Aspects as Components

Marcelo Medeiros Eler and Paulo Cesar Masiero

Dept. of Computer Science, ICMC - University of Sao Paulo,
13560-970 Sao Carlos - SP - BR. P.O.Box 668
{mareler, masiero}@icmc.usp.br

Abstract. An adaptation of the UML Component method to design crosscutting components is briefly presented. Such components are allowed to crosscut only the public interface of base (convencional) components. The design and implementation of crosscutting components using the language JAsCO is discussed.

1 Introduction

Components are units of composition with contractually specified interfaces and explicit context dependencies only [4]. Component-based software development (CBSD) aims at decomposing software into independent modules that are easy to manage, reuse and evolve. Several methods to support CBSD have been proposed [1, 4, 6], which usually provide guidelines on how to encapsulate concerns into components. However, there are crosscutting concerns such as logging, tracing and persistence, that cannot be implemented as components using tools like CORBA, EJB and COM. By their nature, they crosscut the component structure within components and across the components' boundaries [3].

A way to solve this problem is to use containers but the calls to the services provided by them are still spread along several components. Another way is to combine CBSD with aspect-oriented software development (AOSD) to support the implementation of crosscutting concerns as independent modules. The problem with this combination is that, by their nature, aspects may crosscut the internal component structure thus clashing with the component opaqueness. A solution to this is to compromise aspect expressiveness allowing aspects to operate only on the public operations exposed in the components' interfaces and forbidding them of extending any operation through inter-type declarations [3].

We show briefly an adaptation of the UML Components method [1] to produce a component-based design that also includes aspectual (crosscutting) components, preserving the component opaqueness by allowing only crosscutting of public operations in the interface of base components. A brief introduction to the component architecture that is produced and to how crosscutting components can be designed to be reused is presented.

2 Component/Aspect Architecture

We have devised a method to develop software with components and aspects. Using this method, we produce a component/aspect architecture as the one shown in Figure 1 (part A) for a Hotel Reservation System (HRS) [1]. In the architecture, we have crosscuting (LoggingOpMgr) and base componentes (ReservationSystem and CustomerMgr, for example). The crosscutting component provides a crosscutting interface and requires services from business components (UserMgr and LoggingMgr). The diamond and the interface name 'ICC' prefix both indicate a crosscutting interface.

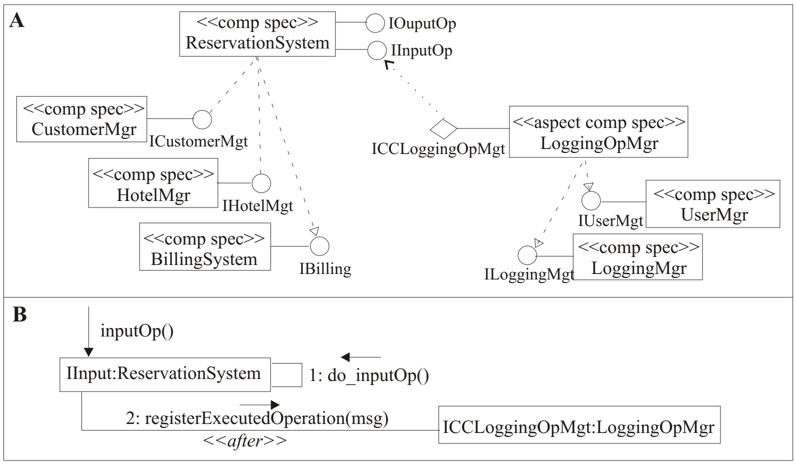


Fig. 1. Component/aspect architecture for the Hotel Reservation System [1]

The semantics for this diagram is that any operation in the interface of IInputOp will be crosscut by the crosscutting component when called by any other component and is enhanced by the behavior specified at the crosscutting interface. Following the guidelines of Clarke and Baniassad [5], the woven behavior may be represented as in Figure 1 (part B) where, for each operation crosscut (e.g. inputOp()), we create a copy of this operation in do_op1() and a new operation inputOp(). In this case, this new operation calls the original operation (do_op1()) and after that calls the crosscutting operation registerExecutedOperation().

3 Design and Reuse of Aspectual Components

There are many possible designs for an aspectual component, all of them influenced by the AOP language used and also by the intended reuse: if white box or black box. A possible way to generalize a component so that it can be reused without change of the code (black box) is to have its code prepared to be used in as many as possible advice configurations such, for example, before, after, and before followed by after. It is difficult to foresee all the possible uses of an around advice because a new behavior is to be executed instead of the original operation; but this can be done for a white box reuse. The different types of advices that are available in the component implementation have to be made clear in its documentation.

The code for the LoggingOpMgr component implemented in JAsCO [2] is shown in Figure 2 (part A). The code is prepared to be used in all three situations

A

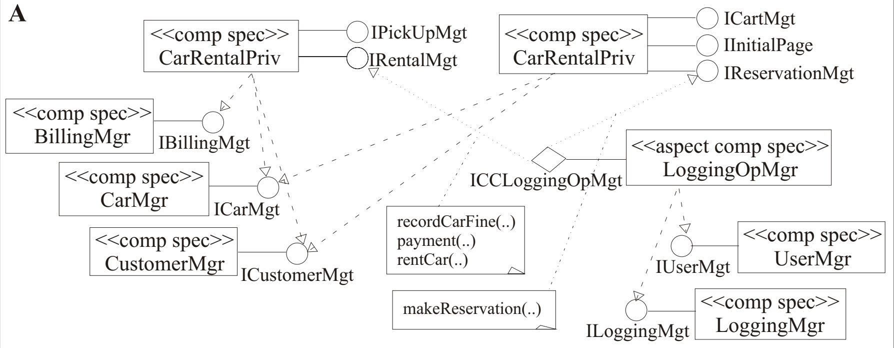
```
class LoggingOpMgr
{
   public void registerExecutedOperation(String msg) {
      // code of registerExecutedOperation
    }
   hook ICCLoggingOpMgt
   {
        ICCLoggingOpMgt(method(..args)) {
            call(method);
        }
        before(){
           global.registerExecutedOperation("BEFORE::"+ msg);
        }
        after() {
            global.registerExecutedOperation("AFTER::"+ msg);
        }
}
```

B

```
static connector installLoggingOpMgr
    LoggingOpMgr.ICCLoggingOpMgt ICCL =
       new LoggingOpMgr.ICCLoggingOpMgt
              ( { * ReservationSystem.makeReservation(..),
                  * ReservationSystem.amendReservation(..),
                  * ReservationSystem.cancelReservation(..),
                  * ReservationSystem.beginStay(..),
                  * ReservationSystem.registerClient(..)});
    ICCL.before();
    ICCL.after();
}
```

Fig. 2. JAsCo Code for the LoggingOpMgr component

A



B

```
static connector installLoggingOpMgr
{
    LoggingOpMgr.ICCLoggingOpMgt  ICCL =
      new LoggingOpMgr.ICCLoggingOpMgt({* CarRentalPub.makeReservation(..),
                                        * CarRentalPriv.recordCarFine(..),
                                        * CarRentalPriv.payment(..),
                                        * CarRentalPriv.rentCar(..)});
    ICCL.before();
}
```

Fig. 3. Component/aspect architecture for a Car Rental System (CRS)

listed above. The connector shown (part B) is realized at the assembly phase and links LoggingOpMgr to the SystemReservation component. The connector defines the advices used (in this case, before and after, in the ICCL.before() and ICCL.after() commands) and the operations to be crosscut.

The LoggingOpMgr's code (Figure 2) is reused as it is in a Car Rental System, whose component and aspect diagram is shown in Figure 3 (part A). The notes used show that now only certain operations of the crosscutting interface have added behavior. Absence of a note means that all of the operations are crosscut. The connector's code (Figure 3, part B) shows the LoggingOpMgr being reused during the assembly phase for the CRS. The connector defines the advice (in this case, before) and the operations that will be crosscut in the interface. Note that the behavior of a crosscutting component does not change when is reused. What changes is the execution of the crosscutting behavior before, after or instead (around) the operations that will have their behavior enhanced.

4 Concluding Remarks

The approach presented also supports the design of functional crosscutting components, not shown in the example. Implementation can also be done using other languages like AspectJ, with some restrictions. For example: using JAsCO, it is possible to change a connector dinamically, what is not possible with AspectJ. Further work is going on to derive more generic designs for black box and white box reuse of crosscutting components as well for crosscutting concerns that are not fully orthogonal such as persistence.

References

1. Cheesman, J.; Daniels, J.: Uml components: A simple process for specifying component-based software. Addison-Wesley, 2000.
2. Suvee, D.; Vanderperren, W.; Jonckers, V.: Jasco: an aspect-oriented approach tailored for component based software development. In: AOSD, 2003, p. 2129.
3. Cottenier, T.; Elrad, T.: Validation of context-dependent aspect-oriented adaptations to components. In: Workshop on Component-Oriented Programming, 2004.
4. Szyperski, C.; Gruntz, G. D.; Murer, S.: Component software - beyond object-oriented programming. Addison-Wesley / ACM Press, 2002.
5. Clarke, S.; Baniassad., E.: Aspect-oriented analysis and design: The theme approach. Addison-Wesley Professional, 2005.
6. Clements, P. C.: From subroutines to subsystems: Component based software development. American Programmer, v. 6, n. 11, 1995.