XE (eXtreme Editor) - Bridging the Aspect-Oriented Programming Usability Gap

Wiwat Ruengmee, Roberto Silveira Silva Filho,
Sushil Krishna Bajracharya, David F. Redmiles, Cristina Videira Lopes

Department of Informatics, Donald Bren School of Information and Computer Sciences, University of California
Irvine, CA, 92697 USA
{wruengme, rsilvafi, sbajrach, redmiles, lopes}@ics.uci.edu

***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. This paper describes XE (Extreme Editor), an IDE that supports developers in managing these issues in the functional aspect-oriented programming domain.**

I. 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 [1]. Aspect abstractions comprise two basic components: the aspect behavior, or advice, and a pointcut 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 [2].

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 [3] has been shown to be flawed [4]. This separation actually adds considerable complexity to aspect-oriented development. Popular general purpose AOP languages such as AspectJ 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 pointcut 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.

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.

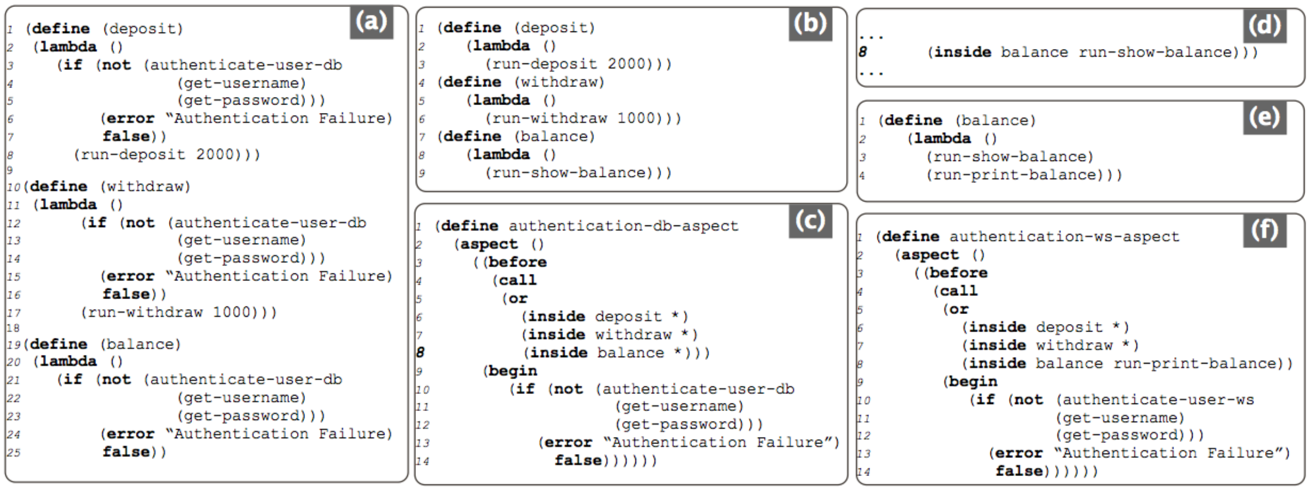
II. Motivating Scenario

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.

A. 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 1 (a).



(a)

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

(define (withdraw)
 (lambda ()
      (if (not (authenticate-user-db
                (get-username)
                (get-password)))
        (error "Authentication Failure)
        false))
      (run-withdraw 1000)))

(define (balance)
 (lambda ()
   (if (not (authenticate-user-db
                (get-username)
                (get-password)))
        (error "Authentication Failure)
        false))
```

(b)

```
(define (deposit)
   (lambda ()
     (run-deposit 2000)))
(define (withdraw)
   (lambda ()
     (run-withdraw 1000)))
(define (balance)
   (lambda ()
     (run-show-balance)))
```

(c)

```
(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))))))
```

(d)

```
...
8       (inside balance run-show-balance)))
...
```

(e)

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

(f)

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

Fig. 1. Steps of evolving a simple banking transaction API

This code was modularized, within an authentication aspect, by factoring out the permission-checking code from the API methods, as shown in Figure 1 (b), and by creating an authentication aspect in XE Scheme as shown in Figure 1 (c). Both base code and aspect code reside in separate files.

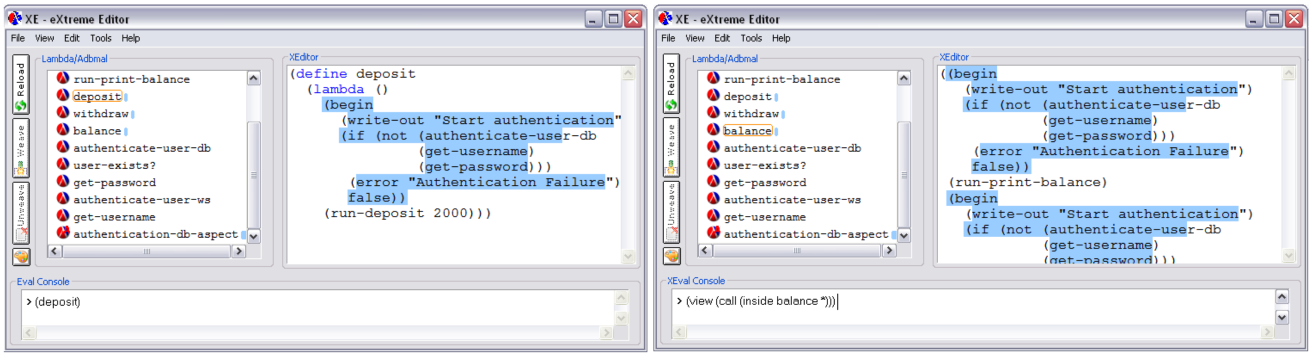
After separating a concern into base code and aspect code, the developers need to reintegrate (or weave) the aspect and base code behaviors by 1) loading both base code and aspect code files into the XE environment, and 2) manually weaving the aspect code with the base code through a menu command in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

B. 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:

- 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.
- 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 1 (e)). 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-user-db` code. We call this situation 'over-matching'.
- After a weaving/execution/testing cycle, where the wrong behavior becomes evident, a developer corrects this problem by modifying the wild card in `(inside balance *)` to a more specific, `(inside balance run-show-balance)`, as seen in Figure 1 (d). 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.
- After another weaving/execution/testing cycle 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` (as seen in Figure 1 (f)), to fix this problem. Let us 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 `balance` and `withdraw`. Moreover, a feature interference condition would also occur with the existing `authenticate-ws-aspect`.
- 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, where the program is syntactically correct but semantically incorrect. 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.



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

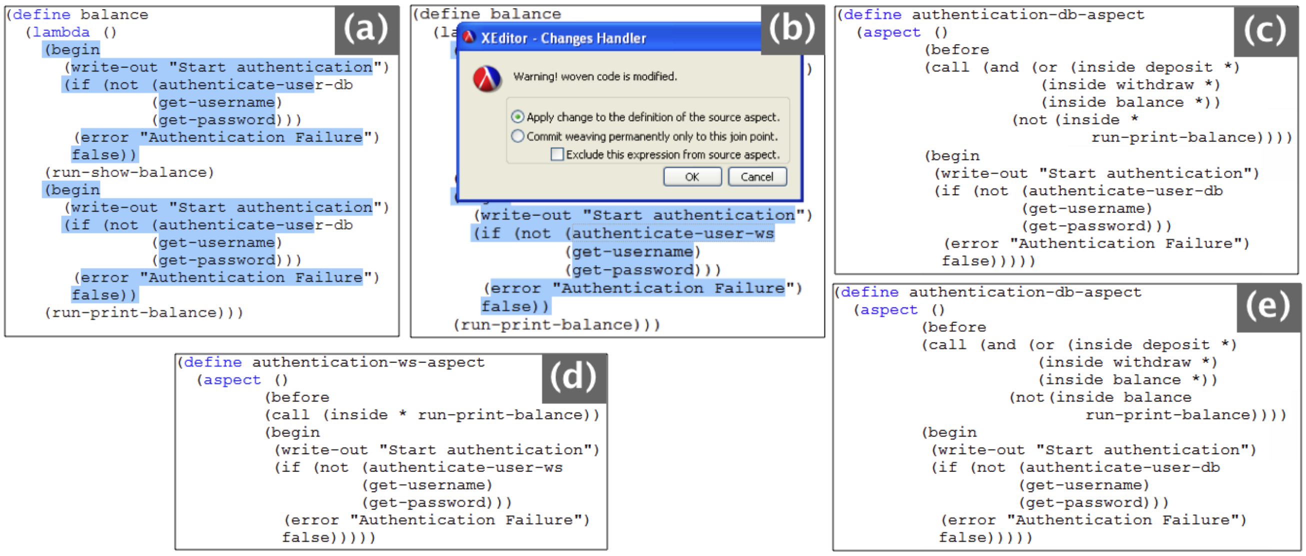
```
(define deposit
  (lambda ()
    (begin
      (write-out "Start authentication"
      (if (not (authenticate-user-db
                  (get-username)
                  (get-password)))
       (error "Authentication Failure")
       false))
    (run-deposit 2000)))
```

```
> (deposit)
```

```
((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. 2. XE main window screenshots. Woven code and Eval console (left), and gather join point view with woven code and XEval console (right)



(a)

```
(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)))
```

(b)

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

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.

(c)

```
(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)))))
```

(d)

```
(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)))))
```

(e)

```
(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. 3. Evolving a simple banking transaction API with XE IDE support

III. Supporting AOP with XE IDE

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 2). XE was implemented in PLT Scheme. For more information about XE implementation, please refer to the Technical Report [5].

With XE IDE support, this section shows how XE IDE facilitates developers in the evolution and debugging of aspect oriented programs in Scheme. As such, we use the same application discussed in Section II and follow the same evolutionary steps.

A. Evolving the Bank Accounting API

The user starts with a woven view in the main XE editor, where the database aspect authentication (Figure 1 (a)) and the base code of Figure 1 (b) appear woven, in the same view, as partially shown in Figure 3 (a).

After implementing the new `run-print-balance` function (code not shown here) 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 3 (a).

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 3 (b), and discussed in the next sub-sections.

1) 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 3 (c)). A new aspect, `authenticate-ws-aspect` is then created to capture and advise `run-print-balance` join points (see Figure 3 (d)).

2) 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

TABLE I
SUMMARY OF USABILITY METRICS COLLECTED WITH AND WITHOUT IDE SUPPORT

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

revised aspect, produced by the XE IDE would become the one shown in Figure 3 (e).

IV. 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 I, 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 developers cognitive load [6].

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);
- 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.

V. RELATED WORK

Programming language constructs such as *Crosscut Programming Interfaces* (or XPIs) [7] and Open Modules [8][9] break the original base code-aspect obliviousness by explicitly defining points in the base code where aspects should be woven. 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.

In-line (or fluid) code weaving is a novel technique toward providing better support for AOP developers. It provides an integrated 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 [10] and Fluid Source Code View [11] 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. Both fluid AOP and XE support editing-time code weaving, and the idea that a single aspect change may impact many places in the base code. However, Fluid AOP lacks the selectivity of views supported by XE.

VI. CONCLUSION AND FUTURE WORK

The main 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. This paper illustrates the benefits of XE both through a motivating example and through a case study that illustrates how XE can reduce context switching, while supporting developers in the detection of under and over-matching conditions. Future work includes extending the XE program model and approach to other programming paradigms, e.g. Java and AspectJ, and support for parallel development, where different programmers participate in the development of software.

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, the Intel Corporation, and by the Royal Thai Army.

REFERENCES

[1] 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*, Jyvskyl, Finland, 1997.
[2] F. Steimann, "The paradoxical success of aspect-oriented programming," ser. SIGPLAN. NY: ACM, Oct. 2006, pp. "481–497".
[3] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented programming: Introduction," *Communications of the ACM*, vol. 44, pp. 29–33, 2001.
[4] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, "Information hiding interfaces for aspect-oriented design." Lisbon, Portugal: ACM, 2005, pp. 166–175.
[5] W. Ruengmee, R. S. Filho, S. Bajracharya, D. Redmiles, and C. Lopes, "Xe (extreme editor) - tool support for evolution in aspect-oriented programming," *Technical Report UCI-ISR-08-1, University of California, Irvine*, June 2008.
[6] M. Kersten and G. Murphy, "Using task context to improve programmer productivity," in *SIGSOFT '06/FSE-14*. ACM Press, 2006, pp. 1–11.
[7] W. G. Griswold, M. Shonle, K. Sullivan, Y. Song, N. Tewari, Y. Cai, and H. Rajan, "Modular software design with crosscutting interfaces," *IEEE Software*, vol. 23, pp. 51–60, 2006.
[8] J. Aldrich, "Open modules: A proposal for modular reasoning in aspect-oriented programming," in *Foundations of Aspect Languages*, 2004.
[9] 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.
[10] T. Hon and G. Kiczales, "Fluid aop join point models." New York, NY, USA: ACM Press, 2006, p. 712713.
[11] M. Desmond, M.-A. Storey, and C. Exton, "Fluid source code views." Washington, DC, USA: IEEE Computer Society, 2006, p. 260263.