Dynamic Aspects in SOFA/DCUP

Nguyen Duy Hoa
Charles University, Prague
Faculty of Mathematics and Physics
Department of Software Engineering
Malostranske namesti 25
118 00, Prague 1
Czech Republic

e-mail: hoa@wenya.ms.mff.cuni.cz
WWW: http://wenya.ms.mff.cuni.cz/~hoa

Abstract: The notion of Aspect-oriented Programming (AOP for short) became very popular in the last two years. Starting with the introduction of AOP at ECOOP '97 workshop, several issues in the area of AOP have been addressed. Many researchers agree that AOP can make a rational change in design and programming style. G. Kiszales (in the ECOOP '97 workshop report [MLT97]) defines the main task of AOP as helping programmers "express each of a system's aspects concern in a separate and natural form, and then automatically combine those separate descriptions into a final executable form using automatic tools." Up to date, there are several attempts to solve the problems of cross-cutting and code tangling, but many of them deal with so-called static aspects (i.e. compile-time weaving). In this paper, the author introduces how aspects can be integrated into the SOFA/DCUP component model. Our approach differs from many other approaches in the way aspects are combined into target objects¹. We use so called "dynamic-aspect" concept, where aspects are combined into target objects dynamically at run-

time, instead of static weaving, where aspects are woven into target objects at compile-time using aspect weaver. In our approach, aspects can be plugged in and out dynamically, triggered objects are not aware of those facts. Also, the author would like to show how a component behaviour protocol could be inspected using a notion of dynamic aspects (a component behaviour protocol is a part of the SOFA/DCUP component specification).

Keywords: Aspect-Oriented Programming, cross-cutting, code tangling, SOFA/DCUP, component, behaviour protocol, dynamic aspects, static aspects.

1. Introduction

The SOFA/DCUP project run by CORBA and Distributed System Research Group at Charles University is developing an environment, whose role is to help customer to access software product in an easy way. The main goal of the DCUP (Dynamic Component UPdating) project, which is the part of the SOFA project (SOFTware Appliances), is to try to deal with the software run-time updating problem.

The main goal of component-oriented programming is to hide

¹ We use a term “target objects” instead of “components”, because they have different meanings in aspect-oriented programming and in component-oriented programming (a component in AOP means triggered object).
implementation details behind the component interfaces. On the other hand, aspect-oriented programming deals with separation of concerns on basic functionality and other behaviours. Both concepts are, in some views, very similar. As shown bellow, both concepts can work together to get more flexibility in design and implementation of software components. The two following subsections provide the reader with detailed view of AOP and SOFA/DCUP component model. The rest of the paper presents our approach to integrating both concepts.

1.1. Aspect-Oriented Programming

The notion of Aspect-Oriented Programming (AOP) had been exploited at the Xerox Palo Alto Research Centre, which points out an important problem in software development process, namely cross-cutting and code tangling over basic component functionality. In ECOOP’97 workshop report [MLT97], we can find several areas in which AOP can be applied (including debugging and tracing, persistence, co-ordination in distributed environment, transaction, replication etc.). As stated in [MLT97], those features are not basic functions of a (core) component, but are required. C. Lopes in [LOP97] pointed out that the implementation of these side features is often done manually. A programmer must take care when implementing the features by hand, the output code is very similar for all modules (cross-cutting and code tangling problem [APJ99]). On the other hand, some languages have implicit support for some specific features, but they are not very popular and are hard to use. The need of a separation of concerns becomes very actual question. R.J. Walker [WBM99] in a pilot study shown that AOP would help programmers to concentrate on implementation of the basic component functionality.

ECOOP’97 addressed two key issues that are necessary to be solved in AOP:

1. Aspect-oriented language (AOL for short): what features belong to AOL and what is to be handled by the component language. It is generally agreed that the core component has to deal only with basic functionality (for example core component for bounded buffer only implements methods for putting and taking elements), other features (concurrency control, distribution etc.) are aspects. The second problem is if the general-purpose AOL exists. It is generally accepted that it is impossible to define such that language, i.e. the solution depends on the specific problem and on the chosen platform.

2. How to weave: Generally, there are two ways how to weave: (i) compile-time weaving, where aspects are written in AOL and then woven with core component into target (executable) code using an aspect weaver. AOL is not necessarily same as the component language. This approach is used in various projects including AspectJ [APJ99], D Language Framework [LOP97] etc. (ii) run-time weaving; in this approach, meta-objects providing different monitoring actions are combined into target objects dynamically, here no weaver is needed, this approach is used in [HNP98] and [CPL98].

There are still several open issues to be answered in the area of aspect-oriented programming, including aspect-oriented design, aspect-oriented analysis etc. Many attempts have been made in order to address what is an aspect. Up to date, none of these questions is definitively answered.
1.2. Overview of the SOFA/DCUP Component Model

At the current stage, the DCUP component model consists of two basic parts: the component architecture specification and its behaviour specification. The component architecture is specified using component definition language (CDL) [MEN98]. The behaviour protocols are detailed in [PVB99]. The next two subsections provide readers with brief descriptions of these two parts. Our current CDL is used to generate target codes in the Java language.

1.2.1. Component Model

Figure 1 shows a typical structure of a DCUP component. Each component contains one component manager and one component builder, which are responsible for managing the particular component. A component can have a several implementation objects and/or number of sub-components to provide its functionality. Objects implementing component services are not accessed directly, but via the particular wrappers only. Also a component may require a number of services provided by other components to complete its own task. Interfaces (provided or required) by the component are specified using standard OMG IDL.

```
Figure 1. SOFA/DCUP Component Model
```

Updating a component means replacing its replaceable part by a new version. Note that all references to inside objects are redirected through wrappers, which are locked while the component is in updating phase. The update can be performed during component run time. The component builder is responsible for successful updating of the component.

At this stage, access to component services (via wrappers) is not controlled. The author believes that with the notion of dynamic aspects, access to wrappers can be monitored and controlled quite well using the dynamic aspects. Since the code implementing wrappers is automatically generated by a compiler from the CDL specification, there is no risk of code inconsistency.
1.2.2. Behaviour protocols

The component behaviour is described at three abstraction levels:

1. **Interface behaviour protocol**: describes behaviour of a concrete interface (i.e. an instance of a specific type at run-time).
2. **Frame protocol**: specifies actions (method invocations) between different interfaces (provided or required) inside the component.

```c
interface ITransaction
{
    void Begin();
    void Commit();
    void Abort();
    protocol: (Begin; (Commit + Abort))*
};
```

*Figure 2. Interface Protocol*

More details on protocol grammar and semantics can be found in [PVB99].

1.3. Summary

The goal of this paper is to present the concept of so-called “dynamic aspects” used in the SOFA/DCUP component model. We also show how a component behaviour protocol can be inspected using dynamic-aspect approach (inspection means here a correct order of method invocations specified by a behaviour protocol).

2. Dynamic aspects

This section introduces our approach to integrating aspects into the DCUP component model. Since many software components are shipped to clients (either end-users or other developers) in a binary form, static aspects cannot help with additional debugging, monitoring, adding new functions etc. As shown above, a DCUP component is composed from Component Manager (CM for short), Component Builder (CBuilder for short), various wrappers and implementation objects. The first question appears is what objects the aspects will affect. Triggering on implementation objects and CBuilder is not a good idea, because programmers can modify them manually, and there is certain risk of code inconsistency. Also triggering CM is not a very good idea, because the scenario of method invocation on CM is always the same. At this stage, we concentrate only on triggering wrappers; once a wrapper is triggered, its embedded implementation object will be inspected as well. There are three possible ways to change object behaviour at run-time.

There are two possible ways to provide “general-purpose” dynamic aspects in Java environment: (i) Modification of JVM message dispatching system, (ii) Binary code modification by a specialised class loader. Both methods are applicable for any systems, not only DCUP components. Unfortunately, they have a large disadvantage (large
overhead in the first approach, impossibility to add/remove aspects during run-time in the second one). Thus, both methods are impractical.

Our approach, in contrast with the two above, is highly flexible. The idea is very simple; each component has one Aspect Manager which holds a list of all aspects (inspection objects – the list is implicitly empty) that operate over objects (wrappers in our context) within this component. Aspects are added and removed dynamically as needed. Each aspect must at least implement three methods: (i) beforeAction: for controlling pre-condition or logging activities etc. (ii) afterAction: to be called when a particular method of the inspected object completed its work. (iii) onError: for exception handling. Each method invocation is first redirected to AspectManager that invokes all beforeAction methods; if no errors occurs, control is passed to the target object for its own computation, otherwise control is passed on directly to the onError part. After the target object completes its work, and before returning to the caller, the after actions are invoked.

The CDL compiler automatically generates code for redirection of calls to AspectManager. Following is the fragment of the code:

```java
public class AccountWrapper
    implements AccountInterface
{
    public void withdraw(double amount)
    {
        AccessGuard.inAccess(true);
        // Prevents from conflicts with updating
        try
        {
            // Prepares parameters’ list
            Object[] params = new Object[1];
            params[0] = new Double(amount);
            AspectManager.doBefore(this, "withdraw", params);
            Target.withdraw(amount);
            AspectManager.doAfter(this, "withdraw");
        }
        catch (Exception e)
        {
            AspectManager.onError(this, "withdraw", e);
            AccessGuard.inAccess(false);
        }
    }
}
```

**Figure 3.** Redirection of calls to aspects

We support three kinds of target-object inspection: (i) per-instance inspection, (ii) per-class inspection and (iii) multi-class inspection. Support for semi-regular expressions (wildcard “*”, list, and alternatives using braces “[“ & “]”) is provided in order to monitor multiple classes or multiple methods. Following examples illustrate those situations:
TAspectManager am = parentCM.getAspectManager();
am.addAspect(aspect1, this, "methodOne, methodTwo");
   // per-instance aspect
am.addAspect(aspect2, "MyClass", "+");
   // per-class aspect, this affects all methods of all
   // instances of the given class
am.addAspect(aspect3, "Window", "+");
   // multi-class aspect, this affects all instances
   // of the class with a name starting with "Window"

Figure 4. Adding aspects

Note that per-instance aspects are not recommended in general due to
overhead, but they are suitable for keeping an object execution trace.

3. Employing static aspects using dynamic aspects

In many cases, there is a need having aspects that exist during the entire object
lifetime, thus it is necessary to configure them statically rather than adding them
dynamically. Static aspects can be specified directly in the component
specification and are included into target code automatically by the CDL compiler,
thus readability of the code is better (instead of other possible ways such as
configuration file etc.) To achieve this, a new construct is introduced into CDL.
One of the possibilities is the following:

```
advise component SomeComponent
{
   per-class ClassA, ClassB
   ("SOFA.DCUP.Log", "SOFA.DCUP.Debug");
   per-instance ClassC {"Tracing"};
};
```

Figure 5. Configuring static aspects

There are still several open issues to be solved (including guard condition,
support for transaction, persistence, concurrency control etc.). Guard
conditions should be supported (at the
specification language level) in both
semantics: pre/post – condition and on-
enter/on-exit actions. Following
examples show the idea:
As illustrated in the example above, both ways are equivalent in computation power. Each has advantage and weakness as discussed bellow.

- **Pre/post condition**: is language independent; i.e. all variables in the condition expression must be interface attributes. On the other hand, it is not too powerful as on-enter/exit action, because it supports only boolean expressions. User-defined exception could not be supported in this approach. Instead of this, a system exception (so-called EConditionViolation) will be thrown.

- **On-enter/exit actions**: Unlike pre & post conditions, they have more computation power in order to evaluate particular condition, thus they can be more practical in some cases. On the other hand, this approach can not be full language independent, and hence it mixes up abstraction level (interface specification) with implementation level.

In many cases, the pre & post-conditions as a boolean expression is strong enough to specify interface behaviour. At the current stage, we support only this kind of specification. If a programmer want more powerful pre-conditions, he/she can simply apply using dynamic aspects (by implementing TAspectInterface as describe in the previous section). Variables appeared in the condition expressions must be declared as interface attributes, method parameters, or constants. CDL compiler than will compose all pre/post-conditions into a final aspect (Java class). Note that in some cases, there is a need for aspect
own attributes, but this can be achieved easily using dynamic aspects.

4. Integrating behaviour protocols

Behaviour protocols are true regular-like expressions; thus it is possible to construct an automata, which is equivalent to the protocol. Therefore, it is not hard to keep a track of the object execution. An object implementing such automata may play a role as an aspect in order to keep the object execution constrained in a predefined way. Such an aspect must be per-instance. The following example illustrates this idea (using interface ITransaction with protocol shown above in 1.2.2):

```java
public class ITransactionWrapper
    implements ITransactionInterface
{
    TAspectManager AspectManager;

    public ITransactionWrapper(
        TComponentManagerInterface parentCM)
    {
        AspectManager = parentCM.GetAspectManager();
        TProtocol protocol = new TProtocol();
        protocol.setProtocol("{Begin; (Commit + Abort)}*");
        AspectManager.add(protocol, this,
            protocol.getAffectedMethods());
    }
}
```

Figure 7. Integrating behaviour protocols

There are still several open issues on behaviour protocol inspection. One of such questions is what to do when a protocol violation is detected. Should the aspect block the target object or raise an exception. Blocking semantic is suitable for a multi-threaded application (e.g. producer/consumer, bounded buffer etc.), but is not applicable for single threaded application (because the application will never be unblocked). On the other hand, blocking semantics is not applicable for a number of problems, for example transaction, where an invocation of Commit/Abort without previous Begin is not acceptable; hence there is no sense in waiting for someone to begin a transaction. The decision of where the waiting semantic and where the exception raising semantics is applicable, is impossible. At this stage, aspect simply raises an error exception. The second open issue raises when the re-entrant operator is used. S. Višnovský [VIS99] shown that an automata can be constructed from protocols without the re-entrant operator. In order to modelling this operator, we will need a more powerfull tools.

5. Evaluation

Up to date, there are several works that deal with AOP, most interesting are AspectJ [APJ99], D Language Framework [LOP97], Synchronisation Rings [HNP98], Meta Object Composition [CPL98]. All of them are for specific purpose only, AspectJ is suitable for tracing and debugging, D Framework focuses only on distributed computations (synchronisation and data
transferring), the synchronisation rings approach is suitable for transaction purposes (including nested transactions). AspectJ and D Framework provides pure static aspects, while the two others (mentioned above) are “dynamic-aspects”. In [CPL98], each object is monitored by the system meta-object, which has the pre-list and post-list of meta-objects for monitoring of the particular object. This approach has a disadvantage that there is a large overhead, i.e. not every object needs to be monitored, also there is needed to modify message dispatching system in order to redirect all method invocations to the system meta-objects. Approach presented in [CPL98] is more similar to our approach, but our concept of dynamic aspects is more flexible, i.e. we allow monitoring of objects as needed.

Our current implementation is in Java (note that the whole SOFA/DCUP architecture is realised in JAVA too). On the other hand, applying the presented concept in other platforms (especially in CORBA Component Model or Enterprise JavaBeans) should not cause any large problem. At this stage, full dynamic aspect manager system is completely implemented and a partial support for behaviour protocol is provided. Future work will include a full support for behaviour protocols (especially for parallel issues) and static configuration of aspects. A performance measurement is also planned.

6. Conclusion

This paper shows that Aspect-Oriented Programming can be realised in a highly dynamic and flexible way. Aspects can be added or removed dynamically during application run time. It is also shown how a component behaviour protocol inspection can be realised through the notation of aspects.

At this place, I would like to express special thank to František Plášil for his many valuable suggestions. The author’s appreciation goes also to all members of CORBA and Distributed System Research Group at Charles University for prototyping and implementing SOFA/DCUP architecture. Finally, I am grateful to Petr Tůma for proof reading the text.

7. Reference


