

Cooperating Aspects

Arno Schmidmeier

AspectSoft

Lohweg 9,

91217 Hersbruck,

Germany

Arno@aspectsoft.de

Abstract.

There are many different options when using AOP tools or languages to solve similar problems. Modern AOP languages offer new features and design ideas, through the combination of object and aspect oriented features. Patterns are therefore required to guide AOP developers and architects around the common pitfalls and unleash the power of Aspect Oriented Programming.

Most AOP examples and AOP patterns describe scenarios, where aspects work on a set of objects. This is not the only use case of aspects. In programs of advanced AOP developers aspects do often cooperate with objects and other aspects. This paper describes two core patterns of interacting and cooperating aspects.

Introduction

Objects have been quite successful in the past to modularize and organize the implementation of the core requirements. Object Oriented technology created new architectures and solved problems. It proved quite successful in creating new architectures. However due to the increasing complexity of modern programs non business concerns got more and more important. Typical examples include transaction handling, persistency, tracing, logging, and contract checking, etc. Object technology failed to handle these additional concerns in a modular way. Aspect oriented programming (AOP) proved quite successful in modularizing these concerns [5][6][7]. However AOP is quite a new programming paradigm, which is going to start to enter mainstream software development. Quite a lot of newcomers to aspect oriented technologies have often the following wrong initial expectations:

- That each aspect should be a totally independent unit, which is often not practical if not impossible. The often communicated concept [8] of modularization of concerns, which was one of the initial motivations of AOP, creates often this expectation. Based on my experience with AOP-systems, I am convinced that this initial expectation is not valid.

E2-1

Copyright Arno Schmidmeier 2005
All rights granted for the purpose of EuroPlop 2005

- that aspect should only operate on objects, and objects should never operate on aspects. In other words aspects can call methods of objects during the advice execution, but objects should not invoke or access aspects willingly.

I am convinced that this strict opinion limits the useful design space significantly. I decided to present two important patterns, where objects and aspects interoperate and cooperate gracefully. These are:

- The Session Aspect pattern is used to create a unique environment for the execution of a control flow.
- The Collaborator pattern is used to pass crosscutting information to an aspect.

This paper starts with a discussion of the main forces of AOP application in current projects. It continues with a discussion of both of the patterns. Finally the appendix contains a short introduction to the features and ideas of AspectJ [9],[21],[22] in order to support the reader, who may not yet be familiar with AOP.

All these idioms and patterns have been extracted from commercial AspectJ projects in which I was involved. All the samples and code fragments are written in AspectJ and the names are influenced deeply by AspectJ terms. However, the patterns are useful outside AspectJ, too. All patterns are applicable to other AOP Frameworks (e.g. aspektwerks[10], JBoss-AOP[11], Nanning[12]) and languages like Sally ([13]). Especially the Collaborator pattern is very useful in other AOP frameworks for different languages (e.g. small python aspects [20]), which do not support such a cflow- pointcut like construct.

Typical Forces in AOP programs

Let us examine first the key forces in AOP programs. –what are the main forces which might constrain us in the application of these patterns:

- *Verbosity of the code*: How verbose a method, or a class is. Is everything, what happens during the execution of a method, written verbatim inside the method, are significant hints given, e.g. by method calls, annotations, etc? A very verbose code makes it easy to understand the composition of activities inside a method or a class. This increases the changes that less skilled developers or new developers on the project spot what is going on during the execution of a method at a first glance. Verbose code in non trivial programs bloats the code base (Lines of Code (LoC) is normally increased at least by one magnitude), because it increases code redundancy and reduces modularity. In non verbose code everything is done in the background by “some magic”. In general you want your code as less verbose as possible, but a verbose as required by the skill set of the majority of your developers.
- *Cleanliness of the base application code*: Do classes and methods from the domain model contain only domain specific code? Resolving this force enables domain specialist or technical less skilled developers, with huge domain knowledge, to reason, develop and maintain the domain model and code more efficiently. It helps to establish and maintain a common domain model. Resolving this force in non trivial systems decrease the force Verbosity of the code.
- *“Technical complexity”*: How complex is the technical design? A low technical complexity helps new or less skilled developers to reason about the architecture or a module. However if the technical complexity is too low, the system will not work as required, or the features can only be implemented through code bloats. (e.g. “the copy on write”) approach, which many not maintainable COBOL programs face today. If the technical complexity is too big, it gets to hard to understand the core mechanisms

E2-2

of the system. The technical complexity of business domain code should be less than the technical complexity of infrastructure code. There are no absolute numbers possible for an acceptable technical complexity. The acceptable complexity depends from the team structure, education and familiarity with the used technology of the team members, size of the project and code base and the used technologies. Some subsystems may have a different “technical complexity” if different individuals or (sub)-teams work on them.

- *Right sized Coupling and dependency*: How much does some element depend from some other element? If systems are very strong coupled, it is hard if not impossible to change one without changing the other. Lose coupling on the other had comes with an increased initial overhead and runtime penalties. Quite a lot of classical patterns (e.g. the Observer [1], publish and subscribe) deal with the problem of Coupling and dependency. Several of them can be easily implemented with aspects [23][24]. Crossing dependencies, where Element A depends from element B on a technical level, but B depends from A at a conceptual level should be avoided. An example of such a scenario and how it could be resolved is given in [25].
- *Non invasiveness*: Can you add a specific new feature to an element A without changing A. Resolving this force means that you can add or remove extra features to an module without any changes in the module. This is often very useful for maintenance.
- *Code bloat (Lines of code)*: How many lines of code have been required to implement a specific problem? Initial development time is at least linear to the lines of code¹. If code bloat can be avoided, development cost can be easily reduced.
- *Modularity*: How modular is the design? Does every module deal only with one concept? Is code written once and only one? Resolving this force makes modification in general cheaper, because for quite a lot of changes you have to touch only one place in the code base. Modules are a prerequisite for potential reuse, the holy grail of software development. Non modular requirements result normally in scattered and tangling code. Non modular code in larger systems causes a code bloat, normally indicated by following typical symptoms:
 - *Scattered code*: How often do we find the same code fragments at different places in the code base?² Scattered code results in code bloat, and reduces the cleanness of the base application code. To verbose code results often in scattered code.
 - *Tangling code*: How interwoven are code fragments dealing with different requirements. Non tangling code improves maintainability.

¹ Assuming same programming language, same programming style and similar problem complexity.

² Repetitions for code fragments, which are enforced by the problem domain of the program or subsystem, do not count. (e.g. Calls to mathematical functions like sqrt() in mathematical programs or calls to biginteger libraries in financial programs)

Session Aspect

How can you provide unique environments for activities during the execution of code blocks,?

Assume you want to use an object persistence framework like hibernate [17] or JDO [14] someone has to deal with transactions. Frameworks like J2EE [15] and Spring [18] can do the transaction handling for you. But often you cannot use such a framework for several reasons, e.g. licensing, platform and deployment issues. In such a scenario you are left alone with the task to do the transaction handling alone.

You have to get a reference to a transaction manager, as a next you have to open a transaction before you can interact with persistent objects and after the job is done you have to close the transaction.

```
public void bar()throws Exception{
    TransactionManager tm=
        SessionTool.getTransactionManager();
    tm.begin();
    //do some stuff
    tm.commit();
}
```

You can move this handling of the transaction easily to an advice. Following advice can do that for you.

```
Object around(): atomarmethods(){
    TransactionManager tm=
        SessionTool.getTransactionManager();
    tm.begin();
    Object toReturn=proceed();
    tm.commit();
    return toReturn;
}
```

However, this solution is sometimes not sufficient. You often have to call additional methods during the processing of “do some stuff” (or the proceed call in the AOP solution), which needs data from the advice section.

In a persistent framework for example, you often have to call persistence infrastructure methods, e.g. query after a collection of objects, to make objects persistent, to delete objects from the database, to cluster objects, to reinitialize objects, etc. Other aspects are sometimes responsible for these calls; in other cases old fashioned objects should do the work. Regardless, who does the job; he needs additional information like the database connection, the active database session or the transaction. This information should be present in the unique environment, which is spanned by the transaction. This environment belongs to the transaction concern and it

Problem:

Example:

should be therefore in the responsibility of the aspect to create the unique environment.

Forces:

We want to minimize the code bloat and increase the Cleanliness of the base application code:

- by eliminating scattering code (setup and teardown code of the special environment)
- by eliminating tangling code (passing around a special reference parameter to an object holding the special environment)

We often want to increase modularization, e.g by moving the unique environment to a library.

Solution:

Ensure that all activities are wrapped up in a program structure, which is selectable with pointcuts. The structure will most often be simply a set of methods. Select all of these “*Session joinpoints*” with pointcuts. Wrap an aspect declared as `percfw()` or `percfwbelow()` around the sections of the control flow of your application. This declaration of the aspect ensures that there is one unique aspect for each activity. This unique aspect can be easily accessed with the `aspectOf()` accessor method during the execution of the code blocks. It is therefore the ideal container for all the special data, because, all objects and aspects have access to the session data by calling the `aspectOf()` accessor of the session aspect. This call is only valid during the execution of the code block. An `aspectOf`-call is invalid outside of the execution of the code block. The aspect can perform in addition some special action at the beginning and the end of the code block to set and clean up the special session environment. (E.g. to open or close the transaction in the sample and initialize some session data, (e.g. set a transactionmanager variable in the sample)).

Sample

Sample Transaction Aspect

```
public aspect TransactionAspect
    percfw(atomarmethods()) {

pointcut atomarmethods(); //...;

private TransactionManager tm;
private Session session;

public TransactionManager getTransactionManager ()
{
    return tm;
}

public Session getSession() {
    return session;
}
}
```

E2-5

Copyright Arno Schmidmeier 2005
All rights granted for the purpose of EuroPlop 2005

```

Object around() throws Exception:
    atomarmethods() && !cflow(atomarmethods()) {

        session=sessionFactory.openSession();
        tm=new LocalTransactionManager(session);
        tm.begin();
        Object toReturn=proceed();
        tm.commit();
        session.close();
        return toReturn;
    }

private static SessionFactory sessionFactory;//=...
}

```

This small sample eliminates the tool class SessionTool. All objects and aspects, which need access to the session or the transactionmanger can get obtain them through the unique environment, which is provided by the TransactionAspect by calling:

```

TransactionAspect.aspectOf()
    .getTransactionManager ()

```

or

```

TransactionAspect.aspectOf().getSession ()

```

Resolved forces

This pattern increases the modularity, because all session setup and tear down code can be moved into the Session aspect. Because the session aspect contains normally code from quite a lot of methods it reduces the lines of code significantly (e.g. 20-30% by refactoring transaction and session management to session aspects in the space of persistent objects), reduces crosscutting and scattered code significantly. This improves the cleanliness of the base application. .

However the verbosity of the code is reduced by the application of the session pattern and the technical complexity is increased. Both forces make is harder for AOP-newbies and domain experts to understand, maintain and debug the interactions with the session aspect. Design stories, usage samples and metaphors create easily a sufficient awareness and understanding for the session aspects, which they normally face during there daily job. If you want to remain some verbosity in the code you could use the pattern marker interface or annotations (if your AOP platform/ language provide them) to remind the reader of the code of the session property. If you use marker interfaces or annotations you should balance verbosity versus tangling and redundant annotations and marker interfaces.

E2-6

Copyright Arno Schmidmeier 2005
 All rights granted for the purpose of EuroPlop 2005

My experience has shown me, that a developers often want to apply the pattern session method in a legacy code bases, which have been written with no aspects in mind. These legacy code bases contain sometimes code with several code blocks, each wrapped with a sessions block and all embedded in one method, like it is demonstrated in following code sample:

```
public void legacyMethod() throws Exception {
    TransactionManager
        tm=SessionTool.getTransactionManager();

    tm.begin();//begin first Transaction
    //do the first transactional stuff
    Collection<Bar> col=foo.getChildren();
    for (Bar obj : col) {
        if (obj.getValue()>100){
            tm.rollback();
            throw
                new InvalidValue("The value is to big");
        }
    }
    foo.setOk(col.size()>10);
    tm.commit();//end first Transaction

    tm.begin();//begin second Transaction
    //do the second transactional stuff
    try {
        foo.activity1OfTransaction2();
        foo.activity2OfTransaction2();
        foo.activity3OfTransaction2();
    } catch (Exception e) {
        tm.rollback();
        throw e;
    }
    tm.commit();//end Second Transaction
}
```

Such a scenario makes it very hard, if not impossible to select the session method with pointcuts and apply this pattern in the legacy code, because the structure of the program is buried in white spaces and in comments. AOP languages do not provide extremely fine grained pointcuts, which would us enable to select these methods. Such a pointcut, (e.g. based on line numbers) would be too brittle in every day's program.

If you do apply this pattern in such a code base you have to ensure, that all activities are wrapped up in a program structure, which is selectable with pointcuts. To shape a wrappable program structure, you often have to perform some refactorings, especial the refactoring pattern extract method [16]³ in order to expose useful joinpoints. In the sample you should extract the two transactional blocks to the two methods `doTheFirstStuff()` and `doSomeOtherTransactionalStuff()`. From these methods you can apply the

³ We believe anyway that such code would benefit from refactoring anyway, because code, which does need an own session, should be grouped in its own modular unit, at least a function.

AOP refactoring move to advice easily. You end up with following domain method structure:

```
public void legacyMethod2() throws Exception{
    doTheFirstStuff();
    doSomeOtherTransactionalStuff();
}

private void doSomeOtherTransactionalStuff() throws
SystemException, exception {
    try {
        foo.activity1OfTransaction2();
        foo.activity2OfTransaction2();
        foo.activity3OfTransaction2();
    } catch (Exception e) {
        TransactionAspect.aspectOf()
            .getTransactionManager()
                .rollback();
        throw e;
    }
}

private void doTheFirstStuff() throws
    SystemException, InvalidValue {

    Collection<Bar> col=foo.getChildren();
    for (Bar obj : col) {
        if (obj.getValue()>100){
            TransactionAspect.aspectOf()
                .getTransactionManager ()
                    .rollback();
            throw
                new InvalidValue(
                    "The value is to big");
        }
    }
    foo.setOk(col.size()>10);
}
```

A call or executionpointcut can select the methods doTheFirstStuff() and doSomeOtherTransactionalStuff() easily and reliable.

Some AOP languages and framework do not support an aspect declared as percfow or a percfowbelow. An AOP library can provide easily a percfow or percfowbelow implementation..

Collaborator

Problem

How can you transfer crosscutting information from the base application to an aspect?⁴

Example

Providing additional information for the pointcut of an advice sounds for some AOP beginners a little to AOPish. For them I have another short sample. Here is it: Assume you have to bill a bunch of methods, which interact with an expensive system on a user base. It is easy to write an aspect with an advice, which does the billing. Following advice might do it.

```
Object around():expensiveMethod(){
    Object toReturn=proceed();
    String user=getUser();
    billUser(user,thisJoinPoint);
    return toReturn;
}
```

However, this advice needs to know which user to bill. (printed bold in the above sample) Collecting and providing the user information is a crosscutting concern to the advice and to the code of the “base application”. We have quite a bunch of expensive methods which must be billed. Each of these methods may be called from several different call stacks and APIs, e.g. a command-line interface, through a web-interface, etc. The user information must be captured at each of these places and passed through the call stacks to the methods which should be billed. It is quite likely that a traditional application might be easily scattered with methods which do either validate and get the user or pass the user information around. So how can you transfer the scattered and crosscutting information, which user to bill into the advice. This is again a special case of the general problem question from above.

Forces

We often want to minimize the code bloat and keep increase the Cleanness of the base application code:

- by eliminating scattering code (collecting the crosscutting information from many places)
- by eliminating tangling code (passing around the collected information)

We often want to increase modularization, e.g. by moving the information collecting code to a library or implement a new pointcut library.

OR

We often have to add new features in a non invasive way, (e.g. code freeze or non available code)

⁴ The same is true for the opposite direction, where information has to be transferred from the aspect to the base program. We use only the direction from the base program to the aspect for an easier problem solution and discussion section. The opposite direction works identically.

Solution

Have two sets of aspects the Collaborators and the workers. The workers aspects implement the additional functionality. The Collaborator aspects provide the worker aspects with additional information, which they need to fulfill their job. The Collaborators must collect the information and store the information in a well known location. Whenever the workers need this information they can get it from the well known location, where the collaborators have stored the information.

Samples

Also the password concern from the billing sample above can be implemented easily with following aspect.

```
public aspect GetUser perCflow(initialUserMethod(string)) {
    public pointcut initialUserMethod(String username);
    before(String username):initialUserMethod(username) {
        this.username=username;
    }
    private String username;
    public String getUsername() {
        return username;5
    }
}
```

The implementation of the getUser() call looks now:

```
public String getUser(){
    return GetUser.aspectOf().getUserName();
    //return "? How do we know the user";
}
```

Resolved forces

The application of the Collaborator pattern, especially in cooperation with other aspect patterns allows us to minimize the required lines of code, by eliminating tangling and scattering even in aspect code. The collaborator is often the enabler for a new “generation of modularity” by enabling new aspect libraries. One side this powerful pattern allows extreme modular and elegant AOP solutions, because this pattern is an extreme trade of modularity, separation of concerns and power versus technical complexity and reduced verbosity, however quite a lot of non AOP aware developers consider the effects of the application of this pattern as magic.

I currently like to apply this pattern therefore only in environments, which need carefully crafted code, e.g. development of core infrastructure

⁵ getUsername() is normally implemented with a more sophisticated algorithm and approach. In a J2EE/Web application for example the user could be often obtained from a cookie or some other servlet session store.

E2-10

functionality, AOP libraries and runtime platforms and I avoid to use this pattern in the core business domain, which will be modeled and maintained by domain experts, AOP ignorant or less skilled developers.

The application of the Collaborator pattern was often the core prerequisite for bug fixes and adding new features in code bases, which could not be changed in an invasive way.

Discussion

I have observed two extreme major application scenarios of this pattern:

- Providing information for the implementation of the advice.
- Providing information for an educated selection of the joinpoints

Mixtures are also common see the discussion of possible usages below.

In the first scenario, the worker aspect uses the additional information, which he gets from the Collaborator for an efficient processing of his advice.

In the second case the additional information is used to narrow the pointcut more efficiently. It is a common way to implement missing pointcuts in aspect oriented languages or frameworks. Some frameworks like nanning, AspectS [19], Sally, Small Python Aspects do not have a cflow or a cflowbelow pointcut and this pattern is often used to implement these missing pointcuts. Following Collaborator aspect provides a generic cflow implementation for small python aspects.

```
cflow_dict={}

class CFlow(aspect):
    def
__init__(self,method,cflowkey,originalmethod=None):
    aspect.__init__(self,method,originalmethod)
    self.cflowkey=cflowkey

    def around(self,*args,**argsh):
        print args
        thread=currentThread()
        print thread
        needscleanup=True
        if cflow_dict.has_key(thread):
            key_dict=cflow_dict[thread]
            if key_dict.has_key(self.cflowkey):
                needscleanup=False
            else:
                key_dict[self.cflowkey]=self.cflowkey
        else:
            key_dict={}
            key_dict[self.cflowkey]=self.cflowkey
            cflow_dict[thread]=key_dict
        print cflow_dict
        toReturn=self.proceed(*args,**argsh)
        if needscleanup:
```

E2-11

```

        key_dict=cflow_dict[thread]
        del key_dict[self.cflowkey]
        if len(key_dict)==0:
            del cflow_dict[thread]
    return toReturn

@classmethod
def isCflow(clazz,cflowkey):
    thread=currentThread()
    if cflow_dict.has_key(thread):
        key_dict=cflow_dict[thread]
        if key_dict.has_key(cflowkey):
            return True
    return False

```

In programming languages, which do provide an if-pointcut (like AspectJ), an if-pointcut is often used, the ideal use-case for the second scenario. If an if-pointcut is not applicable, the additional information must be used as a parameter for the pattern pointcut method [4]. Even AOP languages, like AspectJ, with a feature reach pointcut language need the second usage of this pattern quite often to implement new pointcuts, like the “dflow”[26] or the “needed” perthisflowbelow pointcut from the observer sample below.

A very common example for the use of aspects is the famous observer pattern. Sometimes you have a specific observer for each object. The individual observer can be easily implemented with an aspect which is declared as perthis() or pertarget(). Following sample demonstrates such an aspect. The state of each graphical component should be observed. Following class, which contains another graphical component panel, should be resized.

```

public class SamplePanel extends Panel {
    private Panel child;

    public void setBounds(int x,int y,int w,int h){
        super.setBounds(x,y,w,h);
        //resize child also ...
        //maybe through a layout manager ...
    }
    //some other stuff
}

```

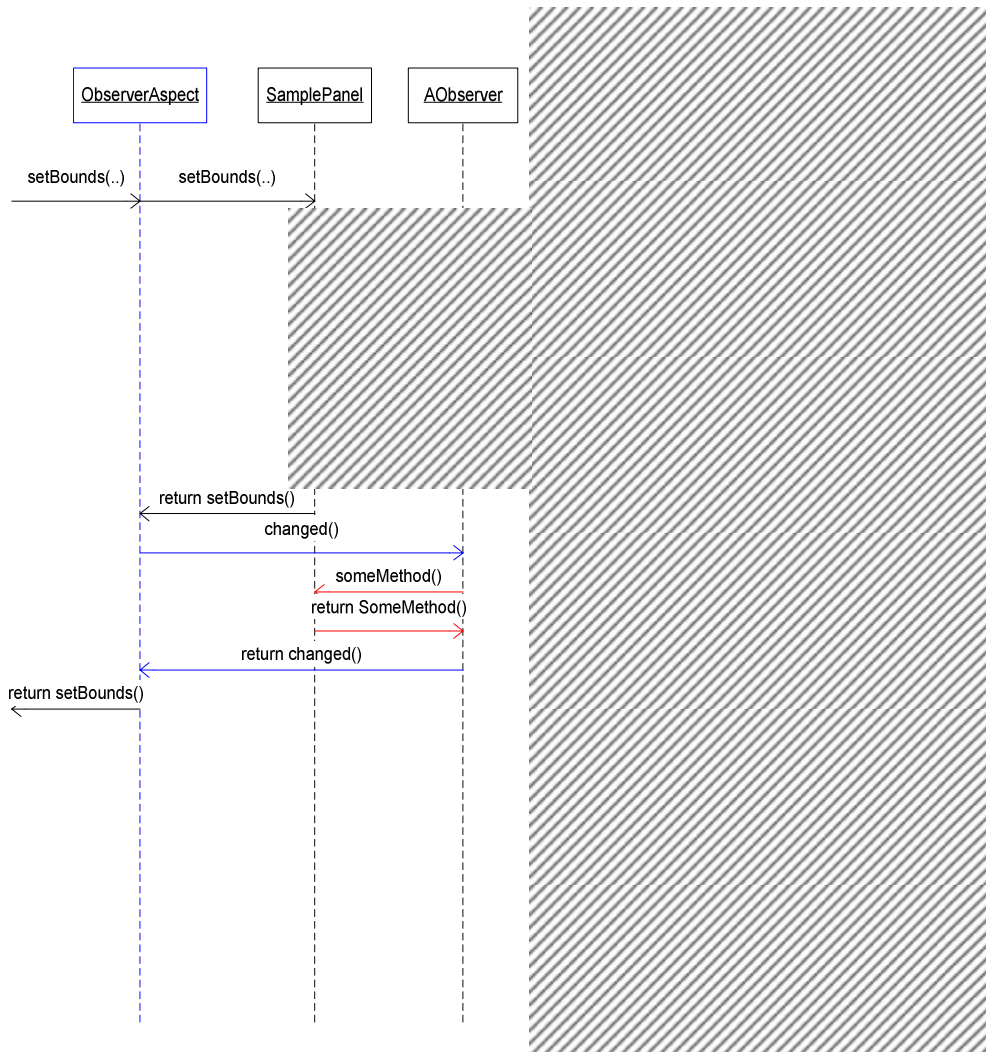
However, some of the writing methods might call other writing methods. E.g. a call to setBounds(int x,int y,int w,int h) might be implemented with calls to move(x,y), and setSize(). Both of these calls would result in an observer call as well. These additional observer calls are often just pure overhead, but sometimes they lead to real problems, e.g. infinite recursion. Just imagine, what happens if someone on the observing call stack calls a writing method of the observed object. This method triggers again the advice execution, which causes the invocation of the writing method, and so on till the program runs out of stack. An easy solution is to perform the

E2-12

advice only, if you are currently not in the controlflow of the joinpoint. Following cflowbelow pointcut can easily guarantee that.

```
pointcut observablecalls():
    writingcalls()&&!cflowbelow(writingcalls());
```

However, graphical elements as in the example contain often other graphical elements. And these elements might be observed, too. Resizing the outer graphical element does often change the size of the inner graphical elements. The observation notification calls in the shaded area should not be excluded by the restriction clause of the pointcut.



However, the sample pointcut from above doesn't select the joinpoints in the shaded section, because the pointcuts are in the cflowbelow of the above

E2-13

pointcut writingmethods(). The cflowbelow pointcut must be therefore a cflowbelow() pointcut, which is only limited to the control flow of a method of a specific object. Such a pointcut might be called cflowbelowperthis() in the AspectJ notation. However AspectJ and all other current AOP languages do not provide such an advice (yet). So you have to collect the required information and calculate the cflowbelowperthis() pointcut by yourself. And this is a crosscutting concern by its own.

Following aspect calculates the cflowbelowperthis pointcut of the graphical sample from above.

```

aspect CflowBelowPerThisAspect{
    Hashtable table=new Hashtable();

    Object around():ObserverAspect.writingcalls(){
        Object obj=thisJoinPoint.getThis();
        boolean needremove=false;
        if (!table.containsKey(obj)){
            needremove=true;
            table.put(obj,obj);
        }
        Object toReturn=proceed();
        if (needremove)
            table.remove(obj);
        return toReturn;
    }

    public boolean cflowBelowPerThis(Object obj){
        return table.containsKey(obj);
    }
}

```

I ignored thread synchronization and error handling for the sake of clarity. The pointcut for the observer aspect might now look like:

```

writingcalls()
&&
    !if(
        CflowBelowPerThisAspect.aspectOf().
        cflowBelowPerThis(thisJoinPoint.getThis()
    )

```

If this pattern is combined with the pattern pointcut method we have the mentioned mixture of these patterns.

Often the set of collaborator aspects consists of one or two aspect classes. If this set of collaborators consists of only one aspect class, it is quite natural to implement the collaborator as a session aspect. In this case the information is normally stored in the session context. (a member variable of the Collaborator).

Conclusion

The discussion of these two patterns demonstrated that there is a need for cooperation of aspects and objects. The order of the cooperating aspects is essential, if the cooperation of aspects and objects should be successful as demonstrated in these patterns. Aspects participating in one of these patterns are hard to remove from a system. If they could be removed at all, they could only be removed if the functionality provided by the whole pattern is removed.

Acknowledgements

There are several people who helped me writing this paper. I want to say thanks to my shepherd James Noble for his very supportive and insightful comments, which helped to improve the paper.

I want to thank Stefan Hanenberg for his fruitful discussion after the pattern discovery.

Finally, I want to say thanks to my wife Eva and my daughter Sophia for their patience, while I was writing this paper.

References (tbd)

- [1] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: “*Design Patterns: Elements of Reusable Object-Oriented Software*,” Addison-Wesley, 1995.
- [2] Hanenberg, S.; Costanza, P.: “*Connecting Aspects in AspectJ: Strategies vs. Patterns*”, First Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD’01, Enschede, April, 2002
- [3] Schmidmeier, A.; Hanenberg, S.; Unland, R.: “*Known Concepts implemented in AspectJ*”, 3rd Workshop on Aspect-Oriented Software Development of the German Informatics Association, March, 2003
- [4] Schmidmeier, A.; Hanenberg, S.; Unland, R.: “*AspectJ Idioms for Aspect Oriented Software Construction*”, EuroPLoP’03 June, 2003
- [5] Schmidmeier, A.: “*Using AspectJ in Component-Based Architectures on the Server Side*”, Invited talk at AOSD’01, Enschede, April, 2002
- [6] Schmidmeier, A.: “*Using AspectJ to Eliminate Tangling Code in EAI Activities*”, practitioners report at AOSD’04, Boston
- [7] Laddad R: “*AspectJ in Action*” Manning Publications
- [8] Gradecki J.D, Lesiecki N.: “*Mastering AspectJ*”, Wiley, March 2003
- [9] AspectJ, <http://www.eclipse.org/aspectj>, June 2004
- [10] AspectWerks, <http://aspectwerkz.codehaus.org/index.html>, June 2004
- [11] JBoss-AOP, <http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/jboss/aop>, June 2004
- [12] Nanning, <http://nanning.snipsnap.org/space/start>, June 2004
- [13] Hanenberg, S.; Unland, R.: *Parametric Introductions*, 2nd International Conference of Aspect-Oriented Software Development (AOSD), Boston, MA, March 17-21, ACM Press, 2003, pp. 80-89.
- [14] Sun: Java Data Object <http://java.sun.com/products/jdo>, June 2005
- [15] Sun: Java Platform Enterprise Edition <http://java.sun.com/j2ee>, June 2005
- [16] Fowler, M: *Refactoring*., Addison-Wesley Professional, June 1999
- [17] Hibernate.org.: *Hibernate*, June 2005
- [18] Spring Framework, <http://www.springframework.org> June 2005
- [19] Robert Hirschfeld. AspectS — Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, pp. 216-232, LNCS 2591, Springer, 2003
- [] Schmidmeier : Aspect oriented Programming with Python, ACCU, April 2005

- [21] Colyer, Clement, Harley: *Eclipse AspectJ*, Addison Wesley, 2004
- [22] Kiselev: *Aspect-Oriented Programming with AspectJ*, Sams,, Juli 2002
- [23] Miles: *AspectJ Cookbook*, O'Reilly & Associates, 2005
- [24] Hannemann, Kiczales. [Design Pattern Implementation in Java and AspectJ](#) , OOPSLA, November 2002
- [25] Schmidmeier: "Let the code look like the design", JAOS 2004
- [26] Kiczales: "Making the Code Look Like the Design", AOSD 2003

APPENDIX Aspect oriented programming with AspectJ

In addition to Java AspectJ provides a number of new language features which will be explained here in more detail: aspect, pointcut, advice and introductions. The intention of this section is to give people which are relatively new in the area of aspect-oriented programming a brief introduction into the programming language AspectJ.

1 A-1.1 Joinpoints

Joinpoints are these points in the program, where someone wants to add new code or replace existing code with new one, in order to simplify, improve or enhance the functionality, design performance, fault tolerance, etc. In AspectJ joinpoints are points in the execution flow like:

- calling
- or executing a method,
- executing an advice,
- reading or writing a variable,
- handling an exception,
- initializing an object
- or class.

2 Pointcuts

A *pointcut* selects a collection of join points. To specify pointcuts AspectJ provides a number of pointcut designator like `call`, `execution`, `adviceexecution`, `get`, `set`, `initialization`, `handler` which select a joinpoints based on the defined program flow. However these Joinpoints are not sufficient to select all relevant joinpoints successfully in all non trivial applications. One way to overcome this solution is the pointcut-method pattern presented in [4]. AspectJ and AOP languages with an even more sophisticated and powerful pointcut languages like [13] offer additional pointcuts to improve the usability and overcome drawbacks of the pointcut method pattern. Some are based on the lexical structure of the application, like `within` and `withincode`. Both select all joinpoints inside a class, package or a method. Some other define the joinpoints based on type of the caller, callee or the arguments. These pointcuts are called `this`, `target` and `args`.

Each pointcut can be combined using Boolean operations. For example the following pointcut has the name `callFooFromAToB`. It describes all join points where a call to the method `foo` of class B is performed within the lexical scope of A.

```
pointcut callFooFromAToB(): within(A) && call(void B.foo());
```

The pointcut consists of two pointcuts which are combined by the logical operator `&&`. The pointcut `within(A)` describes all join points in class A, `call(B.foo())` describes all join points where the method `foo` of class B is called. Named pointcuts (like pointcut `callFooFromAToB`) can itself be used in other pointcut definitions.

AspectJ distinguished between static and dynamic pointcuts. A static pointcut describes join points which can be determined by a static program analysis. In the example above all join points can be determined statically. On the other hand there are join points which cannot be statically determined. For example the following pointcut determines all join points where a message `foo` is sent from an instance of A to an instance of a subclass of B.

E2-17

```
pointcut dCallFoofromAToB(): this(A) && call(void *.foo())
&& target(B+);
```

In contrast to the previous pointcut definition it now depends on the participating objects whether a certain line of code represents a join points described by this pointcut or not. For example a call of `foo` in a superclass of `A` might now be a valid join points as long as the object is an instance of `A`. The pointcut `call(void *.foo())` now determines all call join points to all existing `foo` methods independent of in what classes a method of this signatures occurs.

Pointcuts permit to export parameters. For example the following pointcut binds the runtime object of the sending object which is of type `A` to the variable `a`.

```
pointcut boundA(A a): this(a) && call(void *.foo())
&& target(B+);
```

Often someone wants to select only joinpoints which happen in the callstack of a method, or an advice. These joinpoints can be selected by with the pointcuts `cflow` and `cflowbelow`. E.g. following sample selects all joinpoints which haben in the callstack of the call to a public method `foo` in any class.

```
cflowbelow(call public void *.foo())
```

Please note that `cflow` or `cflowbelow` pointcuts are nearly always used in combination with other pointcuts to limit the scope of the pointcut. The `if` pointcut is another popular way to limit a pointcut. The `if` pointcut requires a Boolean expression, which can be statically evaluated. AOP languages or frameworks, which do not have an `if` pointcut can easily use the pointcut method pattern to select the appropriate pointcut.

3 Type Patterns

AspectJ allows the use of type patterns whenever a single type or a type is required. Table :: contains an overview of the valid type patterns

<code>*</code>	A sequence of any characters, except <code>.</code>
<code>..</code>	A sequence of characters, starting and ending with a <code>.</code>
<code>+</code>	The type itself or any subtype
<code>A B</code>	Type pattern A or type pattern B
<code>A&&B</code>	Type pattern A and type pattern B
<code>!A</code>	Not type pattern A

4 A-1.2 Advice

A piece of advice specifies the code that is to be executed whenever a certain join point is reached. It represents the crosscutting code because it may be executed at several execution points in the program. The declaration of a piece advice needs to specify at what joinpoints, it is meant to be executed, in order to select these joinpoints it uses a single or a combined pointcut statement. Additionally, it must specify at what point in time it is supposed to be

E2-18

executed: an advice may either be executed before or after the original code at a certain joinpoint or may even replace it.

In AspectJ pointcut methods are defined as follows:

```
before() : aPointcut() { ...do something... }
```

This method is executed before a join point determined by the pointcut `aPointcut` is reached (the modifier `before()` is responsible for deciding, at which point of the interaction the method should be invoked). Furthermore an advice can be executed around or after a join point. An advice which is declared around an joinpoint, replaces the original code at this joinpoint. Most often the advice adds only some additional functionality to the original code, (e.g. some caching, some exception handling). AspectJ offers the keyword `proceed` for these scenarios. The keyword `proceed` executes the original code (or the next advice in the advice chain) at this pointcut.

A piece of advice can refer to pointcut parameters. For example the following piece of advice

```
before(A a) : boundA(a) {  
    System.out.println(a.toString());  
}
```

imports the pointcut parameter `a` of type `A` and sends in its body the message `toString`. Inside a piece of advice the keywords `thisJoinPoint` or `thisStaticJoinPoint` can be used which permit to reflect on the current join point.

5 A-1.3 Introductions

Introductions permit to add new members to classes (which is similar to open classes or mixins [3]) or to add new interfaces or superclasses to classes. Syntactically, introductions consists of the member definition and the name of the target type. To add new interfaces to a target type, AspectJ provides the keywords `declare parents`.

```
class A { ... }  
interface NewInterface {...}  
aspect MemberIntroduction {  
    public String A.newString;  
    public void A.doSomething(){...}  
}  
aspect InterfaceIntroduction {  
    declare parents: A implements NewInterface;  
}  
aspect TypePatternIntroduction {  
    public void (A+).doSomething2() {...}  
}
```

The code above contains an aspect `MemberIntroduction` that adds a field `newString` and a method `doSomething` to class `A`. The aspect `InterfaceIntroduction` adds the interface `NewInterface` to class `A`. The target type can be specified using so-called type patterns that permit to apply an introduction to several types at the same time.

6 A-1.4 Aspects

Aspects are class-like constructs which permit to contain all of the above mentioned constructs. In contrast to classes aspect cannot be instantiated by the developer. Instead, aspects define on its own how and when they are instantiated. Aspects can be declared abstract and abstract aspect can be extended (similar to the extends relationship in Java). Abstract aspects are not instantiated and their pieces of advice do not influence the base program. Similar to the member sharing in Java, pointcut definitions are shared along the inheritance hierarchy.

The instantiation of aspects is defined in each aspect's header. Aspects are either singletons, that means there exists only one instance of, or there are instantiated on a per-object basis. That means an aspect is instantiated for each object, which participates in a certain call-join point. e.g.

```
aspect MyAspect perthis(this(A)) {
    //pointcut definition
    // advice definition
    // introduction definition
}
```

The aspect above is instantiated for each instance of A matching the `this(A)` pointcut. By default an aspect is instantiated as singleton that means omitting "perthis(this(A))" in the example above means that there is exactly one instance of the aspect in the system. An Aspect can also be defined as an `perflow` or `perflowbelow` aspect. The runtime creates in this case a new aspect instance for each callstack which passes the pointcut definition inside the `perflow` or `perflowbelow` statement.

```
aspect MyAspect perflow(somepointcut()) {
    //pointcut definition
    // advice definition
    // introduction definition
}
```

Aspects can implement interfaces or extends classes or other aspects. Aspect Inheritance is a powerful feature which is often used in AOP idioms and patterns. For a discussion of several of them see [4].

Abstract aspects can define abstract methods, or abstract pointcuts besides its usual members. Advice can not be overwritten. Please note:

If an abstract advice is subclassed twice (as in following sample),

```
abstract aspect AbstractAspect {
    pointcut somepointcut():call(public * *.somemethod(..));
    before():somepointcut(){
        //someAdviceCode
    }
}

aspect Aspect1 extends AbstractAspect{
}

aspect Aspect2 extends AbstractAspect{
}
```

E2-20

```
}
```

two instances (Abstract1, and Abstract2) of the abstract aspect are created. Both contain the pointcut somepointcut and bind their advice to their pointcuts. This results in the fact that the call to somemethod gets advised twice.

Like classes Aspects can be declared in its own file, together with other classes and aspects in the same file, or even like inner classes as inner aspects. On the other side, aspects can contain inner classes and anonymous inner classes just like “normal classes”.

7 Accessing Aspects

Aspects are normally accessed from advices. The AOP runtime system is responsible to invoke the appropriate advices at the joinpoints selected by the pointcuts. It is very common that aspects interact like any object with OO-libraries and contain state like any object. Other objects or aspects might want to access this state. The aspect could store itself therefore at a well known location, e.g. in a directory service or a singleton, from where other objects or aspects could retrieve and use it. A more convenient alternative is the use of the static accessor function aspectOf, which every non abstract Aspects class does provide.

The aspectOf function returns the aspect, if an aspect has been instantiated and throws an exception otherwise. The aspectOf function requires an additional parameter of type Object for aspect classes declared as perthis or pertarget to differentiate which aspect is referenced. Singleton aspects or aspect declared as perflow or perflowbelow can be accessed without any additional parameter. The AspectJ runtime engine provides for these aspect types the current context implicit to the aspectOf method. You can get therefore the current aspect instance for an aspect declared as

```
aspect MyAspect perflow(somepointcut()){
```

with following call.

```
MyAspect.aspectOf();
```

8 A-1.5 HelloWorld in AspectJ

This section just presents a small AspectJ variation of the well-known Hello World example. The class BaseProgram represent the base program the aspect MyAspect is woven to. The base class just instantiates itself and calls its method sayHelloWorld. The aspect defines a pointcut on the call of this method and a corresponding piece of advice.

```
public class BaseProgram {
    public static void main(String[] args){
        BaseProgram base = new BaseProgram();
        base.sayHelloWorld();
    }
    public void sayHelloWorld (){
        System.out.println("Hello World");
    }
}
```

```
}
```

```
aspect MyAspect {  
    pointcut pc(BaseProgram b): this(b) &&  
        calls(void BaseProgram.sayHelloWorld);  
    void around(BaseProgram b): pc(b) {  
        System.out.println("An instance of "  
            + b.getClass().getName() + " invokes sayHelloWorld");  
        proceed(b);  
        System.out.println("invocation done...");  
    }  
}
```

When `sayHelloWorld` is invoked the `around` advice is executed instead, which first prints out some additional text which depends on the calling object. Then the originally method is executed using the special command `proceed()` which can be used inside `around` advice. After the original message is shown the advice finally prints out some additional text. The result of calling the main method is:

```
An instance of BaseProgram invokes sayHelloWorld  
Hello World  
invocation done...
```

However a plain AOP AspectJ-HelloWorld is also possible. Following code below is the shortest possible HelloWorld AOP solution:

```
public aspect HelloWorld {  
    before():execution(public static void  
        HelloWorld.main(String[])) {  
        System.out.println("Hello World");  
    }  
    public static void main(String[] args) {  
    }  
}
```