in a coherent way. This paper describes XE (Extreme Editor), an IDE that supports developers in managing these issues in the functional aspect-oriented programming domain. We validate our approach through a case study showing how XE reduces the cognitive effort of developers in evolving AOP programs.

XE (eXtreme Editor) - Tool Support for Evolution in Aspect-Oriented Programming

Wiwat Ruengmee, Roberto Silveira Silva Filho,
Sushil Krishna Bajracharya, David F. Redmiles, Cristina Videira Lopes
{wruengme, rsilvafi, sbajrach, redmiles, lopes}@ics.uci.edu

Department of Informatics
Donald Bren School of Information and Computer Sciences, University of California
Irvine, CA, 92697 USA
ISR Technical Report # UCI-ISR-08-1
June 2008

Abstract. In spite of the modularization benefits supported by the Aspect-Oriented programming paradigm, different usability issues have hindered its adoption. The decoupling between aspect definitions and base code, and the compile-time weaving mechanism adopted by different AOP languages, require developers to manage the consistency between base code and aspect code themselves. These mechanisms create opportunities for errors related to aspect weaving invisibility and non-local control characteristics of AOP languages. In short, AOP developers lack adequate support for: 1) visualizing and identifying the exact points in the code where aspects are woven; 2) preventing aspect-base code inconsistencies, and 3) evolving aspect-oriented code in a coherent way. This paper describes XE (Extreme Editor), an IDE that supports developers in managing these issues in the functional aspect-oriented programming domain. We validate our approach through a case study showing how XE reduces the cognitive effort of developers in evolving AOP programs.

Key words: Programming environment, Aspect-Oriented Programming, software development, program understanding, program transformations.

1 Introduction

Aspect-Oriented Programming (AOP) is a methodology that aims to improve the modularity of software systems by encapsulating scattered and tangled code into distinct abstractions called aspects [16]. Aspect abstractions comprise two basic components: the aspect behavior, or advice, and a point cut descriptor (or PCD), which provides references to runtime conditions and the places in the base code where aspect behaviors are woven. Through this mechanism, code weavers combine aspects and base code to form a final program. This programming model promotes the separation of aspect code and base program, allowing their independent modification. From a usability perspective, however, the decoupling between aspects and base code, and the PCD/weaving mechanism raises different problems experienced by many AOP developers [20].

The idea of obliviousness to AOP development that would "allow programming by making quantified programmatic assertions over programs written by programmers oblivious to such assertions" [7] has been shown to be flawed [21]. This separation actually adds considerable complexity to aspect-oriented development. Popular general purpose AOP languages such as AspectJ [15] employ a regular-expression based PCD language that, while very powerful and general, can lead to problems such as over-weaving (when aspects are woven to wrong parts of the code) or under-weaving (when aspects are not woven with the code they should). For example, in AspectJ, by defining a point cut described as *paint*(..)* to implement a GUI refresh aspect, developers can unintentionally advise base code methods such as *paintBrushConfig()*. In these situations, neither the weaver nor the compiler can detect these semantic mistakes that can only be perceived at runtime, when the program behaves erroneously.

Another common problem faced by AOP developers is the lack of support for evolution. A simple method rename can break the PCD-base code contract, removing the method from the set of point cuts advised by an aspect. Moreover, semantic changes in the code may also require adjustments to the aspect-base code contract. For example, the addition of a new GUI widget may require a different type of refresh implementation. In order to support this new behavior, the new repaint method needs to adopt a different name convention from the standard paint() command. Another option is to update the PCD to exclude that particular method from its advice. Both options, however, usually require successive trial-and-error compilation cycles to assure the developer that the change does not impact other methods in the system. It also requires many context switches between base code and aspect code to verify the correctness of the PCD descriptors. Both activities have shown to increase the cognitive load of the developers, leading to potential programming errors [14].

These issues reveal a more fundamental problem in the AOP programming model: the lack of support for developers to detect side-effects of semantic changes in the program. Such common mistakes are syntactically correct, and cannot be automatically detected by the aspect weaver nor by the programming language compiler. Therefore, they are only detected at runtime, when an erroneous behavior manifests itself, an error-prone and tedious process.

In this paper, we present XE, an Integrated Development Environment (IDE) that mitigates these usability issues, originated by the lack of feedback to AOP programmers. By supporting automatic edit-time aspect weaving, XE supports developers in assessing the impact of changes they make in either the aspect or the base code, and helps in the visual detection of both over and under-matching conditions. In doing so, it also avoids the need for repetitive context switches from base to aspect code, which increases the developers' cognitive load. Moreover, in XE both base code and aspect implementations are kept consistent trough the use of an underlying relational model and engine. This feature supports refactoring and evolution through the automatic adjustment of the aspect visualization in response to code changes, and by permitting the end-user to select between different aspect implementations to weave. Finally, through the use of different views, larger-scale projects and debugging are also supported.

XE was developed in PLT Scheme, which was selected for its extensibility, and for the ability of the Scheme language and environment to support runtime code manipulation. These features significantly simplified our prototype implementation.

The rest of the paper is organized as follows: Section 2 briefly introduces the aspect-oriented programming concepts in Scheme used throughout this paper. Section 3 presents a motivating scenario, showing the problems of evolving AOP code without appropriate IDE support; Section 4 describes the main features of the XE IDE. Section 5 discusses XE implementation details; Section 6 shows how XE can be employed to better support developers in evolving AOP programs. Section 7 presents the results of our evaluation, showing the cognitive benefits of using XE tool. Section 8 presents related work and Section 9 concludes.

2 Aspects in Scheme

In this section, we present the pointcut and advice models we developed for Scheme, showing how they are used to implement aspects in the functional programming paradigm. Our AOP language mimics the operators of AspectJ [21].

The aspect definition in Scheme consists of a list of advice expressions, each one binding a set of join points to an interceptor expression (typically a function call in Scheme base program) as shown in Figure 1. The `<advice-type>` can be any of the types supported in AspectJ, namely before, after and around. A query, `<jp-query>`, matches a list of join points of the base program using a regular expression on program identifiers. Two pointcut types are supported: call (representing a function call) and exec (representing a function execution). Sub-expressions inside a query can be combined using logical operators 'and', 'or', and 'not'. Finally, `<interceptor-exp>` represents a set of Scheme expressions.

```
(aspect <aspect-arguments-list>
        (<advice-type> <jp-query> <interceptor-exp >)
        ...
        (<advice-type> <jp-query> <interceptor-exp >))
```

Fig. 1. Scheme aspect meta-model

The current implementation of XE aspect extensions is based on the subset of the Scheme language implemented by PLT Scheme, named DrScheme[1]. The interpreters used in XE are derived from those implemented in Abelson and Sussman's book [1]. Hence, XE extends PLT Scheme to support aspects as described in Figure 1.

3 Motivating Scenario

In this section, we illustrate the use of aspects in XE Scheme, showing a simple example of the problems faced by aspect-oriented developers in 1) identifying the exact points in the code where aspects are woven; 2) detecting aspect-base code inconsistencies, and 3) evolving aspect-oriented code in a coherent way.

3.1 Simple banking transaction API

Our Scheme aspect extension was used to modularize a simple banking transaction API. The original API had three major operations: deposit, withdraw, and balance inquire. This API was later evolved to support new features as discussed in this section. We present the steps necessary to modularize and evolve this banking application without any extra XE support. The only available mechanisms are the weaving and execution of XE Scheme code. Our goal is to mimic the functionality provided by existing AOP weavers and compilers, where no extra IDE support is provided.

In the original code, the authentication concern was scattered through the program, individually handled by each function of the API as seen in Figure 2.

This code was modularized, within an authentication aspect, by factoring out the permission-checking code from the API methods, as shown in Figure 3, and by creating an authentication aspect in XE Scheme as shown in Figure 4. Both base code and aspect code reside in separate files.

After separating a concern into base code and aspect code, the developers need to reintegrate (or weave) the aspect and base code behaviors by 1) loading both base code and aspect code files into the XE environment, and 2) manually weaving the aspect code with the base code through a menu command in XE Scheme. Section 4 explains the weaving and unweaving processes in XE Scheme in more detail.

[1] http://www.drscheme.org

```
;;(1) Deposit procedure definition
(define (deposit)
   (lambda ()
      (if (not (authenticate-user-db
                  (get-username)
                  (get-password)))
          (error "Authentication␣Failure")
          false))
      (run-deposit 2000)))

;;(2) Withdraw procedure definition
(define (withdraw)
   (lambda ()
      (if (not (authenticate-user-db
                  (get-username)
                  (get-password)))
          (error "Authentication␣Failure")
          false))
      (run-withdraw 2000)))

;;(3) Balance procedure definition
(define (balance)
   (lambda ()
      (if (not (authenticate-user-db
                  (get-username)
                  (get-password)))
          (error "Authentication␣Failure")
          false))
      (run-show-balance)))
```

Fig. 2. Original code with scattered authentication mechanisms

```
(define (deposit)
    (lambda ()
      (run-deposit 2000)))

(define (withdraw)
    (lambda ()
      (run-withdraw 1000)))

(define (balance)
    (lambda ()
      (run-show-balance)))
```

Fig. 3. Base code without the factored-out authentication code

```
(define authentication-db-aspect
  (aspect ()
    ((before
      (call
        (or
          (inside deposit *)
          (inside withdraw *)
          (inside balance *)))
      (begin
        (if (not (authenticate-user-db
                   (get-username)
                   (get-password)))
            (error "Authentication␣Failure")
             false))))))
```

Fig. 4. Database authentication aspect definition

3.2 Evolving and debugging the application

The original API was very simple; it supported screen-only balance display and the authentication was based on information stored in a local database (password file). In order to broaden its applicability, the API was extended to support new functionality: 1) a new print balance function that directs balance inquiry results to a network printer, and 2) a distributed (web-based) authentication mechanism that grants access to the network printer. The steps for this course of evolution were:

1. The first step towards the evolution of the banking API is to implement the new `run-print-balance` function that sends the balance inquire output to a printer.
2. The next step is to extend the balance definition to invoke this function whenever a balance is requested. The balance is then printed both to the screen and to the printer (the result code is shown in Figure 5 (1)). After this evolution step, the program becomes semantically incorrect. The `(inside balance *)` interceptor, which directs the weaving of the database authentication code with the base code, will incorrectly advise `(run-print-balance)` with the `authenticate-db-user` code. We call this situation "over-matching".
3. After a weaving/execution/testing cycle, where the wrong behavior becomes evident, a developer corrects this problem by modifying the wildcard in `(inside balance *)` to a more specific, `(inside balance run-show-balance)`, as seen in Figure 5 (2). This modification, however, creates another semantically inconsistent situation that we call "under-matching". After this change, the `run-print-balance` is no longer advised by any aspect, while it should have been advised by the new `authentication-ws-aspect` aspect.
4. After another weave/execute/test when the print jobs could not be completed due to the lack of authentication, the developers detect their mistake. They then create the new aspect, namely `authentication-ws-aspect`, to fix this problem. Let's suppose that they use a copy-and-paste programming strategy to reuse the old authentication aspect, the `authenticate-user-db`, as a template for the new `authenticate-user-ws`. In this process they forget to remove the (inside balance *) and (inside withdraw *) PCD expressions. This common mistake would again result in "over-matching" of other function calls inside

```
;;(1) Added new procedure
(define (balance)
    (lambda ()
      (run-show-balance)
      (run-print-balance)))

;;(2) Modified old aspect
(define authentication-db-aspect
  (aspect ()
    ((before
      (call
        (or (inside deposit *)
                (inside withdraw *)
                (inside balance run-show-balance)))
        (begin
          (if (not (authenticate-user-db
                    (get-username)
                    (get-password)))
            (error "Authentication Failure")
             false))))))

;;(3) New aspect
(define authentication-ws-aspect
  (aspect ()
    ((before
      (call
        (inside balance run-print-balance))
        (begin
          (if (not (authenticate-user-ws
                    (get-username)
                    (get-password)))
            (error "Authentication Failure")
             false))))))
```

Fig. 5. Updated base and aspect code

balance and withdraw. Moreover, a feature interference condition would also occur with the existing `authenticate-ws-aspect`.

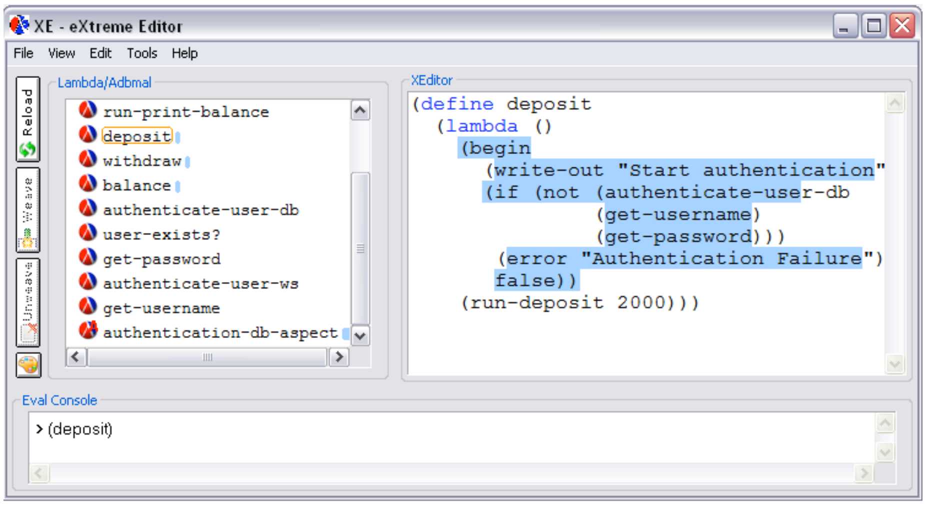
5. At last, after removing the extra PCD expressions, the evolution step is complete and the program is correct.

In the previous scenario, we showed how a simple evolution step can result in different inconsistent code stages that can result in syntactically correct, but semantically wrong programs. Additionally, it required repetitive switches of context from the base code to the aspect code, followed by repetitive weaving/execution cycles. The root of the problem is the lack of mechanisms to support developers in identifying these situations. In this paper, we argue for the need of instant-feedback mechanisms as supported by XE IDE.

4 XE IDE Features

In order to address the types of problems described in the last section, we developed XE (Extreme Editor), an IDE that supports developers in the evolution, refactoring, and debugging of aspect-oriented programs (see Figure 6 and Figure 7). As such, XE combines different visualizations, automatic tracking of aspects and base code, and edit-time aspect code weaving. Combined, these features help developers overcome the usability problems induced by the AOP model. The main features of XE, and their benefits, are described in this section.

Figure 6 and Figure 7 present the main screen of XE. In Figure 6, different aspect and lambda definitions are shown. In Figure 7, the main editor shows the base code with any woven aspect that the user may have selected. The Eval console (bottom of the IDE) allows users to run the program whereas the XEval console allows users to interact with meta-level functions, for instance functions to generate call graphs or to gather different joint points in a single view (Section 4.2).



```
run-print-balance
deposit
withdraw
balance
authenticate-user-db
user-exists?
get-password
authenticate-user-ws
get-username
authentication-db-aspect
```

```
(define deposit
  (lambda ()
    (begin
      (write-out "Start authentication"
      (if (not (authenticate-user-db
                 (get-username)
                 (get-password)))
       (error "Authentication Failure")
       false))
    (run-deposit 2000)))
```

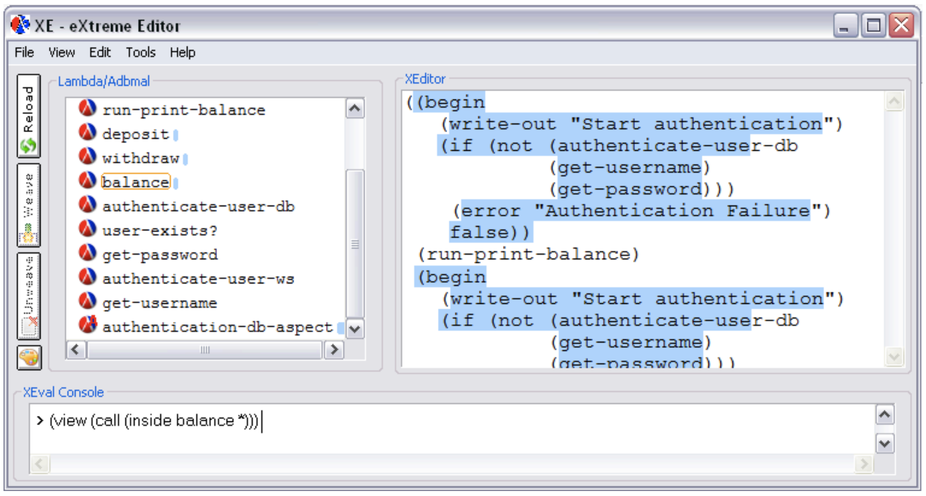
```
> (deposit)
```

Fig. 6. XE main window screenshot illustrates woven code and Eval console

4.1 Editing time code weaving

XE provides two mechanisms for interacting with the crosscutting structure of the system: 1) the multiple aspects selector (Figure 6), that permits developers to select a sub-set of aspects to automatically weave with the base code; and 2) edit-time code weaving (Figure 8), that automatically shows the aspect code (highlighted), in-line with the base code (normal font). Through the use of these two mechanisms, a programmer can directly interact with both aspects and base code implementations.

For example, if one renames a function argument in a woven view, that change is retained and incorporated into the base code. By the same token, changes to the woven aspect code can either be incorporated into the original aspect declaration or can be permanently incorporated to the base code itself. Figure 9, shows a user dialog that provides the programmer the option of incorporating the changes to the original aspect code or to the local function declaration only. Moreover, changes committed to the source aspect definition are automatically incorporated and updated in all views of program. In the case of incorporation of the aspect code to the base code, the PCD (the aspect's query definition) is automatically modified to exclude the



```
run-print-balance
deposit
withdraw
balance
authenticate-user-db
user-exists?
get-password
authenticate-user-ws
get-username
authentication-db-aspect
```

```
((begin
   (write-out "Start authentication")
   (if (not (authenticate-user-db
               (get-username)
               (get-password)))
     (error "Authentication Failure")
     false))
 (run-print-balance)
 (begin
   (write-out "Start authentication")
   (if (not (authenticate-user-db
               (get-username)
               (get-password)))
```

```
> (view (call (inside balance *)))
```

Fig. 7. XE main window screenshot illustrates gather join point view with woven code and XEval console

current function declaration. This action prevents an aspect behavior to be woven twice in the same function point.

By supporting the in-line and consistent modification of both base code and aspect code, XE also allows developers to avoid switching contexts, between base code and aspect code, in the middle of a task.

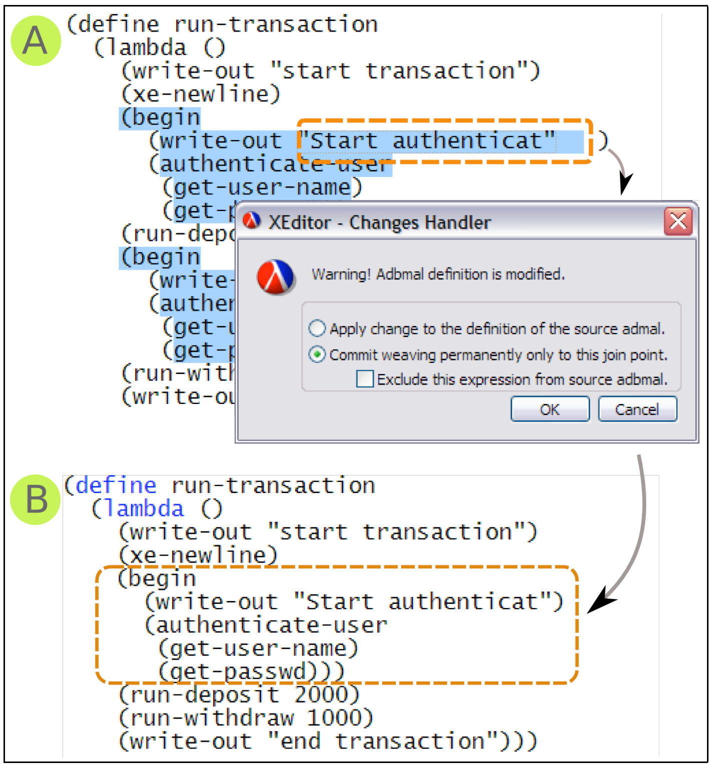
```
(define deposit
  (lambda ()
    (begin
      (write-out "Start authentication")
      (if (not (authenticate-user-db
                 (get-username)
                 (get-password)))
        (error "Authentication Failure")
       false))
    (run-deposit 2000)))
```

Fig. 8. Example of edit-time code weaving

4.2 Multiple views

In addition to the editing-time weaving of code, XE provides two additional views: 1) the gather joint point view, and 2) the call graph view (see below). Both are particularly important for large-scale projects, when the extension of program code may hinder its complete understanding by single developers. In these situations, it can be difficult for the developers to visually inspect the code and detect over- or under-matching conditions using the edit-mode view alone. These two additional views help to reduce information overload by allowing developers to skip through relevant code snippets or to inspect specific program execution slices. More details are presented in the next sections.

Gather join point model view A gather join point view allows developers to filter out different identified join points from the base code, displaying them, together, into a single editable active view. Using this view,



```
(define run-transaction
  (lambda ()
    (write-out "start transaction")
    (xe-newline)
    (begin
      (write-out "Start authenticat" )
      (authenticate-user
       (get-user-name)
       (get-p
    (run-depo
    (begin
      (write-
      (authen
       (get-u
       (get-p
    (run-wit
    (write-ou
```

Warning! Adbmal definition is modified.

( ) Apply change to the definition of the source admal.
(●) Commit weaving permanently only to this join point.
☐ Exclude this expression from source adbmal.

```
(define run-transaction
  (lambda ()
    (write-out "start transaction")
    (xe-newline)
    (begin
      (write-out "Start authenticat")
      (authenticate-user
       (get-user-name)
       (get-passwd)))
    (run-deposit 2000)
    (run-withdraw 1000)
    (write-out "end transaction")))
```

Fig. 9. Incorporation of aspect code to base code

a developer can interactively test a PCD descriptor expressiveness, detecting any under and over-matching situations, or selectively inspect different points in the code. Similar to the normal code view, the editing of the woven aspect code or the base code itself are both allowed, any changes being immediately reflected in other parts of the code.

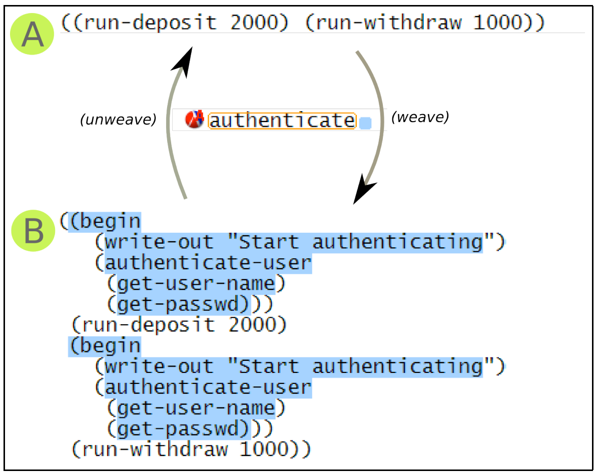
A developer produces different gather join point views by specifying queries to XE's meta-level interpreter using the XEval pane (see Section 5). For example, by querying `(view (call (run-transaction *)))`, a view is produced that shows any function invocations within `run-transaction`, as seen in Figure 10 (A). Gather join point view also includes woven code of function calls that are advised by aspects. As shown in Figure 10 (B), `run-deposit` and `run-withdraw` are advised by `authenticate-aspect`.

Call graph view The call graph view allows developers to visually inspect individual program execution traces, looking for inconsistent function calls or aspect weavings. A novel feature of this view is the ability to show the aspect code in-line with the base code, thus supporting developers in identifying under and over-matching conditions in a program execution trace.

This view is produced by specifying a starting point (function) and an ending point (another function) that constitutes a function call chain. For example, `(call-graph (inside main run-transaction) (inside run-deposit run-show-balance))` results the list of all intermediate functions calls between run-transaction and run-show-balance, as seen in Figure 11 (A).

5 XE Implementation

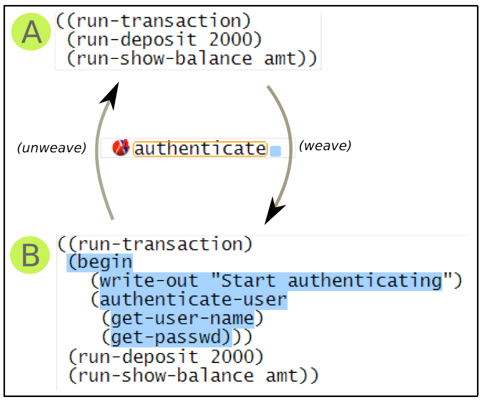
This section describes the main XE components and their role in the implementation of IDE.



```
((run-deposit 2000) (run-withdraw 1000))
```

```
((begin
   (write-out "Start authenticating")
   (authenticate-user
    (get-user-name)
    (get-passwd)))
 (run-deposit 2000)
 (begin
   (write-out "Start authenticating")
   (authenticate-user
    (get-user-name)
    (get-passwd)))
 (run-withdraw 1000))
```

Fig. 10. Gather joint-point view



```
((run-transaction)
 (run-deposit 2000)
 (run-show-balance amt))
```

```
((run-transaction)
 (begin
   (write-out "Start authenticating")
   (authenticate-user
    (get-user-name)
    (get-passwd)))
 (run-deposit 2000)
 (run-show-balance amt))
```

Fig. 11. Call graph view

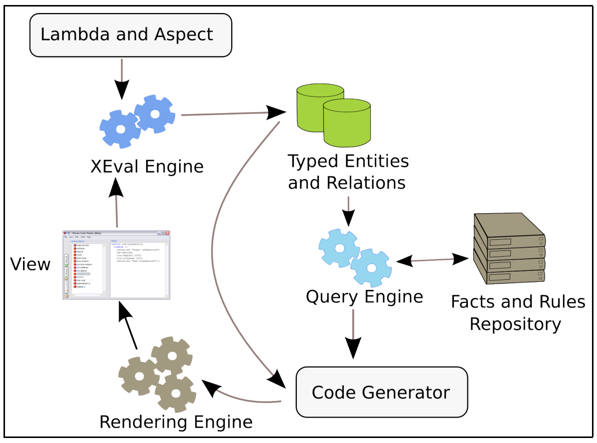


Fig. 12. XE architecture

In its core, the XE IDE integrates different components around a common relational model. The typed entities and relations (top of Figure 12), store representations of both aspects and base code. These elements are selected through the use of a query engine that combines these entities based on Facts and Rules Repository. These rules are automatically produced based on the PCD descriptors of each aspect. Once selected, aspects and base code are combined into the final program that is presented, according to different views, to the developers through PLT rendering engine. Current XE implementation supports the same set of regular Scheme language as in SICP book [1]. XE also uses the same interpreter, EVal, to execute the program. The EVal interpreter is incorporated to XE so that a developer can execute the program and inspect the behavior when aspect is woven or unwoven. In addition, XE provides meta-level interpreter, XEval. XEval is responsible for integrating aspect and base Scheme codes into a single view so that it is easy for a developer to understand and manage the program. It is central to views such as: gather join point and call graph, as previously discussed in Section 4.2. These components and their contribution to the system are described in, more detail, in the following sections.

5.1 Relational model and query support

The internal representation of programs (base code and aspect definitions) in XE is neither text- nor s-expression-based, but relational. Both aspect and base codes are stored as typed entities a relational database. A program is, therefore, a database that can be easily queried. Aspects are 'stored procedures' that define active queries. In the functional paradigm, these queries are implemented as meta-level editing functions that encapsulate the aspect advice and the point cut descriptors. Point cut descriptors are queries in the content of the program (function names, for example).

5.2 XEval engine

XEval engine is the component responsible for evaluating the plain text Scheme code. In addition to applying (or executing) the scheme code, XEval engine is also responsible for converting plain code text into an internal tabular (relational) representation of base code and aspects.

5.3 Typed entities and relations

All the source representations of the program that a programmer interacts with are views rendered in the XE user interface using typed entities and relations information. The views are generated on-demand; as a result of interaction with XE, or as requested by the developers. For example, clicking on one of the names in a list view of function definitions triggers a rendering of the Scheme representation of the code. To increase querying performance, query engine maintains the Facts and Rules Repository of the program synchronized with the typed entities and relation information so that aspect queries can be quickly resolved.

5.4 Rendering engine and code generator

Under the hood, a Rendering Engine that processes an annotated Scheme representation of the code from the Code Generator enables the editing-time visualization of aspects and base code. The Renderer Engine is also responsible to provide a rich drawing capability on a canvas that also acts as a structured code editor. It supports basic functionality like syntax highlighting and pretty printing. More importantly, it supports custom decoration of crosscutting fragments of code that are composed as a result of applying aspects on existing lambda definitions. Currently this decoration comes in the form of custom color highlighting of code fragments.

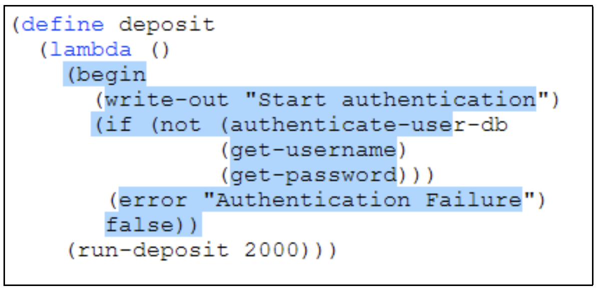
The code generator produces an intermediate representation of the scheme code to be rendered. This intermediate representation is fully annotated with various static-meta information about the many code fragments that constitutes a view. The key role of this meta-information is to support syntax highlighting, code coloring, in-line weaving of code. For example, using this information, the rendering engine gathers and combines code fragments from the: Typed Entities and Relations component and the query results from Query Engine.

6 Supporting AOP with XE

This section shows how XE supports developers in the evolution and debugging of aspect oriented programs in Scheme. For such, we use the same application discussed in Section 3 and follow the same evolutionary steps.

6.1 Evolving the bank accounting API

The user starts with a woven view in the main XE editor, where the database aspect authentication (Figure 4) and the base code of Figure 3 appear woven, in the same view, as shown in Figure 13.



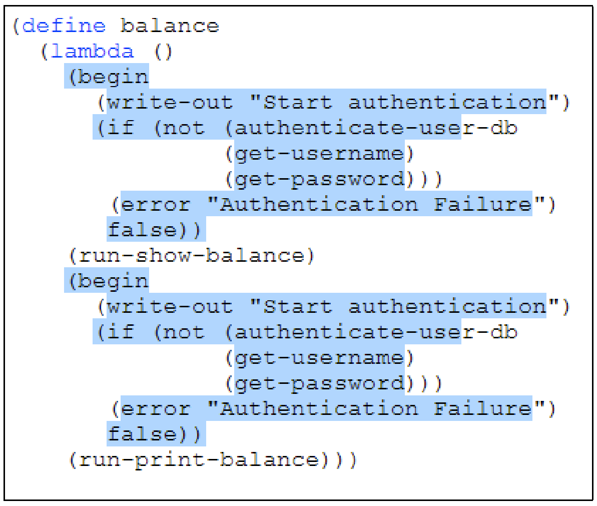
```
(define deposit
  (lambda ()
    (begin
      (write-out "Start authentication")
      (if (not (authenticate-user-db
                 (get-username)
                 (get-password)))
       (error "Authentication Failure")
       false))
    (run-deposit 2000)))
```

Fig. 13. Visualizing and identifying the points of woven code

After implementing the new `run-print-balance` function and extending the balance definition to invoke this function; whenever a balance is requested, the authentication aspect is automatically woven to that function call as shown in Figure 14.

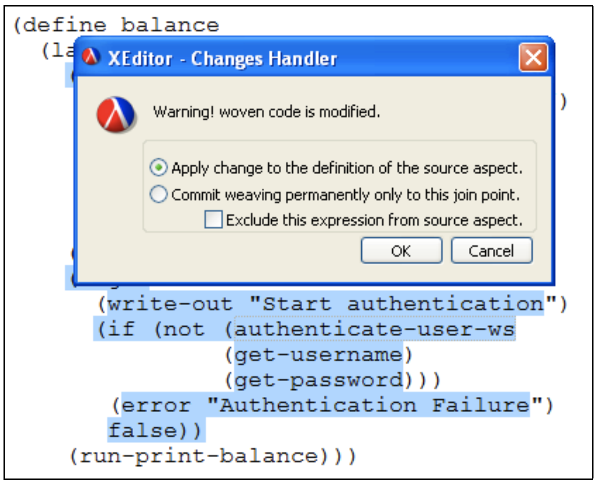
As the over-matching condition becomes obvious, the developer fixes this problem by changing the inlined aspect code from `authenticate-user-db` to `authenticate-user-ws`. After modifying the in-line code from the source aspect, the developer is given two options as shown in Figure 15, and discussed in the next sub-sections.

Apply the changes to any join point that match this PDC of the aspect Should the developer choose this option, the authenticate-db-aspect is automatically revised to exclude any run-print-balance calls from being advised by this aspect (see in Figure 16). A new aspect, authenticate-ws-aspect is then created to capture and advise run-print-balance join points (see Figure 17).



```
(define balance
  (lambda ()
    (begin
      (write-out "Start authentication")
      (if (not (authenticate-user-db
                 (get-username)
                 (get-password)))
        (error "Authentication Failure")
       false))
    (run-show-balance)
    (begin
      (write-out "Start authentication")
      (if (not (authenticate-user-db
                 (get-username)
                 (get-password)))
        (error "Authentication Failure")
       false))
    (run-print-balance)))
```

Fig. 14. Visually detect the over-matching condition



```
(define balance
  (la
```

XEditor - Changes Handler

Warning! woven code is modified.

- [✓ Apply change to the definition of the source aspect.]
- Commit weaving permanently only to this join point.
- Exclude this expression from source aspect.

```
      (write-out "Start authentication")
      (if (not (authenticate-user-ws
                 (get-username)
                 (get-password)))
        (error "Authentication Failure")
       false))
    (run-print-balance)))
```

Fig. 15. Dialog providing different change strategies

Permanently incorporating aspect code to this join point If the developer chooses to incorporate the existing aspect code to the base code, XE automatically revises the aspect PDC to exclude the incorporated join point, thus preventing the re-weaving of the aspect to the new join point. In this case, the revised aspect, produced by the XE IDE would become the one shown in Figure 18.

7 Evaluation

In this section, we evaluate the usefulness of XE by quantitatively comparing the efforts required to perform our motivation scenario tasks with and without XE support. As such, we compute different metrics as shown in Table 1, collected during the process of evolving the banking API described in this paper. These metrics were chosen to elucidate common problems such as over- and under-matching as well as context switching, which have been shown to increase the developer's cognitive load [14].

These metrics represent:

- the number of major steps that the developer needs to perform during the course of evolution in order to detect the different programming errors (which includes debugging cycles);

```
(define authentication-db-aspect
  (aspect ()
         (before
         (call (and (or (inside deposit *)
                        (inside withdraw *)
                        (inside balance *))
                     (not (inside *
                              run-print-balance))))
         (begin
          (write-out "Start authentication")
          (if (not (authenticate-user-db
                    (get-username)
                    (get-password)))
           (error "Authentication Failure")
           false)))))
```

Fig. 16. Excluding run-print-balance from PDC

```
(define authentication-ws-aspect
  (aspect ()
         (before
         (call (inside * run-print-balance))
         (begin
          (write-out "Start authentication")
          (if (not (authenticate-user-ws
                    (get-username)
                    (get-password)))
           (error "Authentication Failure")
           false)))))
```

Fig. 17. Creating new aspect to capture run-print-balance procedure call join points

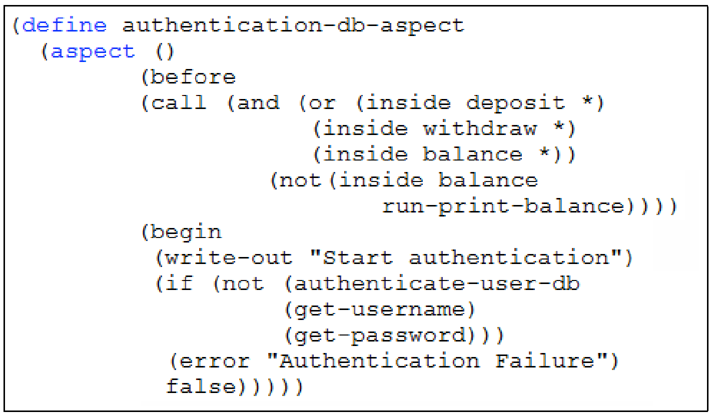
- the number of context switches between aspect and base code files, which potentially increases the cognitive load of developers;
- the number of times an over-matching condition was present during the course of the task without any feedback provided to the developer by the IDE;
- And the points in the program where the code became inconsistent (or semantically wrong) without any notice to the developers. This number includes over and under-matching conditions.

This simple task analysis shows how the visualization and consistency mechanisms build on XE can better support developers in evolving and understanding AOP code by reducing the number of context switches and by making explicit under and over-matching conditions, preventing common AOP programming mistakes.

8 Related Work

In the literature, different strategies have been proposed to address the deficiencies of the AOP programming model. This section discusses some of these approaches, comparing its benefits and shortcomings with those provided by XE.

Programming language constructs. Some examples of programming language constructs developed to address the AOP obliviousness deficiencies include: *Crosscut Programming Interfaces* (or XPIs) [9], Open Modules [2][18], and the use of source code annotations as employed by JBoss AOP and Spring Framework AOP. These approaches break the original base code-aspect obliviousness by explicitly defining points in the base code where aspects should be woven. In spite of these advances, developers still need to manage



```
(define authentication-db-aspect
  (aspect ()
         (before
         (call  (and (or (inside deposit *)
                         (inside withdraw *)
                         (inside balance *))
                      (not (inside balance
                               run-print-balance))))
         (begin
          (write-out "Start authentication")
          (if (not (authenticate-user-db
                   (get-username)
                   (get-password)))
           (error "Authentication Failure")
           false)))))
```

Fig. 18. Revised aspect after incorporating of aspect code to base code

Table 1. Summary of usability metrics collected with and without IDE support

Item	Without Tool	With Tool
# Steps	4	2
# Context switches	3	0
# over-matching	1	0
# under-matching	1	0
# Points where code becomes semantically wrong, without any feedback to the developer	3	0

configuration files (as in Spring and JBoss) or adopt general code conventions and design rules as in XPIs and Open Modules. Moreover, these approaches, while representing a significant step in addressing aspect-base code inconsistencies, do not support the semantic co-evolution of aspects and base code as shown in our examples.

IDE-level approaches. These approaches address the AOP usability issues by providing tool support around existing compilers, tools and languages. In this category, different systems, including our own, have been developed as follows.

The Eclipse AJDT [2] supports the visualization of places in the base code advised by AspectJ joint points through the use of decorators along the code. It does not provide in-line visualization, requiring developers to constantly switch context between base and aspect code in order to: understand the semantics of the aspect code in order to perform semantic changes in the aspects or base code.

AspectBrowser [8], Aspect Visualizer [3] and ActiveAspect [4] are tools that permit the visualization of locations in the code where aspects are woven. These tools, however, are based on birds-eye visualizations of these relations, and lack the in-line visualization capability of XE, together with the ability to maintain the consistency of base and aspect code.

SourceWeave.NET [12] is a project that supports source level weaving of classes written in various .NET languages (C#, VB.NET and J#). It uses a XML descriptor file to specify the interaction between the aspects and representative components. The technique uses a mapping to identify join point shadows (areas in the source where join points may emerge) and a pointcut-to-joinpoint binding to isolate parts of the source. A unique feature of SourceWeave.NET is the support for cross-language weaving of aspect definition and base

[2] http://www.eclipse.org/ajdt/

code. Even though SourceWeave.NET supports weaving at source code level, it does not target edit-time code weaving as supported by XE.

There are few other tools that support source level weaving of aspects. Aspect.NET, for example, is an AOP framework for Microsoft .NET. Another example is AspectC++ that provides a static aspect-oriented system for C/C++, supporting the same kinds of pointcut designators as AspectJ, and providing source-to-source translation from its AOP language to regular C/C++. None of these tools, however, allow the viewing of the target application code after aspect weaving. They also do not provide any functionality that allows developers to gauge the changes in the code when aspect definition has been changed.

In-line (or fluid) code weaving is a novel technique toward providing better support for AOP developers. It provides in-line representation of aspect and base code in a single view, minimizing context switching and helping in the identification of over and under-matching situations.

Fluid AOP [11] and Fluid Source Code View [5] are IDEs that provide the ability to temporarily shift a program (or other software model) to a different structure (or view), allowing developers to operate over that view. The idea of shifting back and forth between different structural representations of a program is deeply ingrained in XE. Both fluid AOP and XE support weaving (applying) at the level of source code share the common idea that a single change has multiple effects to the base code. However, Fluid AOP lacks the selectivity of views supported by XE. In Fluid AOP, developers cannot weave and unweave aspects arbitrarily, they cannot change the aspect code in place, nor debug their application through additional views such as the call graph and gather-joint-point.

Aspect-JEdit [19] is a tool that supports editing of in-line aspects. It supports depicting multiple views of the program, revealing aspects in the editing context. However, it does not support the code querying capability of XE. Furthermore, to see the effect of woven code in Aspect-JEdit, developers must manually mark the code in different colors in all the places where they are in-lined. With XE, simply assigning a color to an aspect associates all the woven code (due to that aspect) with that particular color.

Relational code representation. The relational presentation of programs used in XE is not a new idea. For example, systems as CodeQuest [10] and the work of Lam et al. [17] use a relational model to store information of program in a relational database system. XE leverages this characteristic to support multiple views and aspect-base code consistency.

XE also shares some similarities with the Decal tool by Janzen and Volder [13]. Like XE, Decal uses a relational database as a common representation of the program so that views can be generated on-demand. Decal and XE have a similarity in a way that they produce effective textual views. In summary, the major difference of XE over most of the above implementations is that they ignore editing and interaction capabilities of woven code and do not support more advanced features such as the call graph and gather-joint-point, which are particularly important in large-scale code bases.

9 Conclusion

AOP has gained popularity and been implemented in a wide variety of applications, for example, J2EE web applications and large-scale enterprise applications [6][22]. While the Aspect-Oriented paradigm has supported the modularization of crosscutting concerns, separating base and aspect code in disjoint entities, this novel programming paradigm comes with extra usability costs. In particular, the excessive context switching between base and aspect code, and the lack of support for developers in detecting over- and under-matching conditions can result in different programming errors.

In this paper, we presented XE, a programming environment that addresses these issues at the IDE level. The contributions of this paper include: A discussion of the main usability problems found in AOP languages; A scheme implementation for AOP allowing the construction of AOP programs in the functional model, and XE, an IDE that addresses fundamental usability issues existing in current AOP languages. In our experiments, the use of in-line code editing and the support for automatic PCD adjustment have shown to prevent many of the common errors induced by the AOP programming model. This paper shows the benefits of XE both through a motivating example and through a case study that shows how XE can reduce context switching, while supporting developers in the detection of under and over-matching conditions. Future work also includes extending XE program model and approach to other programming paradigms, i.e. Java and AspectJ, and support for parallel development, when different programmers participate in the development of software.

10 Acknowledgements

This research was supported by the U.S. National Science Foundation under grant numbers 0534775, 0205724 and 0326105, an IBM Eclipse Technology Exchange Grant, by the Intel Corporation, and the grant by the Royal Thai Army.

References

1. Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 2nd edition, 1996.
2. J. Aldrich. Open modules: A proposal for modular reasoning in aspect-oriented programming. In *Foundations of Aspect Languages*, 2004.
3. A. Clement, A. Colyer, and M. Kersten. Aspect-oriented programming with ajdt. In *ECOOP Workshop on Analysis of Aspect-Oriented Software*, July 2003.
4. Wesley Coelho and Gail C. Murphy. Activeaspect: presenting crosscutting structure. pages 1–4, St. Louis, Missouri, 2005.
5. Michael Desmond, Margaret-Anne Storey, and Chris Exton. Fluid source code views. pages 260–263, Washington, DC, USA, 2006. IEEE Computer Society.
6. A. Duck. Implementation of aop in non-academic projects. Bonn, Germany, 2006. ACM Press.
7. T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44:29–33, 2001.
8. W. G. Griswold, Y. Kato, and J. J. Yuan. Aspectbrowser: Tool support for managing dispersed aspects, 2000.
9. W. G. Griswold, M. Shonle, K. Sullivan, Y. Song, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23:51–60, 2006.
10. Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.
11. Terry Hon and Gregor Kiczales. Fluid aop join point models. pages 712–713, New York, NY, USA, 2006. ACM Press.
12. A. Jackson and S. Clarke. Sourceweave.net: Cross-language aspect-oriented programming. *Lecture Notes in Computer Science*, 3286/2004:115–135, 2004.
13. D. Janzen and K. de Volder. Programming with crosscutting effective views. pages 197–222. Springer-Verlag, 2004.
14. Mik Kersten and Gail Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14*, pages 1–11. ACM Press, 2006.
15. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Communications of the ACM*, 44:59–65, 2006.
16. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, Jyväskylä, Finland, 1997.
17. M. Lam, J. Whaley, V. Livshits, M. Martin, D. Avots, and M. Context-sensitive program analysis as database queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–12, 2005.
18. N. Ongkingco, P. Avgustinov, L. Hendren, O. de Moor, G. Sittampalam, and J. Tibble. Adding open modules to aspectj. In *International Conference on Aspect-Oriented Software Development*, 2006.

19. T. Panas, J. Karlsson, and M. Hgberg. Aspect-jedit for inline aspect support. March 2003.
20. Friedrich Steimann. The paradoxical success of aspect-oriented programming. SIGPLAN, pages 481–497, NY, October 2006. ACM.
21. Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. pages 166–175, Lisbon, Portugal, 2005. ACM.
22. Daniel Wiese, Regine Meunier, and Uwe Hohenstein. How to convince industry of aop. In *Sixth International Conference on Aspect-Oriented Software Development*, Canada, 2007.