Sophisticated crosscuts for e-commerce*

Rémi Douence, Olivier Motelet, Mario Südholt

École des Mines de Nantes
4 rue Alfred Kastler, 44307 Nantes cedex 3, France
Corresponding author: `motelet@emn.fr`

1 Introduction

AOP [2] introduced the notion of crosscutting concerns in programming. An aspect groups a *crosscut* (aka. *pointcut* in Aspect-J [3]) – which relates several *points of interest* (aka. *join points*) of the base application – with an *action* (aka. *advice*) to be performed. This article argues for a clear separation of *crosscut* and *action* definitions. The tools for AOP currently available support only simple crosscut definitions, which hinders the separation of crosscuts and actions. In this paper we advocate that more sophisticated crosscuts can solve this problem.

This article discusses two aspects in the context of a simple e-commerce application. First, we present this AOP example using simple crosscut definitions only. Then we give an alternative version where sophisticated crosscut definitions enable us to clearly separated crosscuts and actions. Finally, we briefly discuss benefits of elaborate crosscuts definitions in AOP.

2 A simple discount and security aspect for e-commerce

2.1 A basic e-commerce application

In this article, we consider the case of a web-based e-commerce application as a running example. A customer of such an application navigates inside the shop's web site in search for interesting products: he goes forward from web pages to other web pages while displaying items. This navigation is represented in the following by the command `search`. During his search, he may also go back to some products that were displayed on previous pages by performing the `back` command. The customer can also purchase the displayed products by issuing the command `buy`. This last command does not generate a new web page and does not therefore act on the navigation state. The application allows any combination of these commands. For the sake of simplicity, we suppose the initial page already displays a selection of best-seller products to be bought and `back` has no effect on the initial page. In this setting, a concrete usage scenario could read as follows:

```
search; buy; search; back; search; buy
```

*This work has been partially funded by the EU project "EasyComp" (`www.easycomp.org`), no. IST-1999014191.

This trace means that the customer made a first search for a specific product and ordered it, then he performed a second search that he canceled, and he finally purchased the result of a third search.

The previous scenario illustrates the base functionality of the e-commerce application. We are interesting in introducing additional behavior like a discount policy for rewarding repeated purchases and a security policy based on authentication before payments. Such transverse features can be implemented as aspects in the sense of AOP.

2.2 AOP from a monitoring perspective

To make the presentation more precise, we present the examples in a framework defining AOP from a monitoring perspective [1]. In this framework, execution monitors serve as an operational model for AOP. The execution of the base program generates events so that monitors can survey the execution. Such events could model both the control flow (e.g. method/function calls) and the data flow (e.g. through assignments) of the base program. A crosscut can then be defined as a pattern of events, which are tested for each time an event is emitted. An aspect can thus be defined by grouping in a rule a pattern of events with an action to be triggered:

```
aspect = when aPatternOfEvents perform aFunctionCall
```

This definition can be read as: each time the base application performs the execution sequences described in `aPatternOfEvents`, pause the base application execution, call the function `aFunctionCall` with eventually some information about matched events, and resume the base application execution. We presented a prototype for JAVA implementing this framework and a DSL for formal crosscut definitions in [1].

2.3 Informal definition of the discount and security aspects

In the monitor-based framework, the discount policy can be expressed as:

```
discount = when buy a product
           perform apply discount if not first payment
```

In case of the trace given in Section 2.1, a discount will be applied to the second purchased product. Similarly, the security policy could be defined as:

```
security = when buy a product perform authenticate the user if not already not done
```

In this case only the first payment of the user requires his authentication. Obviously, in general the `back` action can interfere with the security aspect: once the user is back to a page loaded before the last authentication process, he will have to be authenticated again when the next purchase occurs. For instance, when performing the following sequence of commands `search; buy; back; search; buy`, the user must be authenticated twice.

3 Defining the aspects using simple crosscuts

Figure 1 defines a discount aspect in a JAVA-like syntax. The aspect defines two variables. The variable `firstBuy` which is initialized to `true` is used to identify the first occurrence of the command `buy`. The variable `discountRate` keeps track of the incremental discount to be applied. When a command `buy` is detected, *either* it is the first occurrence of this event and no discount is applied but the boolean tag is changed, *or* the price is reduced using `discountRate`.

```
aspect Discount {
  boolean firstBuy = true;
  float discountRate = 1.0;

  when buy perform {
    if (firstBuy)
      firstBuy = false;
    else {
      discountRate -= 0.01;
      price *= discountRate;
    }
}
```

Figure 1: A discount aspect (version 1)

```
aspect Security {
  boolean authenticated = false;
  int afterAuthenticate;

  when search perform {
    if (authenticated)
      afterAuthenticate++;
  }
  when back perform {
    if (authenticated) {
      if (afterAuthenticate == 0)
        authenticated = false;
      else
        afterAuthenticate--;
    }
  }
  when buy perform {
    if (!authenticated) {
      authenticate();
      authenticated = true;
      afterAuthenticate = 0;
    }
  }
}
```

Figure 2: A security aspect (version 1)

Similarly, Figure 2 defines a security aspect. The boolean variable `authenticated` specifies whether the next command `buy` requires an authentication. The integer variable `afterAuthenticate` counts the number of `back` commands allowed such that the user remains authenticated.

In these two aspects definitions, we can distinguish two pieces of code in the actions introduced by the keyword ***perform***:

Book-keeping code. This code deals with boolean tags and integer counters. They are used to express sophisticated crosscutting conditions such as "unless it is the first payment" or "if it (i.e. authentication) was not done before" (see Section 2.3).

Action code. This code performs changes on the base level, i.e., dealing with variables of the base application (e.g. `price`) and "real" actions (e.g. `authenticate()`).

These aspect definitions are unsatisfactory because they do not provide a clear separation of crosscut and action specifications.

4 Defining the aspects based on sophisticated crosscuts

Figure 3 presents another definition of the discount aspect in which the crosscut is defined using a event patterns which represent *sequences of execution points* instead of a single event as in the previous section. The pattern matching is implemented by the function `enableDiscount()`. This function skips the first occurrence of the `buy` event and calls the function `enableDiscount2()`. This second function returns a crosscut when the next `buy` event occurs. At the same time, it also continue to detect the following occurrences of `buy` (i.e. the next crosscuts). These two concurrent tasks (i.e. returning the detected crosscut *and* continuing to detect further crosscuts) are implemented with the help of the parallel operator `|||` and a recursive call. (These constructions are formally defined in [1].) Note that this new version of the discount aspect does not need a boolean tag in order to distinguish the first occurrence of `buy` from the following ones. Note also that the action introduced by ***perform*** is not polluted with book-keeping code as is the case in the previous definition.

Figure 4 redefines the security aspect. In this case, the pattern matching functions are `authRequired()` and `authRequired2()`. The function `authRequired2()` has an integer counter as parameter. Note that in this version the action introduced by ***perform*** is also not polluted with book-keeping code simply calls `authenticate()`.

These examples demonstrate that an expressive language for crosscut definitions is useful to obtain a clear separation between crosscut and action definitions. This separation makes the aspect specifications easier to understand: the programmer can read crosscuts and actions separately. The aspect specifications are also more reusable since the programmer can modify crosscuts and actions separately.

5 Discussion

In this article, we presented two different definitions of the same aspects. One, discussed in Section 3, relies on simple crosscuts definitions by restricting a crosscut to a single point of interest and polluting action specifications with book-keeping code. The other definition, exemplified in Section 4, relies on richer crosscuts definitions and specifies a crosscut as a

```
aspect Discount {
  float discountRate = 1.0;

  when enableDiscount() perform {
    discountRate -= 0.01;
    price *= discountRate;
  }

  Crosscut enableDiscount() {
    Event e = nextEvent(buy);
    return enableDiscount2();
  }
  Crosscut enableDiscount2() {
    Event e = nextEvent(buy);
    { return new Crosscut(e);
      |||
      return enableDiscount2();
    }
  }
}
```

Figure 3: A discount aspect (version 2)

```
aspect Security {

  when authRequired() perform {
    authenticate();
  }

  Crosscut authRequired() {
    Event e = nextEvent(buy);
    { return new Crosscut(e);
      |||
      return authRequired2(0);
    }
  }
  Crosscut authRequired2(int n) {
    Event e = nextEvent();
    switch (e) {
      case search: return authRequired2(n+1);
      case back  : if (n == 0) return authRequired();
                   else        return authRequired2(n-1);
    }
  }
}
```

Figure 4: A security aspect (version 2)

pattern of events denoting sequences of execution points to be matched. Sophisticated crosscut definition provides a clean separation of crosscuts and actions. We argue that this separation supports the study of aspect interactions.

Study of aspect interactions using sophisticated crosscuts. Aspect interaction is an important issue of AOP. In our e-commerce scenario, both aspects may interact. For instance, in the usage scenario `search; buy; back; search; buy`, both discount and security aspects crosscut at the second `buy`. In this case, the two corresponding actions must be carefully ordered. Indeed, a failed authentication should cancel the discount procedure: the security action must be executed before the discount action. On the other hand, simple and realistic restrictions on the application scenario could prevent their interaction: for instance, if each `buy` flushes the web page cache in order to forbid too large a number of `back` commands, both aspects can never interact and their order is no more important.

Aspect interaction issues can often be handled as crosscut interaction issues and can be studied by formally analyzing crosscuts. A clear separation of crosscut and action then allows to focus the analysis on the sole crosscuts definitions.

Perspectives. The tools for AOP currently available do not provide satisfying support for sophisticated crosscut definitions. There are some interesting first steps (like AspectJ's `cflow()` primitive which avoids polluting the action code with a stack-like crosscutting behavior) but they are limited to predefined primitives.

In [1], we introduced a crosscut language expressive enough to define sophisticated crosscuts and allowing a clear separation of crosscut and action. Its semantics could serve as a formal base for crosscuts definition analysis as we detailed in proving certain crosscut equivalences — a special case of interaction. This topic remains to be studied further to achieve a general method for the analysis of aspect interaction.

References

[1] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. Technical Report 01/3/INFO, École des Mines de Nantes, 2001.

[2] G. Kiczales et al. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th Europeen Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[3] G. Kiczales et al. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2001. To appear, preprint version: see AspectJ home page, `www.aspectj.org`.