!IMPORTANT!

Some language elements described in this text are not yet supported in the current JAsCo version (0.8.x). These are:

- Multiple hook constructors
- perall, percflow, permethod and perclass. Perobject is supported!

Gotchas when migrating to 0.8.5 from an earlier JAsCo version:

- Terminology changed for compliance with AspectJ:
  - replace → around
  - execute → execution
  - calledobject → thisJoinPointObject
  - calledmethod → thisJoinPoint

TABLE OF CONTENTS

1 Introduction

JAsCo is an aspect-oriented extension to the Java language. The JAsCo approach is primarily based upon two existing AOSD approaches: AspectJ and Aspectual Components [LLM99]. AspectJ's main advantage is the expressiveness of its "pointcut"-language. The pointcut language describes the condition on which the corresponding aspect advice is executed. These conditions are able to contain a wealth of constructs ranging from lexical properties of the program to dynamic information as activation records on the stack. However, aspects are not reusable, since the context on which an aspect needs to be deployed is specified directly in the aspect-definition. To overcome this problem, Karl Lieberherr et al. introduce the concept of Aspectual Components. In Aspectual Components, aspects are specified independently as a set of abstract join points. Explicit connectors are then used to bind these abstract joinpoints with concrete joinpoints in the target application. This way, the aspect-behavior is kept separate from the base components, even at run-time. JAsCo combines the expressive pointcut declarations of AspectJ with the aspect independency idea of Aspectual Components. JAsCo does however restrict the joinpoints that are possible to the public interface of the components, meaning public methods and fired events.

The JAsCo language is kept as close as possible to the regular Java syntax and concepts. Only a minimal number of new keywords and constructs are introduced. The JAsCo language introduces two new concepts: *aspect beans* and *connectors*. An aspect bean is a regular Java bean that is able to declare one or more logically related hooks as a special kind of inner classes. Hooks are generic and reusable entities and can be considered as a combination of an abstract pointcut and advice. Because aspect beans are described independently from a specific context, they can be reused and applied upon a variety of components. Connectors have two main purposes: instantiating the abstract aspect beans onto a concrete context and thereby binding the abstract pointcuts with concrete pointcuts. In addition, a connector allows specifying precedence and combination strategies between the aspects and components.

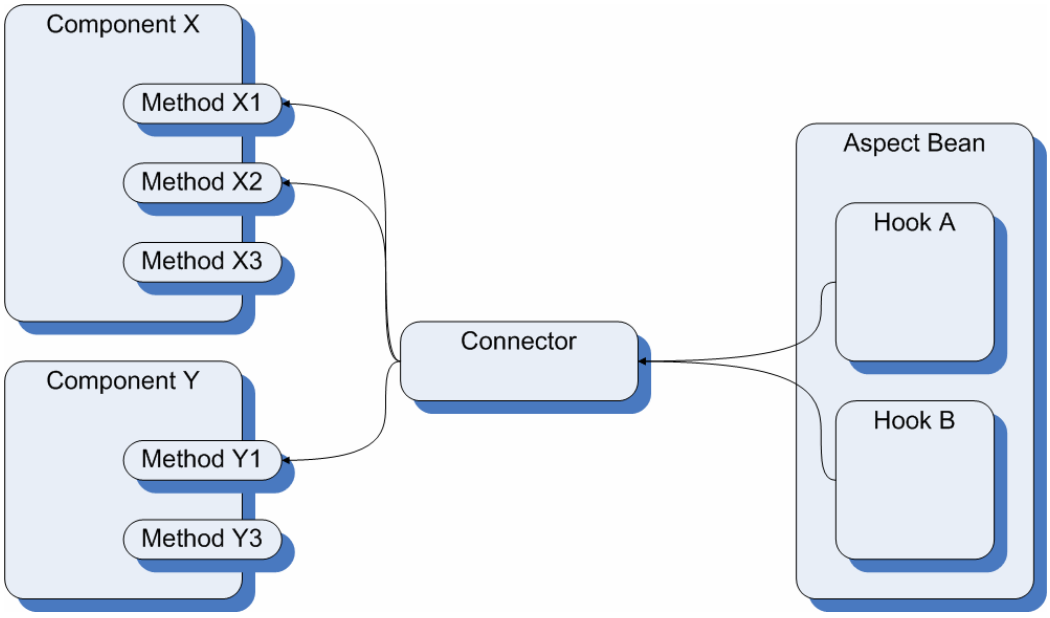

Figure 1: Schematic overview of JAsCo.

To make the JAsCo language operational, we introduce a new "aspect-enabled" component model. The JAsCo Beans component model is a backward compatible extension of the Java Beans component model

where traps are already built-in. These traps are used to attach and detach aspects. As a result, JAsCo beans do not require any adaptation whatsoever to be subject to aspect application. The JAsCo component model enables run-time aspect addition and removal. In addition, aspects remain first class entities at run-time as they are not weaved and spread into the target components.

Figure 1 gives a schematic overview of the JAsCo approach. At the left-hand side, two normal Java Bean components are depicted which declare a number of methods and some events. At the right-hand side an aspect bean is shown that contains two hooks. A connector is then used to instantiate the abstract hooks to the concrete methods and events of the target component. Notice that the aspect bean is also able to declare normal methods and events. As such, the aspect bean can also be employed as a regular Java bean.

The next section explains the different features the JAsCo-language has to offer. The syntax of aspects and connectors is discussed using a small example. Section 3 proposes an Adaptive Programming extension to the JAsCo language and section 4 introduces the stateful aspects extension.

2 Language

2.1 Introductory example

```
1  class AccessManager {
2
3     PermissionDb p_db = new PermissionDb();
4     User currentuser = null;
5     Vector listeners = new Vector();
6
7     boolean login(User user, String pass)  {
8        if(p_db.check_pass(user,pass)) {
9           currentuser = user;
10          return true;
11       }
12       else return false;
13    }
14    void logout() {
15       currentuser = null;
16    }
17
18    void addAccessManagerListener(AML listener) {
19       listeners.add(listener);
20    }
21    void removeAccesManagerListener(AML listener) {
22       listeners.remove(listener)
23    }
24    void fireAccessEvent(AccessEvent event) {
25       Iterator iter = listeners.iterator();
26       while(iter.hasNext()) {
27          AML listener = (AML) iter.next();
28          listener.accessEventOccured(event);
29       }
30    }
31
32    hook AccessControl {
33
34       AccessControl(method(..args)) {
35         execution(method);
36       }
37
38       around() {
39         if(p_db.check(currentuser))
40            return proceed();
41         else throw new AccessException();
42       }
43    }
44 }
```

When?

What?

Code Fragment 1: AccessManager aspect bean.

An example of a crosscutting concern is access control. Database systems for instance, need some control mechanism to manage user access to the data they hold. A similar concern could be stipulated in an ordinary application. Imagine a piece of software that runs on an operating system that allows only one user at the same time to be logged in. If users do not have the required permission, the access to some components of this system is denied.

Code Fragment 1 shows an access control aspect, specified using JAsCo. Notice that this aspect bean appears to be a normal Java Bean at first sight. Indeed, an aspect bean is able to declare normal fields and methods and even events (line 18-30 in Code Fragment 1). The crosscutting behavior itself is specified using the hook construct, which is a special kind of inner class. A hook specifies at least one constructor (line 34 till 36) and one advice (line 38 till 42), and is able to contain any number of ordinary Java class-members. A constructor of a hook specifies the condition on which the hook is triggered. When a hook is triggered, the hook advices are executed. Parameters of a hook constructor are abstract method parameters that are bound to concrete methods or events in a connector. The constructor body specifies the condition itself on which the hook is triggered using these abstract method parameters. The constructor (line 34) of the `AccessControl` hook specifies that this hook can be deployed on every method which takes zero or more arguments as input. The constructor body (line 35) specifies that the hook is triggered whenever the method or event which is bound to the abstract method parameter `method` is executed.

The advices of a hook are used to specify the various actions a hook needs to execute when hook is triggered. The `AccessControl` hook specifies only an around method (line 38 till 42), which substitutes the behavior of the method that is about to execute when the hook is triggered. The around method checks if the currently logged-in user has the proper access-permissions. This is done by querying the permissions database about the current user. When no problems are encountered, the original method is executed. Otherwise, an `AccessException` is thrown.

Deploying an aspect within an application is done by making use of connectors. Imagine that our application contains a printer component. Only people who have the appropriate print permissions, may access this component. Code Fragment 2 shows the connector that is used for instantiating the `AccessControl` hook upon the `Printer` component:

```
1   static connector PrintAccessControl {
2
3      AccessManager.AccessControl control =
4       new AccessManager.AccessControl(Printer.doJob(PrintJob));
5
6      control.around();
7   }
```

Code Fragment 2: Connector that instantiates the AccesControl hook onto the doJob method of the Printer component.

A connector contains three kinds of constructs: one or more hook-instantiations (line 3-4), zero or more advice executions (line 6), and finally any number of regular Java constructs. The `PrintAccessControl` connector contains one hook instantiation `control`, which instantiates the `AccessControl` hook onto the `doJob` method that is defined in the `Printer` component interface. As such, the abstract method parameter `method` of this instance of the `AccessControl` hook is bound to the `doJob` method of the `Printer` component. Afterwards, the execution of the around advice on the control hook is specified. To sum up, the connector of Code Fragment 2 declares that the around method of the `AccessControl` hook should be executed, whenever the `doJob` method of the `Printer` component is executed. As a result, invoking the `doJob` method of the `Printer` component is restricted to users who have the required permissions.

A connector is also able to instantiate a hook onto the firing of an event, which is the main method of communication of the Java Beans component model. As a result, when the event is fired, the hook is triggered. Section 2.3.1.3 explains this in more detail.

2.2 Aspect Bean Syntax and Informal Semantics

In this section, the aspect bean syntax is discussed in detail. An aspect bean is able to contain one or more hooks that contain the crosscutting behavior itself. A hook is a participant of an aspect, and is used for specifying:

- when the normal execution of a method of a component should be "cut".
- what extra behavior should be executed at that precise moment in time.

For specifying the "when" part, a hook constructor and the isApplicable method are used. The "what" part is implemented in hook advices.

2.2.1 Hook Constructor

A hook constructor is responsible for specifying in an abstract way "when" the hook's behavior should be triggered. In that sense, the hook constructor is similar to a kind of abstract pointcut. A hook constructor contains one or more abstract method parameters that denote the context where the hook can be instantiated upon. The abstract method parameters are bound to concrete methods or events when the hook is instantiated in a connector. Code Fragment 3 illustrates a hook constructor which declares three abstract method parameters. Method `m1` has to be bound to a method with two parameters: one of type `String` and one of type `int`. It is also possible to specify an abstract method parameter that can be bound by any method, using the `..` construct. Method `m2` in Code Fragment 3 is an example of this. The `..` construct can also be combined with specific types, but should always be the last argument type. For example, method `m3` specifies that it can be bound to any method that has as first argument type `PrintJob`[1]. For example, in Code Fragment 1, the `method` abstract method parameter is used in the around advice in order to call the original method bound to the `method` abstract method parameter.

The abstract method parameters are bound to concrete methods or events when the hook is instantiated in a connector. The binding mechanism is call-by-value, the abstract method parameter is bound to a reflective representation of the method. Call-by-name parameter passing is not applied in order to allow separate compilation. As JAsCo is an extension of Java, the scoping rules are also not altered. Also abstract method parameter invocations are lexically scoped in such a way that variable accesses in the method bound to the abstract method parameter are looked up in the lexical scope of the original method definition rather than in the lexical scope of the hook. For example, when `m1` in Code Fragment 3 is bound to a method `test` and the method body of `test` refers to a local variable `x`, then the `x` defined in the class that contains `test` is accessed and not an `x` defined in the aspect bean or hook. Notice that variable accesses inside the hook implementation (not the original implementation) will use the lexical scope of the hook to find the correct value.

The parameters of abstract method parameters are also qualified by parameter names in addition to types. This is because the parameters of an abstract method parameter are bound to the actual parameters that are passed to the method bound to the abstract method parameter. For example, the first parameter of method `m1` is named `s`. When the hook is triggered at run-time, the parameter `s` is bound to the first actual parameter passed to the method bound to `m1`. Parameters of abstract method parameters that are specified

[1] Notice that specifying application specific types like for instance PrintJob in a hook constructor, renders a hook less reusable. If possible, more generic or standard Java types are preferred.

using the .. construct, are an exception to this rule. These parameters are bound to an array of all the remaining actual parameters that are passed to the method bound to the abstract method parameter at hand. For example, abstract method parameter m3 specifies ..args2 as second parameter. The value of arg2 is then an array that contains all the actual parameters except the first actual parameter where the method bound to abstract method parameter m3 is executed with. Notice that there is no confusion possible when an abstract method parameter takes multiple arguments using the .. syntax and when the first actual argument is an array. In this case, the parameter contains an array of which the first element is the actual argument, which is again an array.

Parameters of abstract method parameters have a scope that extends all over the hook, they can thus be directly accessed in the advices. The parameter passing mechanism is call-by-value. As such, changing the parameter's value does not have an effect on the value of the original method argument. It is however possible to alter the arguments of which the method bound to the abstract method parameter is invoked with. See section 2.2.2 for a detailed explanation.

```
1  aHook(m1(String s, int i),m2(..args),m3(PrintJob pj, ..args2) {
2          execution(m1) && !(cflow(m2) || withincode(m3));
3  }
```

Code Fragment 3: An advanced hook constructor.

Specifying concrete types for parameters of abstract method parameters allows constraining the applicability of the hook to only methods of that signature. Attempting to bind the hook to an incompatible method signature in a connector results in a compile-time error. An incompatible method signature means either less or more parameters or types that are not equal to or not a subtype of the specified type. It is also possible to constrain the return type of an abstract method parameter. When no return type is specified, the abstract method parameter matches methods disregarding return type. When a return type is specified, the abstract method parameter can only be bound to methods that return that type or a subtype.

The constructor body specifies the triggering condition itself using the abstract method parameters. Every condition specification should at least specify the execution keyword[2]. This keyword states that the hook will be triggered each time the method bound to the abstract method parameter is executed. The execution keyword can be combined using the logical operators and (&&), not (!) and or (||) with three other keywords: cflow, withincode and target.

The cflow(x) expression evaluates to true when the currently executing method is in the control flow of the method bound to abstract method parameter x. In other words, this means that when starting the execution of the currently executing method, the method bound to abstract method parameter x is not yet finished. Consequently, an activation record corresponding to the method bound to abstract method parameter x is still on the stack.

The withincode(x) expression is more restrictive than cflow and obliges the method bound to x to be the direct caller of the currently executing method. In other words, an activation record corresponding to the method bound to x is the first activation record on the stack after the activation record of the currently executing method. For example, Code Fragment 3 specifies that aHook is triggered when the method bound to m1 executes when it is outside the control flow of the method bound to m2 and not directly called by the method bound to m3.

The `target(x)` expression allows to safely cast the target object of a method execution joinpoint (i.e. the object that executes the method) to a certain type `x`. When the target object is not of the defined type or a subtype of that type, the hook is not triggered. The passed type can be a full type name (including packages) or only the class name if the package or class is imported using the Java import syntax.

```
1  aHook(m1(String s, int i)) {
2          execution(m1) && target(String);
3  }
```

Code Fragment 4: Using Target.

Notice that it is possible to specify more than one hook constructor to allow one hook to be instantiated onto several different abstract contexts. This increases the reusability of a hook. When a hook is instantiated in a connector, the correct hook constructor is resolved.

2.2.2 Advices

In order to specify the "what" part of a hook, advices are used. At least one advice has to be specified, otherwise the hook does not do anything. Notice that this is not enforced in the current JAsCo implementation. Six kinds of advices are available: *before*, *around*, *after*, *after throwing*, *after returning, around throwing* and *around returning*. A before advice is able to specify behavior that should be executed before the original method that causes the hook to be triggered, is executed. Likewise, around and after advices specify behavior that executes respectively instead of and after the joinpoint where the hook is triggered. An after advice executes disregarding whether the original method returns successfully or throws an exception.

An after throwing advice executes after the original method, but only if an exception is thrown. The after throwing advice specifies one argument, namely the type of exception the advice is interested in. When a subtype of the specified type is thrown, the advice is also executed. As such, when specifying *Exception*, all normal exceptions are captured. Multiple after throwing advices for capturing different kind of exceptions can be declared. Similar to AspectJ, the after throwing advice always rethrows the exception. Code Fragment 5 illustrates an example after throwing advice that catches all exceptions and prints a log message.

```
1      after throwing (Exception ex) {
2         System.out.println("exception occurred: "+ex.getMessage());
3      }
```

Code Fragment 5: After throwing advice.

Similarly, an after returning advice executes after the original method, but only if it returns successfully. The after returning advice specifies one argument, namely the type of the return type the advice is interested in. When a subtype of the specified type is returned, the advice is also executed. As such, when specifying *String*, all objects of type String and subtypes of String are passed to the after returning advice. Multiple after returning advices for capturing different return types can be declared. Similar to AspectJ, the after returning advice cannot influence the actual returning, for that an around advice is required.

```
4      after returning (String s) {
5         System.out.println("exception occurred: "+s);
6      }
```

Code Fragment 6: After returning advice.

Around returning and around throwing are similar to after returning and after throwing respectively except that they allow altering the return value or decide to not rethrow the thrown exception. Code Fragment 7 illustrates an example around returning advice that applies a discount to a price that has been returned by a price calculation function.

```
1      around returning (int price) {
2         return price*discountPercentage;
3      }
```

Code Fragment 7: Around returning advice that allows altering the return value.

Code Fragment 8 illustrates an around throwing advice that captures the thrown exception and does not rethrow the exception. The around throwing advice might of course also decide to rethrow the exception if necessary. Als notice the invocation of proceed although the around advices have already executed. Proceed will always make sure to invoke the next advice or original method in the advice chain that has not yet executed. This allows for implementing a fallback mechanism that captures exceptions thrown by advices and makes sure the following advice in the chain can execute as well. When the original method has already been executed, the `ProceedFinishedException` exception is thrown. In order to invoke the advice chain all over again, `invokeAgain` method can be used. For more explanation, see the Java Doc for the thisJoinPoint keyword API.

```
1      around throwing (IOException ex) {
2         log("Exception captured");
3         return proceed();
4      }
```

Code Fragment 8: Around returning advice that allows altering the return value.

All other advices (except the after/around throwing and after/around returning advice) have no arguments. Before and after advices do not have return types, the return type of around advices is Object. Notice that this is not specified in an around advice specification, see for example Code Fragment 2 (line 20). It is also possible to return primitive types like for example `int`, the JAsCo system automatically boxes primitive types to their object representation. As explained in the previous section, advices are able to access abstract method parameters and parameters of abstract method parameters.

Before and after advices are not able to influence the original method execution. Around advices are however able to do this. They can choose whether to invoke the original behavior and are also able to pass different arguments. Likewise to AspectJ, this is achieved by specifying the "proceed" keyword. When no arguments are passed to proceed, the original method is executed with the original arguments. Code Fragment 9, line 2, illustrates an example of this by invoking the proceed method.The proceed method can however also be employed to invoke the original method with new arguments. The signature of proceed method is the same signature as the abstract method parameter passed to the execute or call pointcut designator.Code Fragment 9 line 1 shows an example where the method bound to `m1` is called with a newly specified string `"test"` and the original parameter `i`. Remember that parameters of abstract method parameters have a scope that extends all over the hook, so the argument `i` of `m1` is visible here.

```
5      proceed("test",i);
6      proceed();
```

Code Fragment 9: Invoking abstract method parameters of the hook constructor of Code Fragment 3.

2.2.3 Inheritance

Inheritance of both aspect beans and hooks is supported. As a hook is a special kind of inner class, the inheritance semantics is equivalent to inheritance of inner classes. An aspect bean that extends another aspect bean inherits all the hooks declared in the parent aspect bean. Of course, the child aspect bean can also declare new hooks. It is also possible to define a new hook with the same name as a hook in a parent aspect bean. In that case, the new hook replaces the hook of the parent aspect bean. Code Fragment 10 illustrates an extended version of the `AccessManager` aspect bean of Code Fragment 1. The hook `TracingAccessControl` extends the `AccesControl` hook and adds a before advice that prints out some tracing information every time an access attempt is made. The original advice and constructor of `AccessControl` are both inherited by the `TracingAccessControl` hook. As a result, when this hook is triggered, it first prints some tracing information and then verifies access.

```
1  class TracingAccessManager extends AccessManager {
2
3    hook TracingAccessControl extends AccessControl {
4
5      before() {
6        //print some tracing info
7      }
8    }
9 }
```

Code Fragment 10: Extended Access Manager aspect bean.

2.2.4 Other constructs

An aspect bean supports several other constructs that are useful in an aspect-oriented context. The following sections present:

- *thisJoinPointObject* keyword: a way to refer to the object corresponding to the currently executed joinpoint,
- *isApplicable* method: is able to define an additional hook triggering condition in plain Java,
- *refinable* methods: a mechanism to defer implementation of certain behavior to the connector,
- *global* keyword: allows to refer to the aspect bean context from a hook,
- Multiple methods in the same aspect bean,
- Abstract method parameter reflection API.

2.2.4.1 The thisJoinPointObject keyword

The critical reader might have noticed that the `AccessControl` hook of Code Fragment 1 checks access for given users globally without context information. In other words, for every part of the system where this instance of the `AccessControl` hook is applied to, the user is either granted access or refused. As a result, if one wants to realize a per component access policy, one has to instantiate an `AccessControl` hook with its own permissions database for each component in the system. This can be avoided when the component where the access attempt is made to is passed to the query in the permissions database. JAsCo allows referring to the object instance that executes the method that causes the hook to be triggered. This can be done using the `thisJoinPointObject` keyword. Code Fragment 11 illustrates how the `thisJoinPointObject` keyword could be used in the AccessControl example. The new around method checks admission both on user id and on the component that is accessed (line 2).

```
1       around() {
2         if(p_db.check(currentuser,thisJoinPointObject.getClass()))
3             return proceed();
4         else throw new AccessException();
5       }
```

Code Fragment 11: Using the thisJoinPointObject keyword to improve the AccessControl hook.

2.2.4.2 The isApplicable method

The `isApplicable` method is an extra construct that allows a more fine-grained control over when the hook should be triggered. Often, whether a hook is triggered or not depends on more than the conditions that can be specified in a constructor. For example, a hook implementing an anniversary discount business rule is only triggered when the current date is the client's birthday. The `isApplicable` method allows specifying these additional conditions. In absence of such a construct, these conditions have to be tested in all the aspect advices that are implemented. Furthermore, making this condition explicit by introducing a new keyword has the advantage that it allows more elaborated aspect combination strategies as the hook is only triggered when both the constructor body and the `isApplicable` method evaluate to true. In addition, introducing a new keyword allows optimizing this condition performance-wise in comparison to having it repeated in each hook advice. Code Fragment 12 illustrates an example of the `isApplicable` method. In this improved version of the `AccessControl` hook, the hook is only triggered when the user is not an administrator of the system because administrators are granted access to all components. When the hook is triggered, the around advice inherited from the parent hook is executed.

```
1  class ImprovedAccessManager extends AccessManager {
2
3    hook ImprovedAccessControl extends AccessControl {
4
5      isApplicable() {
6         return !p_db.isAdmin(currentuser);
7      }
8    }
9  }
```

Code Fragment 12: AccessControl with administration check using the isApplicable method.

2.2.4.3 Refinable methods

```
1    hook AccessControl {
2
3      AccessControl(method(..args)) {
4        execution(method);
5      }
6
7      around() {
8        if(p_db.check(currentuser)
9            return proceed();
10       else noAccessPolicy(currentuser);
11     }
12
13     refinable void noAccessPolicy(User user);
14
15   }
```

Code Fragment 13: Refinable method noAccessPolicy allows to customize the access policy in a connector.

It is possible to specify refinable methods in a hook by employing the *refinable* keyword. In this way, it is possible to specify highly customizable hooks where some part of the logic is only known when

instantiating the hook in a connector. As a result, one hook instantiated in different connectors can have varying behavior. Code Fragment 13 illustrates a new version of the *AccessControl* hook with a customizable policy for users that are denied access. A refinable method is introduced, named `noAccessPolicy` (line 13), that is called each time a user is rejected (line 10). This allows a connector to implement a specific rejection policy, for example blocking after three false attempts. Notice that it is also possible to provide refinable methods with an implementation by overriding the method in a child hook. Refinable methods also allow a kind of aspectual polymorphism [MO03] as the specific implementation of the behavior is able to vary over the concrete types contained in the context the aspect bean is applied to.

JAsCo supports specifying default behavior for refinable methods in a hook. This can be achieved by implementing the refinable method instead of omitting the implementation. See Code Fragment 14 for an example. The default behavior is only used when the refinable method is not provided with an implementation in a connector or when the refinable method is not implemented in a child hook. Of course, one might wonder why the refinable keyword is there, since it is also possible to have an implementation. Why not allow every method to be redefined in a connector? The reason is performance. Methods that can be re-implemented in a connector require extra logic to fetch the implementation from the connector. This logic relies heavily on reflection, which demands a significant run-time overhead.

```
1      refinable void noAccessPolicy(User user) {
2           throw new AccessException();
3      }
```

Code Fragment 14: Providing an refinable method with default behavior.

.

2.2.4.4 Referring to the Aspect Bean

Regular Java inner classes have a lexical scope that includes their defining Java class. As such, it is possible to execute methods of the outer class from within the inner class. At the moment, JAsCo hooks do not have a lexical scope that includes the aspect bean. This is mainly a technical limitation of the current JAsCo implementation; it is the intention to provide this in the future. In order to allow hooks to read fields and call methods of the enclosing aspect bean, the `global` keyword is introduced[3]. For example, suppose the no-access policy of Code Fragment 13 has to log out every user that attempts to access a part of system where the user has no access permissions for. This can be achieved by calling the logout method of the enclosing aspect bean as illustrated in Code Fragment 15. Notice that the `global` keyword is still useful when the JAsCo implementation is improved to fully implement hooks as inner classes. In Java inner classes, it is not possible to refer to the enclosing Java class in order to for example pass it to a method invocation. JAsCo is able to support this using the `global` keyword.

```
1      void noAccessPolicy(User user) {
2           global.logout(user);
3      }
```

Code Fragment 15: Using the global keyword for accessing the enclosing aspect bean in which the hook is defined in order to log out the user at hand.

[3] The global keyword is actually not very well chosen, because it gives the impression to refer to global entities like global variables, rather than to the enclosing aspect bean. Because JAsCo is already applied in practice in several projects, the global keyword is kept.

Hooks are meant to be semantically equivalent to Java inner classes. This means that the "this" keyword in a hook refers to the hook instance and not to the enclosing aspect bean instance.

2.2.4.5 Specifying Multiple hooks in the same Aspect Bean

It is also possible to specify multiple hooks in the same aspect bean. Hooks defined in the same aspect bean are able to share the enclosing aspect bean context. As such, data and behavior can be shared between those hooks. For example, the `AccessManager` aspect bean expects that the login method is invoked explicitly onto the aspect bean. If the application already has a user management system, the application has to be aware of the fact that an aspect is present in order to invoke the login method on the aspect bean. It is possible to avoid this by declaring another hook that is responsible for 'capturing' the current user. As such, this additional hook can be instantiated on the already existing user management system in order to fetch the current user. Code Fragment 16 illustrates the `CapturingAccessManager` aspect bean that declares in addition to the `AccessControl` hook, a second hook named `CaptureUser`. This hook implements an after advice that fetches the current user and stores this information in the aspect bean. This information, namely the current active user, is then used in the `AccessControl` hook for verifying whether the currently logged-in user has access to the system.

```
1   class CapturingAccessManager {
2
3      PermissionDb p_db = new PermissionDb();
4      User currentuser = null;
5
6      void setUser(User user) {
7
8          currentuser =user;
9      }
10
11     hook CaptureUser {
12
13        CaptureUser(method(User user)) {
14          execution(method);
15        }
16
17        after() {
18          global.setUser(user);
19        }
20     }
21
22     hook AccessControl {
23
24         AccessControl(method(..args)) {
25           execution(method);
26         }
27
28         around() {
29           if(global.p_db.check(global.currentuser))
30              return proceed();
31           else throw new AccessException();
32         }
33     }
34  }
```

Code Fragment 16: Specifying multiple hooks in the same aspect bean in order to capture the user at hand and to control access to the components in the system.

2.2.4.6 Reflection on the executed method

The executed method is available in the advices through the *thisJoinPoint* keyword. This keyword support a reflection API, which is a subset of the normal Java Method reflection API. The following methods can be invoked: (see the java doc for the full list of available methods)

- `String getName()` **:** Returns the name of the method bound to this abstract method parameter.
- `String getClassName()` **:** Returns the full class name (with packages) is which the method bound to this abstract method parameter is defined.
- `int getModifiers()` **:** Returns an integer that represents the modifiers (private,static,synchronized, etc…) of this method. Use java.lang.reflect.Modifier to query which modifiers this integer represents.
- `Object[] getArgumentsArray()` **:** Returns the values of the actual parameters where the method bound to the abstract method parameter has been originally called with.
- `Class[] getArgumentTypes ()` **:** Returns the argument types of the method bound to the abstract method parameter.
- `Object getCalledObject()` **:** Returns the object instance where the method that is bound to the abstract method parameter is invoked upon.
- `Object invokeJAsCoMethod() throws Exception` **:** This method invokes the method bound to the abstract method parameter using the arguments that were originally passed to this method. The result is the same as calling the abstract method parameter directly without supplying arguments (see Code Fragment 9, line 2).
- `Object invoke(Object obj, Object[] args) throws Exception` **:** This method invokes the method bound to the abstract method parameter and allows to change both the receiver and the arguments of the method. The `obj` parameter is the receiver of the method. The `args` parameter represents an array with actual arguments for the method. Notice that both the receiver type as the number and types of the arguments have to be correct, otherwise an exception is thrown.

Code Fragment 17, line 2, illustrates how the abstract method parameter reflection API can be used to implement a logging hook. In this before advice (line 7-10), the name and class name of the method bound to the abstract method parameter `method` are printed on the standard output. As a result, every time the hook is triggered, some tracing information about the currently executing method is printed.

```
1  hook SimpleLogging {
2
3    SimpleLogging(method(..args)) {
4       execution(method);
5    }
6
7    before() {
8      System.out.println("Executing"+calledmethod.getName()+
9        " IN "+ calledmethod.getClassName());
10   }
11 }
```

Code Fragment 17: Using the abstract method parameter reflection API to print detailed tracing information.

2.3 Connector Syntax and Informal Semantics

A connector is responsible for at one hand instantiating a set of logically related hooks onto a concrete context and at the other hand thoroughly specifying how the instantiated hooks co-operate. For the latter, both precedence as combination strategies are supported.

2.3.1 Instantiating Hooks

In order to deploy a hook onto a given concrete context, the instantiation concept of object-orientation is used. Every abstract method parameter a hook constructor defines has to be bound to a concrete method by specifying a method signature in the connector. It is also possible to specify import declarations in a connector. As such, the method signatures do not have to contain full package names. Code Fragment 18 illustrates how a hook that contains the constructor of Code Fragment 3 can be instantiated.

```
1   import wsml.printing.*;
2
3   static connector TestConnector {
4      AnAspectBean.aHook testHook = new AnAspectBean.aHook(
5           void Classx.method1(java.lang.String, int),
6           int Classy.method2(float),
7           Object Printer.print(PrintJob)
8      );
9   }
```

Code Fragment 18: Instantiating a hook that specifies the constructor of Code Fragment 3.

2.3.1.1 Instantiating Hooks on multiple methods.

It is also possible to instantiate the same hook on several concrete methods at the same time. This can be achieved by specifying multiple method signatures delimited by commas and grouped by braces for a single abstract method parameter. In Code Fragment 19, the `AccessControl` hook Code Fragment 1 is instantiated on both the `startJuggling` and `stopJuggling` methods of the `Juggler` bean. There is an important difference when instantiating a hook on multiple methods using only one expression in comparison to employing several instantiation expressions: the former generates a single hook instance for all the methods while the latter generates separate instances. Instantiating a hook on multiple concrete methods as in Code Fragment 19 causes the same hook to be triggered for all the concrete methods. Instantiating a hook on a single method at a time causes a different hook instance to be triggered for each method.

```
1   static connector MultipleConnector {
2
3      AM.AccessControl control = new AM.AccessControl({
4         void sunw.demo.juggler.Juggler.startJuggling(),
5         void sunw.demo.juggler.Juggler.stopJuggling()
6      });
7   }
```

Code Fragment 19: Instantitaing the AccessControl hook on several methods.

2.3.1.2 Instantiating Hooks using wildcards.

JAsCo also supports wildcard expressions for specifying the method signatures in a hook instantiation. Both the star (`*`) and the question mark (`?`) wildcard expression are supported. The star wildcard expression matches with any finite number of characters (including none) while the question mark matches with only a single character. Characters are here all the possible characters that can be used in the Java

language. Instantiating the `AccessControl` hook on both the `startJuggling` and `stopJuggling` methods of the `Juggler` bean can thus also be realized using wildcards as depicted in Code Fragment 20. Notice that the example wildcard expression of Code Fragment 20 also matches the following method: `sunw.demo.juggler.JuggleD.Juggling()`. Of course, it is also possible to combine wildcard expressions with the multiple concrete methods construct of the previous paragraph.

```
1   static connector WildcardConnector {
2
3      AM.AccessControl control = new AM.AccessControl(
4         void sunw.demo.juggler.Juggle?.*Juggling()
5      );
6   }
```

Code Fragment 20: Using wildcard expressions to instantiate a hook on several methods.

2.3.1.3 Instantiating Hooks on events

In the Java Beans component model, components receive requests or information from other components by regular method invocations. For sending information or requests however, components fire events. JAsCo is able to intercept these events, by instantiating a hook with the `onevent` keyword. For example, if the `Printer` component throws an event when it finishes printing a document, this event can be intercepted by a hook in order to execute some appropriate action. Code Fragment 21 shows the instantiation of a logging hook which writes some statistics to file, when a printing job has finished. Currently JAsCo does not explicitly support vetoable events. It is possible to instantiate a hook on the firing of a vetoable event, but not on the acceptance or rejectance of such an event.

```
1  static connector EventConnector {
2
3     Logging.FileLogger logger = new
4        Logging.FileLogger(
5            onevent Printer.jobFinished(PrintEvent)
6        );
7  }
```

Code Fragment 21: Instantiating a hook on the firing of the jobFinished event of the Printer component.

2.3.1.4 Implementing refinable hook methods.

```
1   static connector ImplRefinableConnector {
2
3      AM.AccessControl control = new AM.AccessControl(* *.*(*)) {
4        void noAccessPolicy(User user) {
5               String str = "False attempt by :"+user.getName();
6               logger.setOuput(logger.ASCII);
7               logger.logToFile(str);
8               wsml.security.SecurityManager.banUser(user);
9        }
10     };
11  }
```

Code Fragment 22: Implementing the noAccessPolicy refinable method in a connector.

Refinable methods in a hook need to be provided with an implementation in a connector. The syntax for specifying a refinable method implementation is very similar to the syntax used for anonymous classes in Java. For example, suppose the no-access policy in the hook of Code Fragment 13 has to be customized to log every failed access attempt to a file and to ban in the future every user that tries to access a part of the system where she/he has no access permissions for. Code Fragment 22 illustrates how this is implemented

in JAsCo. The AccessControl hook is instantiated on all possible methods and events using a wildcard expression. Line 4 specifies the method signature of the refinable method that is implemented. Line 5 constructs the string that has to be logged by querying the username. Afterwards, the logger is initialized to log ASCII output (line 6) and the logged string is send to the logger (line 7). The last line (line 8) invokes the *banUser* method on the *SecurityManager* in order to ban the current user in the future. Notice that it is also possible to implement multiple abstract method parameters in a single hook instantiation expression.

2.3.1.5 Instantiating Hooks on annotations

From Java 1.5 onwards, Java supports attaching Meta-Data to methods, fields and classes. JAsCo supports instantiating aspects on only those methods that define a certain annotation. This can be achieved by placing the name of the annotation before a method signature. Either place a full name including packages or import the packages and specify only the annotation name. Suppose we want to check for permissions on all methods in the system that are marked using the @Sensitive annotation:

```
1  static connector AnnotationConnector {
2
3      AM.AccessControl control = new AM.AccessControl(
4        @security.Sensitive void * *.*(*));
5  }
```

Code Fragment 23: Instantiating a hook on Java 1,5 meta-data (annotations).

2.3.1.6 Instantiating Hooks on all subtypes of a given type

JAsCo supports instantiating hooks to all subtypes of a given class or interface. This is done by adding the plus operator to the end of a method signature. The following code fragment shows an example where a hook is instantiated to all classes that implement the ISecure interface. When specifying a concrete method name instead of a wildcard, the hook is only instantiated to that method of all subtypes of the given class or interface. Note that in case the superclass is a class and not an interface, the hook will be triggered on that class as well as on all subclasses.

```
6  static connector SubtypeConnector {
7
8      AM.AccessControl control = new AM.AccessControl(
9        @security.Sensitive void * ISecure.*+(*));
10 }
```

Code Fragment 24: Instantiating a hook on Java 1,5 meta-data (annotations).

2.3.1.7 Instantiating hooks on other hooks.

The previous example is in fact a very bad example because it hard codes the crosscutting logging concern in the connector. It is better to modularize the logging concern in a reusable hook and to instantiate it onto the AccessControl hook in a connector. Applying aspects on aspects is hardly ever supported in current aspect-oriented approaches (notable exception is EAOP [DFS02]). JAsCo however, does support aspects that interfere with other aspects. Code Fragment 25 shows how a logging hook can be applied onto the AccessControl hook. First the AccessControl hook is instantiated and the noAccessPolicy refinable method is implemented to ban the user when a false access attempt is made (line 3-7). Afterwards, the FileLogger hook is instantiated onto the noAccessPolicy refinable method of the AccessControl hook (line 9-10). As a result, when the noAccessPolicy method of the AccessControl hook is invoked, the FileLogger hook is triggered and some logging information is saved to file. Afterwards, the normal execution of the noAccessPolicy method

resumes and the user is banned. Consequently, this connector has the same result as the connector specified in Code Fragment 22. The logging concern is however better modularized in comparison to Code Fragment 22.

```
1   static connector HooksOnHooksConnector {
2
3     AM.AccessControl control = new AM.AccessControl(* sy*.*(*)) {
4        void noAccessPolicy(User user) {
5               wsml.security.SecurityManager.banUser(user);
6        }
7     };
8
9     Logger.FileLogger logger = new
10        Logger.FileLogger(AM.AccessControl.noAccessPolicy(User));
11  }
```

Code Fragment 25: Applying the FileLogger hook onto the AccessControl hook.

It is crucial to be extra careful when instantiating hooks onto other hooks because this can easily lead to an infinite loop of two hooks causing each other to be triggered. Indeed, if the `AccessControl` hook is applied on the whole system as in Code Fragment 22, it would also affect the `FileLogger` hook. As a result, when the `noAccessPolicy` method is invoked by the `AccessControl` hook, the `FileLogger` hook is triggered. But when the `FileLogger` hook is triggered the `AccessControl` hook is at its turn triggered because it is applied on whole the system. Afterwards, the `AccessControl` hook invokes again the `noAccessPolicy` method and the application is stuck in an infinite loop. The current JAsCo implementation does not check for possible infinite loops at run-time. It is possible to check for potential infinite loops but the performance penalty will be high. The developer can however easily avoid an infinite loop caused by hook-on-hook instantiation by explicitly specifying that one of the hooks should not be triggered by the other hook. In Code Fragment 25 for example, the `AccessControl` hook could use an extended constructor as depicted in Code Fragment 26. The second abstract method parameter has to be bound to the methods of the `FileLogger` hook in order to avoid an infinite loop.

```
1      AccessControl(method(..args), excluded(..args)) {
2        execution(method) && !cflow(excluded);
3      }
```

Code Fragment 26: Extending the AccessControl hook constructor to exclude certain parts of the system, notably other aspects.

2.3.1.8 Automatically creating multiple hook instances

Normally, when a hook is instantiated on several points of the application in one expression, there is only one hook instance created for all those points. If one wants to have different hook instances for different points in the application, one has to specify separate expressions for every point in the application where a unique hook instance is required. JAsCo also allows to automatically create multiple hook instances in a single hook instantiation expression. Currently, five constructs are supported `perall`, `perobject`, `perclass`, `permethod` and `percflow`. `Perall` creates a new hook instance each time the hook is triggered. This is useful when the hook has to perform some calculations and thereby store partial results that are not needed afterwards. `Perobject` generates one instance of the corresponding hook for each object instance where the currently executing method is invoked upon when the hook is triggered. In other words, for every matching object instance in the system, there is a separate instance of the hook. `Perclass` is similar to `perobject`, but then for classes. `Perclass` generates a new hook instance for every class type where the hook is triggered upon. `Permethod` means that there is a separate and unique

instance for each method that causes the hook to be triggered when the method is executed. `Percflow` is similar to the `percflow` construct in AspectJ and instantiates a separate hook for every control flow through a joinpoint where the hook is triggered. Code Fragment 27 illustrates how `perobject` is used to instantiate a separate instance of the access control hook for every Juggler bean instance present in the application. `Permethod`, `perclass`, `perall` and `percflow` are specified in the same way as `perobject` in Code Fragment 27.

```
1   static connector PerobjectConnector {
2
3      perobject AM.AccessControl control = new AM.AccessControl(
4         void sunw.demo.juggler.Juggler.startJuggling()
5      );
6   }
```

Code Fragment 27: Automatically instantiating several instances of the same hook using the perobject keyword.

Several of the five per–keywords can be combined to realize an even more customizable hook instantiation. At the moment, the following combinations are supported: `perobject` with `permethod` or `percflow`; and `perclass` with `percflow`. The result of combining them is in fact the union of all the instances that would be normally generated when the keywords are applied separately. As a result combining `perall` with any other keyword does not make a lot of sense because this results in fact again in a `perall` behavior. Similarly, combining `permethod` and `perclass` generates the same as `permethod` alone. Code Fragment 28 shows an example of the combination of `perobject` and `permethod`. The result is that a new `AccessControl` instance is created for each method in each instance of the `Juggler` component. Calling for example the `startJuggling` method twice on the same `Juggler` instance causes the same unique hook instance to be triggered. However, calling this method on another `Juggler` instance or calling the `stopJuggling` method on the first `Juggler` component results in another hook being triggered.

```
1   perobject permethod AM.AccessControl c = new AM.AccessControl(
2       void sunw.demo.juggler.Juggler.*()
3   );
```

Code Fragment 28: Combining the perobject and permethod keywords.

Notice that one should be cautious when using any of the five per-keywords because this potentially generates a lot of hook instances. As a result, these keywords require a lot of resources and slow down an application significantly!

2.3.1.9 Instantiating multiple hooks of the same aspect bean

It is possible to define more than one hook in a single aspect bean. If two or more of these hooks are instantiated in the same connector, they share the same aspect bean instance. As a result, changing data defined in the enclosing aspect bean in one hook is reflected in all the other hooks specified in that aspect bean. This is also the case when multiple instances of the same hook are created either manually or by employing the special keywords `perall`, `perobject`, `perclass`, `permethod` or `percflow` in a connector. Notice that if two hooks are instantiated in different connectors, the hooks each refer to a separate instance of the same aspect bean.

2.3.2 Executing Hook Advices

Apart from specifying on which concrete points in the application a hook should be instantiated, a connector also allows to specify explicitly which advices have to be executed when the hook is triggered. In addition to the before, around and after advices, an extra implicit advice, called default, can be employed. Specifying the default advice causes the execution of all the advices that are specified in the hook. For example, specifying the around advice of the `AccessControl` hook as in Code Fragment 2 or specifying the default advice as in Code Fragment 29 results in the same behavior, namely the around advice is executed every time the hook is triggered. It is also possible to omit any advice execution specification, the default advice is then implicitly assumed.

```
1   static connector PrintAccessControl {
2
3      AccessManager.AccessControl control =
4       new AccessManager.AccessControl(Printer.doJob(PrintJob));
5
6      control.default();
7   }
```

Code Fragment 29: Default advice to execute the implemented advices of a hook.

2.3.3 Precedence Strategies

A double motivation exists for permitting to specify advices in the connector. The first advantage is that it enables advanced users of an aspect to tightly control the execution of the aspect behavior. The default method on the other hand provides an easy way for deploying an aspect within an application, without needing any knowledge about how the aspect behavior is executed. The second advantage of this approach is that it provides a partial solution for *the feature interaction* problem [PSC+01]. When multiple aspects are applied upon the same joinpoint of an application, some way is needed to order the execution of their behaviors. JAsCo partly addresses this open issue in AOSD by the specification of the advice executions in the connector. All advices of hooks that are triggered, are executed in the sequence specified in the connecter. If no explicit sequence is specified, they are executed in the order the hooks are instantiated. This means that all before methods are first executed in the sequence the corresponding hooks are instantiated. Afterwards, all around advices are executed in the sequence the corresponding hooks are instantiated. Finally, all after methods are executed in the same sequence. The precedence of the advices is enforced at run-time. If only a subset of the instantiated hooks are triggered, the sequence specified in the connector is still used for that subset. Advices of other hooks are discarded. Notice that it is impossible to specify more than one type of advice in the same hook. In the current JAsCo implementation, two before methods for instance can not be implemented in the same hook. Executing the same type of advice for a given hook instance, results in a compile-time error.

```
1   static connector PrintAccessControl {
2
3      AccessManager.AccessControl control =
4           new AccessManager.AccessControl(Printer.*(*));
5
6      Logger.FileLogger logger =
7           new Logger.FileLogger(Printer.*(*));
8
9      LockingManager.LockControl lock =
10          new LockingManager.LockControl (Printer.*(*));
11
12     logger.before();
13     lock.before();
14     control.around();
15     lock.after();
16     logger.after();
17  }
```

Code Fragment 30: Explicitly specifying precedence of several hooks.

For example, imagine that only one user at the same time may address the `Printer` component of the example in Code Fragment 2, and that the access to the printer still needs to be managed. In addition, some logging needs to be performed before a lock is requested and after the lock is released. Code Fragment 30 illustrates how the logging, access control and locking aspects can be instantiated simultaneously on the `Printer` component. JAsCo allows arranging the execution of the aspect behaviors by specifying the order in the connector body. Whenever mutual join points of the `control`, `logger` and `lock` hook instances are encountered, the sequence in which the advices are specified in the connector is enforced. In the case of Code Fragment 30, first some logging is activated by invoking the before advice on the `logger` hook (line 12). Afterwards, the access to the `Printer` component is locked, by calling the *before* advice on the `lock` *hook* (line 13), so that no other user can access it. Next, the `AccessManager` checks if the user has the correct permissions to use the printer (line 14). Afterwards, the lock is freed (line 15), by calling the *after* advice, so that other users can access the `Printer` component again. Finally, some logging is executed again (line 16).

Notice that it is possible to write a connector where an after advice execution is specified earlier than a before advice execution. This does not result in the execution of the before advice after the after advice, which would be very counterintuitive. Instead, the before advices are executed always before the execution of arounds and afters, but in the sequence specified in the connector. For example, the sequence specified in Code Fragment 31 results in fact in the same behavior as the sequence in Code Fragment 30.

```
1      logger.before();
2      lock.after();
3      logger.after();
4      lock.before();
5      control.around();
```

Code Fragment 31: Specifying before and after advices in a weird sequence.

In comparison to AspectJ, JAsCo allows a more fine-grained control on the order in which aspects should be executed. In Code Fragment 30 for example, the sequence of the logger and locking hooks differs for the before and after advices. This is not possible in most aspect-oriented approaches. In addition, JAsCo allows the precedence of aspects to vary over different applications as this is not hard coded into the aspects, but specified in the connectors.

```
1   static connector AroundConnector {
2
3      Hook1 hook1 = …
4      Hook2 hook2 = …
5      Hook3 hook3 = …
6
7      hook1.around();
8      hook2.around();
9      hook3.around();
10  }
```

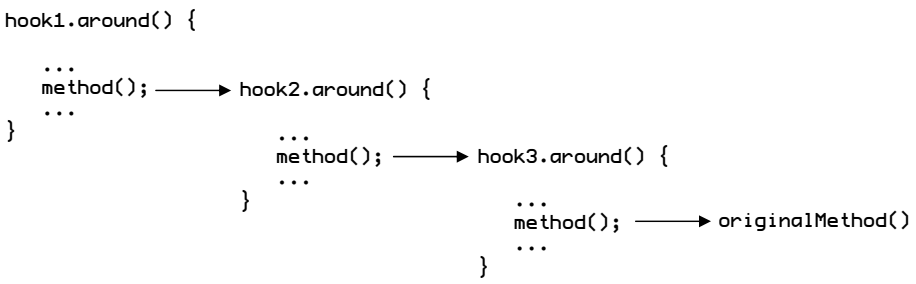

Figure 2: Chaining several replace advices.

In the examples presented in the previous paragraphs, there is always a single around advice execution specified. When multiple around advices are employed on the same joinpoint, the AspectJ around chaining approach is employed. This means that the firstly specified around method is executed first. If this around advice invokes the original behavior, the second around advice is executed instead of the replaced behavior and so on. When the last around advice invokes the original behavior, then the original method is executed. Of course, this chain can be broken at any time when one of the around advices does not call the replaced behavior. Figure 2 illustrates an example of chaining several around advices. If `hook1` calls the original behavior in its around advice, the around advice of `hook2` is executed instead. Likewise, if `hook2` calls the original behavior in its around advice, the around advice of `hook3` is executed instead. The invocation of the original behavior in the last hook however, named `hook3`, does result in the execution of the original behavior.

2.3.4 Combination Strategies

Precedence strategies are a solution to some feature interaction problems, however other combinations of aspects require a more expressive way of declaring how they cooperate. For example, one might have to specify that when aspect A is triggered, aspect B can not be triggered. This problem could be solved by introducing an extra connector keyword *excludes* which specifies that aspect A excludes aspect B. However, other aspect combinations require additional keywords and it seems impossible to be able to define all possible combinations in advance. That's why we propose a more flexible and extensible system that allows to define a combination strategy using regular Java. A `CombinationStrategy` interface is introduced (see Code Fragment 32) that needs to be implemented by each concrete combination strategy. A JAsCo `CombinationStrategy` works like a filter on the list of hooks that are applicable at a certain

point in the execution. Applicable hooks are hooks where the constructor and `isApplicable` method both evaluate to true.

```
1   interface CombinationStrategy {
2             public HookList validateCombinations(Hooklist);
3   }
```

Code Fragment 32: The CombinationStrategy interface.

The combination strategy interface can be implemented to realize the desired cooperation behavior. Each combination strategy needs to implement the `validateCombinations`-method, which filters the list of applicable hooks and possible modifies the behavior of individual hooks. For example, the combination strategy of Code Fragment 33 specifies an exclusion strategy. The combination strategy is initialized with two hooks in its constructor. The `validateCombinations` method specifies that the second hook is removed from the list of hooks that have to be triggered, whenever the first hook is present. As such, the second hook is not included in the hooks that are triggered and the advices of the second hook are not executed whenever the first hook is present.

```
1   class ExcludeCombinationStrategy implements CombinationStrategy
2   {
3       private Object hookA, hookB;
4       ExcludeCombinationStrategy(Object a,Object b) {
5            hookA = a;
6            hookB = b;
7       }
8
9       HookList validateCombinations(Hooklist hlist) {
10
11           if (hlist.contains(hookA)) {
12                hlist.remove(hookB);
13           return hlist;
14      }
15  }
```

Code Fragment 33: Implementing an exclusion combination strategy.

The exclusion combination strategy has been used effectively in the context of business rules that are implemented using aspect-oriented technologies [CDS+03]. For example, take an online shop where a business rule is specified that gives a discount of 20% to frequent customers. In addition, another discount business rule specifies that a user receives a 5% discount on his/her birthday. The business policy is that the frequent customer discount may not be cumulated with other promotions. As a result, when the frequent customer business rule is applicable, the birthday discount can not be applied. This collaboration can be implemented using the `ExcludeCombinationStrategy` of Code Fragment 33 as illustrated in the connector of Code Fragment 34. This connector instantiates two hooks on the checkout method of the online shop: one birthday discount hook (line 3) and one frequent customer hook (line 6). Afterwards, the behaviour methods of the hooks are specified (line 9-10). Finally, the exclude combination strategy is instantiated on both the frequent customer and the birthday hooks and added to the connector using the `addCombinationStrategy` keyword. The resulting application does not trigger the birthday discount hook when the frequent customer business rule is triggered. Notice that both hooks have to specify their triggering condition (frequent customer and birthday) in the `isApplicable` method (see section 2.2.4.2). Otherwise, the frequent customer hook would be always applicable and as a consequence the birthday discount is never executed.

```
1   static connector DiscountConnector {
2
3       Discounts.BirthdayDiscount birthday =
4            Discounts.BirthdayDiscount(* shop.checkout(*));
5
6       Discounts.FrequentDiscount frequent =
7            Discounts.FrequentDiscount(* shop.checkout(*));
8
9       birthday.around();
10      frequentcustomer.around();
11
12      addCombinationStrategy(
13         new ExcludeCombinationStrategy(frequent, birthday));
14  }
```

Code Fragment 34: Using the exclusion combination strategy to specify a discount combination.

It is also possible to define multiple combination strategies in the same connector. All these combination strategies are then merged using an approach similar to UNIX pipes. The sequence in which they are specified corresponds to the order in which they are employed in the pipeline. The first combination strategy receives the list of applicable hooks and filters them. The second combination strategy then receives this filtered list, performs its own filtering logic and passes the result on to the next combination strategy and so on. The hook list returned by the last combination strategy is then the list of hooks that have to be triggered at the current joinpoint.

```
1       HookList validateCombinations(Hooklist hlist) {
2
3            if (hlist.contains(hookA) && hlist.contains(hookB)) {
4                 hookB.setDiscount(hookB.getDiscoount() * 2);
5            return hlist;
6       }
```

Code Fragment 35: Adapting hook behavior depending on dynamic conditions using a combination strategy.

Normally, combination strategies function as a kind of filter on the list of applicable hooks at a certain joinpoint. However, combination strategies can also be employed to change some properties of the hooks depending on dynamic conditions. For example, suppose the birthday discount is meant to be increased instead of being not applicable when the user is a frequent customer. This can be easily achieved by the combination strategy of Code Fragment 35. In this `validateCombinations` method, the discount of `hookB` is doubled whenever the `hookA` is present in the list of applicable hooks at this joinpoint. Notice that when a hook is instantiated using one of the per-keywords (see section 2.3.1.8), multiple hook instances correspond to the same variable in a connector. However, only a single hook instance is able to be applicable at each joinpoint. As such, combination strategies only work with that single hook instance. When properties of a hook are changed as in Code Fragment 35, this property is only changed in the single hook instance that is applicable at the current joinpoint.

2.3.5 Other constructs

A connector supports several other constructs that are useful in an aspect-oriented context. The following sections present:

- *static* keyword: allows to instantiate aspects on instance or class level,
- regular Java code in a connector,

- the run-time connector reflection API .
- Connector combination strategies.

2.3.5.1 The static keyword

The attentive reader might have noticed that all connectors in the previous examples are preceded by the keyword `static`. This keyword states that the connector has to be applied on class level meaning that a hook is triggered regardless of the concrete object instance that executes the triggering method. Normally, this is the desired behavior for hooks that are instantiated in the connector. However, a connector can also be declared non-static, by omitting the `static` keyword. This causes the connector to only trigger hooks on those object instances that are explicitly registered with the connector at run-time. As a result, if the application does not register a single object instance, none of the hooks are able to be triggered, even if they are applicable on that context. In order to register object instances to a connector, the run-time connector API has to be used. See section 2.3.5.4 below for a more detailed explanation.

2.3.5.2 Connector priority

For defining the priority of a connector, an annotation (Java 1.5 meta-data) can be used. Simply define the following annotation before all other connector modifiers:

```
@jasco.runtime.connector.Prioriy(value=x) static connector Test {
      ...
}
```

Notice that Java that this only works for Java 1.5. See connector API (2.3.5.4) for dynamically altering connector priority.

2.3.5.3 Java Code in a connector

It is also possible to include regular Java code in a connector in order to perform more elaborated calculations. For example, if the discount hooks of Code Fragment 34 have to be initialized with a discount, this can be specified in a connector as illustrated in Code Fragment 36. This connector invokes the `setDiscount` method (line 9-10) on both the birthday and frequent customer discount hooks to initialize the hooks with a concrete discount.

```
1   static connector DiscountConnector {
2
3       Discounts.BirthdayDiscount birthday =
4            Discounts.BirthdayDiscount(* shop.checkout(*));
5
6       Discounts.FrequentDiscount frequent =
7            Discounts.FrequentDiscount(* shop.checkout(*));
8
9       birthday.setDiscount(0.02);
10      frequent.setDiscount(0.20);
11      ...
```

Code Fragment 36: Initializing hooks in a connector.

In case the connector instantiates a hook with one of the per-keywords (see section 2.3.1.8), the hook instance variable does not refer to a single hook, but to a group of hooks. As a result, invoking a method on this variable invokes this method in fact on the whole group. For example, if the `birthday` discount of Code Fragment 36 would be instantiated using the `perobject` keyword, then the `setDiscount` method is invoked on every instance that is generated because of the perobject keyword.

Specifying other Java code apart from initializations of hooks and combination strategies is discouraged in the current JAsCo approach. The reason is that a connector should be solely used for instantiating hooks and specifying how they cooperate. All other logic has to be modularized in normal java beans and aspect beans!

2.3.5.4 Run-time connector API

JAsCo provides a standardized API for each connector at run-time. This API supports limited reflection like for example querying which hooks the connector has instantiated. In addition, two properties of the connector can be adapted: whether the connector is static or not and whether the connector is enabled or not. The following paragraphs explain the run-time connector API in more detail.

- `static Connector getConnector()` : A connector is implemented using the singleton pattern because only one instance for each connector is needed per application. This method retrieves the unique instance of the connector at hand where all the methods discussed in the following paragraphs can be invoked on.
- `void addInstance(Object object)` : This method registers an object instance to the connector. If the connector is non-static, it only triggers hooks on instances that are registered to the connector in this way. When the connector is static, it already triggers on all possible object instances and as a result this method has no direct observable effect. Obviously, the connector copes with this object instance when it is afterwards changed to non-static modus.
- `void removeInstance(Object object)` : This method is the counterpart of the method discussed in the pervious paragraph and removes object instances from the set of objects where the connector triggers hooks on in non-static mode.
- `Iterator getInstances()` : This method retrieves all object instances that are registered with the connector. Even when a connector is static, this method returns the list of currently registered object instances, although they are of no use as long as the connector is in static mode.
- `Iterator getHooks()` : This method allows to query all hooks that are instantiated by the connector. When the connector instantiates hooks using one of the per-keywords (see section 2.3.1.8), all the generated hook instances end up in the returned collection.
- `Iterator getMethods()` : Returns the list of methods in textual form where the hooks in the connector are instantiated onto. Methods that are specified using wildcards expressions are returned as such and not as a list of all the possible methods that match with the pattern.
- `void setEnabled(boolean bol)` : This method allows to enable and disable a connector at run-time. A disabled connector freezes all its hooks so that they are not triggered any longer. Although it is possible to compile and load connectors at run-time, this requires quite some overhead due to the compilation process. When the same connector has to be activated and de-activated often during the execution of the application at hand, it is better to enable and disable the connector using this method rather than compiling and removing the connector using the command-line tools.
- `boolean isEnabled()` : This method queries whether the connector is currently enabled or not.

- `void setAdaptOnClasses(boolean bol) :` This method allows to set the connector in static mode or in non-static mode. Invoke the method with `true` as parameter in order to change the connector to static mode. Calling the method with a `false` parameter, changes the connector to non-static mode. In non-static mode the hooks in the connector only trigger on instances explicitly registered with the connector. In static mode, the hooks trigger on all instances.
- `boolean adaptOnClasses() :` This method queries whether the connector is static or non-static. Returns `true` in case of a static connector and `false` in case of a non-static connector.
- `void setPriority(int k) :` Dynamically alters the connector priority, re-weaves if necessary.
- `int getPriority() :` Returns the priority.
-

2.3.5.5 Connector Combination Strategies

```
1   interface ConnectorCombinationStrategy {
2       public Vector validateCombinations(Vector,RuntimeContext);
3   }
```

Code Fragment 37: Connector combination strategy interface.

The connector registry is the central accessing point of the JAsCo run-time infrastructure and manages the active aspects and connectors. It also has a public interface that allows to enroll connector combination strategies. Connector combination strategies can be used to control the execution sequence of connectors. Likewise to normal combination strategies, connector combination strategies can be implemented in plain Java and allow to filter the list of all connectors at each joinpoint encountered. Code Fragment 37 illustrates the `ConnectorCombinationStrategy` interface. By implementing the `validateCombinations` method, the programmer is able to filter the list of connectors available at this joinpoint. The RuntimeContext parameter can be used to do some reflection about the current joinpoint. The Vector of connectors returned is used to find all hooks that are have to be triggered. When deleting a connector from the list, the hooks that are instantiated in this connector are not being triggered any longer. Deleting a connector from the list does not cause this connector to become permanently removed from the system, it is only removed from the list of connectors at the current joinpoint.

```
1   class Rearranger implements ConnectorCombinationStrategy {
2
3         public Vector validateCombinations  (Vector l,
4             RuntimeContext c) {
5
6            Collections.sort(l,new Comparator() {
7                public int compare(Object o1, Object o2) {
8                    String s1 =  o1.getClass().getName();
9                    String s2 =  o2.getClass().getName();
10                   return s1.compareTo(s2);
11                }
12           });
13           return l;
14        }
```

```
15 }
```

Code Fragment 38: Implementing a ConnectorCombinationStrategy in order to re-arrange connector sequence.

By implementing the `ConnectorCombinationStrategy`, arranging the connector sequence is possible. In addition, it is also possible to change some properties of the connectors using the run-time connector API. Code Fragment 37 illustrates a connector combination strategy that changes the sequence of the supplied connectors. The sequence of the connectors is changed to the alphabetic order of their names.

In order to enroll the connector combination strategies, the connector registry run-time API has to be used. Apart from adding and removing connector combination strategies, a limited form of reflection is supported. The run-time connector registry API supports the following methods:

- static void addConnectorCombinationStrategy(`ConnectorCombinationStrategy strategy`) : This method adds a connector combination strategy to the connector registry. Connector combination strategies are executed in the sequence they are added. As such, the connector combination strategy that is added latest, filters the hooks returned from the previous connector combination strategy.
- static void removeConnectorCombinationStrategy(`ConnectorCombinationStrategy strategy`) : This method removes a connector combination strategy from the connector registry.
- static Iterator getConnectors() : This method returns an iterator over the list of connectors. The connectors can be accessed and some properties can be changed using the connector run-time API.
- static void addConnectorRegistryListener(ConnectorRegistryListener l) : This method adds an observer to the connector registry. This observer is notified each time:
    - A new connector is detected in the system.
    - A connector is removed from the system.
    - A connector combination strategy is added.
    - A connector combination strategy is removed.

static void removeConnectorRegistryListener(ConnectorRegistryListener l) : This method removes the observer l from the connector registry.

3 Adaptive Programming and JAsCo

The JAsCo language presented so far is focused on describing aspects in the traditional AspectJ meaning. Adaptive Programming allows modularizing a different kind of crosscutting concerns using Adaptive Visitors. This section presents an extension of the JAsCo language that recuperates ideas from Adaptive Programming. The JAsCo Adaptive Programming implementation allows regular aspect beans to function as adaptive visitors. As such, aspect beans are able to implement both traditional AspectJ aspects and adaptive visitors, which increases their reusability even more. Aspect beans are instantiated as adaptive visitors onto a given traversal strategy in a special kind of connectors, named traversal connectors. In addition, traversal connectors are able to specify explicit precedence and combination strategies between cooperating adaptive visitors represented as aspect beans.

3.1 Introduction

Adaptive Programming [Lie96, LOO01] aims at isolating crosscutting concerns that originate from scattering an operation that involves several participants among these participants in order to adhere to the Law of Demeter. Adaptive Programming solves a very different kind of crosscutting concerns in comparison to other AOP approaches. In fact, Adaptive Programming is complementary to most other AOP approaches and can thus be employed simultaneously. An extension for AspectJ, called DAJ [SL02, LL02], has been recently introduced. DAJ combines the Adaptive Programming ideas with the AspectJ language. The Adaptive Programming ideas are applicable to the JAsCo approach as well. However, the ideas of JAsCo regarding independent aspects and expressive combinations of aspects and components, are also valuable for Adaptive Programming in the context of component-based software engineering. To motivate the possible contributions of the JAsCo ideas to current Adaptive Programming technologies, consider the following example:

```
1 class DataStorePersistence extends Visitor {
2
3    int i = 0;
4
5   public void before(DataStore store) {
6       if(isChangedSinceLastVisit(store)) {
7         FileOutputStream fw = new FileOutputStream("state"+i++);
8         ObjectOutputStream writer=new ObjectOutputStream(fw);
9         writer.writeObject(store.getData());
10        writer.close();
11      }
12   }
13
14   public boolean isChangedSinceLastVisit(DataStore store) {
15       //returns true if store is changed since the last visit
16   }
17 }
```

Code Fragment 39: DataStoreSerializer Adaptive Visitor that allows to serialize each visited data store on file.

The example of Code Fragment 39 illustrates a `DataStorePersistence` adaptive visitor implemented using the DJ library [OL01]. The `DataStorePersistence` adaptive visitor allows taking an incremental backup of the data stored in every visited `DataStore`. The visitor implements a before advice that fetches the data from each store and saves it to a file when the state of the store is changed

since the last visit. If the store is not changed since the last visit, the backup behavior is not executed. When this adaptive visitor is applied to an application, the visitor traverses the entire application and before visiting an object of type `DataStore` that is changed since the last backup, the data held in the store is written to file. As such, an effective incremental backup can be taken of the state of all the `DataStore` objects contained within an application. Code Fragment 40 demonstrates how the Adaptive Visitor of Code Fragment 39 can be used to backup all `DataStore` objects starting from the root system object of the application. Notice that using the adaptive visitor, the backup method does not need to hard-code the relations between the components. As a result, the `DataStorePersistence` visitor remains applicable when additional `DataStore` instances are added to the system or when the relationships between the system components change completely.

```
1 public void backup() {
2
3   ClassGraph cg = new ClassGraph(“system”);
4   Strategy sg = new Strategy("from system.Root to *");
5   TraversalGraph tg = new TraversalGraph(sg, cg);
6   tg.traverse(mySystemRoot,new DataStorePersistence());
7 }
```

Code Fragment 40: Instantiating a DataStoreSerializer in order to traverse the system for taking a backup of the state of the application.

In the context of component-based software engineering, a component has to be independent from other components in order to achieve a loosely coupled system. As such, a component can not rely on one specific component to realize its behavior. Translating this requirement to Adaptive Programming means that the adaptive visitors need to be completely independent from the components they visit. By the very nature of Adaptive Programming, they are already independent of the architecture of the component-based application at hand. However, adaptive visitors still refer to specific component types and specific component APIs, rendering the visitor not as reusable as required by CBSE. In Code Fragment 39 for instance, the `DataStore` type and the `getData` method are hard coded into the adaptive visitor. As a result, it is not possible to apply the same Adaptive Visitor onto a different context. In JAsCo however, aspect beans are abstract and reusable entities which do not depend onto specific component types in order to function properly. As a consequence, integrating the aspect independence idea into Adaptive Programming contributes to achieving a higher reusability and flexibility of adaptive visitors.

Another area where JAsCo ideas contribute to Adaptive Programming consists of the expressive combinations of aspects that JAsCo allows to describe. In the current Adaptive Programming implementations, there is only limited support to combine several adaptive visitors in such a way that they visit the same traversal strategy together. For example, suppose that in addition to the backup behavior, a log has to be kept of every object saved to file. This can only be achieved by adding some logging behavior to the `DataStorePersistence` adaptive visitor. As such, the logging concern is tangled with the backup concern. A visitor that implements logging behavior already exists in the DJ library, namely the `TraceVisitor`. Therefore, the tangled logging concern can be solved if a strong combination mechanism is available that allows to specify that the `TraceVisitor` and the `DataStorePersistence` visitor jointly visit the same traversal. However, the `TraceVisitor` can only be triggered whenever the `DataStorePersistence` visitor saves the visited object to file. Current Adaptive Programming implementations cannot specify such expressive combinations. For example, the DJ library allows combining several visitors on the same traversal strategy, but the composition mechanism is limited to specifying the precedence. As such, the JAsCo ideas concerning

expressive combinations of aspects using precedence and combination strategies are also applicable into the context of Adaptive Programming.

In the next sections, an extension to the JAsCo language is proposed, that recuperates the ideas of Adaptive Programming into the JAsCo language. The Adaptive Programming extension to JAsCo stays as close as possible to the original JAsCo concepts, while offering support for Adaptive Programming into a component-based context.

3.2 Employing Aspect Beans as Adaptive Visitors

An adaptive visitor is in fact very similar to a set of related advices as it is able to group several before, after and around methods that have to be executed when the corresponding component type is visited. Therefore, it seems natural to employ a regular JAsCo aspect bean as a kind of abstract and loosely coupled adaptive visitor. As such, aspect beans are able to describe both traditional aspects and traversal oriented aspects, which increases the reusability of aspect beans even more. Code Fragment 41 illustrates the aspect bean that implements the backup behavior introduced in the previous section. The aspect bean contains one hook, namely the `Backup` hook that implements the crosscutting backup behavior. The constructor of the hook expects one abstract method parameter, namely `triggeringmethod`. The hook is triggered when the method bound to the `triggeringmethod` abstract method parameter is executed. An `isApplicable` method defines an additional condition on which the hook is triggered, namely that the currently visited object is changed since it was last visited. If so, the hook has to be triggered. Similar to the adaptive visitor of Code Fragment 39, one before method is implemented that serializes the visited object to file using the abstract method `getDataMethod`. The `getDataMethod` is responsible for fetching the data from the called object and has to be implemented in a connector or child hook. Thus, the aspect bean does not refer to a specific component type or API. As such, the aspect bean remains completely independent and reusable.

```
1  class DataPersistence {
2
3    hook Backup {
4      int i = 0;
5
6      Backup(triggeringmethod(..args)) {
7        execution(triggeringmethod);
8      }
9
10     isApplicable() {
11      //returns true if thisJoinPointObject is changed since last
visit
12     }
13
14     before() {
15       FileOutputStream fw = new FileOutputStream (“state”+i++);
16       ObjectOutputStream writer=new ObjectOutputStream(fw);
17       writer.writeObject(getDataMethod(thisJoinPointObject));
18       writer.close();
19     }
20
21     public refinable Object getDataMethod(Object context);
20   }
21 }
```

Code Fragment 41: DataPersistence aspect bean that specifies a reusable backup aspect.

```
1 connector PersistanceConnector {
2
3   DataPersistence.Backup hook = new
4     DataPersistence.Backup(* DataStore.set*(*)) {
5       public Object getDataMethod(Object thisJoinPointObject) {
6         DataStore store = (DataStore) thisJoinPointObject;
7         return store.getData();
8     }
9   };
10
11  hook.before();
12 }
```

Code Fragment 42: Instantiating the DataPersistenceAspectBean in a regular JAsCo connector.

The `DataPersistence` aspect bean can be used as a traditional JAsCo aspect as illustrated by the connector of Code Fragment 42. This connector instantiates the `Backup` hook on every method of the `DataStore` class which name starts with `set`. As a result, the `triggeringmethod` abstract method parameter is bound to every method of the `DataStore` class which name starts with `set`. Every time the state of a `DataStore` object instance is altered using a method which name starts with `set`, the hook is triggered and a backup is taken. The `getDataMethod` refinable method of the hook is implemented by invoking the `getData` method on the called object. Notice that casting the called object to a `DataStore` is safe because the hook is only instantiated on objects of type `DataStore`.

In order to instantiate an aspect bean as an adaptive visitor, a new kind of connector is introduced, namely a *traversal connector*. A traversal connector specifies a traversal strategy and instantiates hooks as adaptive visitors on specific components within the specified traversal. Code Fragment 43 illustrates an example traversal connector that instantiates the aspect bean of Code Fragment 41 on the *from system.Root to* * traversal strategy (line 1). The `visiting` keyword allows declaring on which specific classes encountered within the traversal, the hook has to be triggered. In this example, the `Backup` hook is only triggered on `DataStore` objects. The result of applying the traversal connector of Code Fragment 43 is the following: "traverse the object structure as specified by the traversal strategy *from system.Root to* * and invoke the before advice of the `Backup` hook each time a `DataStore` object is visited during the traversal". Likewise to a regular connector, the `getDataMethod` is implemented in order to fetch the data of the visited objects.

```
1 traversalconnector BackupTraversal("from system.Root to *") {
2
3   DataPersistenceAspectBean.Backup hook = new
4     DataPersistenceAspectBean.Backup(visiting DataStore) {
5        public void getDataMethod(Object visitedobject) {
6         DataStore store = (DataStore) visitedobject;
7         return store.getData();
8       }
9   };
10
11  hook.before();
12 }
```

Code Fragment 43: BackupTraversal connector .

The main difference between aspects in the "AspectJ sense" and Adaptive Visitors consists in the invocation. Traditional aspects are invoked implicitly whenever the current joinpoint matches the pointcut specification of the aspect. For example, in the connector of Code Fragment 42, the backup hook is

triggered every time a method which name starts with `set` is executed on a `DataStore` object. Traversal strategies however, need to be invoked explicitly in order to start the traversal. For example, Code Fragment 44 illustrates how the traversal specified in Code Fragment 43 is invoked. The `mySystemRoot` is an instance of the `system.Root` class from where the traversal has to start.

```
1 public void backup(system.Root mySystemRoot) {
2      BackupTraversal myBackup = BackupTraversal.getTraversal();
3      myBackup.traverse(mySystemRoot);
4 }
```

Code Fragment 44: Invoking the BackupTraversal connector of Code Fragment 43.

The DJ library is employed behind the scenes in order to implement the behavior described by a traversal connector. As such, the backup behavior realized using JAsCo aspect beans and traversal connectors, is the same as the behavior established using the DJ library in Code Fragment 40. The involved adaptive visitors, which are represented as JAsCo aspect beans, are however now truly reusable and independent of specific component types.

```
1   class SearchBean {
2
3      Object searchObject;
4      List visitednodes = new List();
5
6      Object getSearchObject() {
7         return searchObject;
8      }
9      void setSearchObject(Object o) {
10        searchobject=o;
11     }
12     List getResultingPath() {
13        return visitednodes;
14     }
15
16     hook BuildPath {
17
18       BuildPath(visitingmethod(..args)) {
19         execution(visitingmethod);
20       }
21
22       around() {
23
24         if(thisJoinPointObject==global.getSearchObject())
25            ;
26         else {
27            global.visitednodes.add(thisJoinPointObject);
28            visitingmethod();
29         }
30       }
31     }
32  }
```

Code Fragment 45: Search aspect bean that allows to build a path of objects visited in order to reach a specific object.

One might wonder to which method the abstract method parameter `triggeringmethod` of the hook of Code Fragment 41 is bound. Indeed, the connector of Code Fragment 43 specifies that this abstract method parameter is bound to the "visiting DataStore" concept, which is not a concrete method signature. However, the result of the "visiting DataStore" declaration is that all `DataStore` objects encountered in the system are visited. The visiting behavior itself can be perceived as a method execution and is also

implemented as such in the DJ library. As a result, the method bound to the `triggeringmethod` abstract method parameter corresponds to the implicit method that implements the visiting behavior itself. This allows the aspect bean to effectively manage the visiting process itself in a hook. When a hook implements an around advice, the hook might decide to stop the visiting process by not invoking the abstract method parameter which is bound to the implicit visiting method. This is illustrated by the aspect bean of Code Fragment 45. This aspect bean searches a specific object and builds a path of objects visited while traversing the object structure. The hook `BuildPath` specifies a constructor which expects one abstract method parameter, named `visitingmethod`, that has to be bound to the visiting declaration in a connector. In addition, the hook implements an around advice that is responsible for building the list of visited nodes. When the object to search for has been reached, the traversal stops because the method bound to the abstract method parameter `visitingmethod` is not invoked. Otherwise, the current visited object is added to the list of visited nodes and the traversal is continued by invoking the method bound to `visitingmethod`. Notice that the semantics of the `thisJoinPointObject` keyword is also changed when the hook is used as an adaptive visitor. Instead of referring to the object where the currently executing method has been invoked upon, the `thisJoinPointObject` keyword refers to the currently visited object. This is the logical behavior because if the traversal behavior itself is perceived as a method, the visiting of an object can be perceived as the execution of that method on that object.

Code Fragment 46 illustrates a traversal connector that instantiates the `BuildPath` hook on all the classes in the system using a wildcard. The resulting traversal starts at an instance of the class `system.Root` and visits every reachable object from that instance. When the object that has to be found is located, the traversal is stopped.

```
1 traversalconnector SearchTraversal("from system.Root to *") {
2
3       SearchBean.BuildPath builder = new
4                 SearchBean.BuildPath(visiting *);
5
6       builder.around();
7 }
```

Code Fragment 46: Instantiating the BuildPath hook on all the classes within the system.

3.3 Advanced features of JAsCo Traversal Connectors

3.3.1 Precedence and Combination Strategies

An aspect bean, employed as adaptive visitor, is completely independent of concrete components and other aspect beans. A strong mechanism is provided in order to combine them. Similar to regular JAsCo connectors, traversal connectors are also able to explicitly declare precedence and combination strategies. This allows specifying several adaptive visitors onto a single traversal strategy as a complex combination of several cooperating aspect beans. The following paragraphs illustrate how the JAsCo precedence and combination strategies can be used in the context of Adaptive Programming.

When two or more aspect beans are combined in one traversal connector specification, it is important to be able to explicitly declare precedence of the involved beans. Code Fragment 47 illustrates a traversal connector that instantiates both the `Backup` hook and the `FileLogger` hook introduced in section 2.3.3. Likewise to the example of Code Fragment 43, the `Backup` hook is instantiated on the visiting of class `DataStore`. The `FileLogger` hook is instantiated on all the classes which are visited during the traversal using a wildcard. The traversal connector allows to explicitly control precedence of both hooks

when they both visit the same object (line 11-12). In this case, the before advice of the logger hook has to be triggered prior to triggering the before advice of the backup hook.

```
1   traversalconnector BackupTraversal("from system.Root to *") {
2
3     DataPersistenceAspectBean.Backup backup = new
4      DataPersistenceAspectBean.Backup(visiting  DataStore) {
5         public void getDataMethod(Object context) {
6            DataStore store = (DataStore) context;
7            return store.getData();
8         }
9     };
10
11    Logger.FileLogger logger = new
12         Logger.FileLogger(visiting *);
13
14    logger.before();
15    backup.before();
16  }
```

Code Fragment 47: Specifying a precedence strategy in a traversal connector.

However, suppose the collaboration between the `Backup` hook and the `FileLogger` hook changes so that a log is kept of all objects that have been saved to file. Remember that the `Backup` hook does not save objects of which the state is not altered since the last visit. As such, the `FileLogger` hook can only be triggered when the `Backup` hook is triggered. For these kinds of collaborations among hooks, combination strategies can be employed.

Code Fragment 48 illustrates a twin combination strategy that takes two hooks as input in its constructor (line 4-7). The `validateCombinations` method specifies that the second hook (`hookB`) is only kept in the list of hooks when the first hook (`hookA`) occurs in the list of hooks. As such, the second hook can only be triggered when the first hook is also triggered. Using this "twin" combination strategy, the traversal connector of Code Fragment 47 can be extended to only allow the logging hook to be triggered when the backup hook is also triggered. This is illustrated by Code Fragment 49. The `TwinCombinationStrategy` is instantiated with both the backup and logger hooks. Likewise to combination strategies in regular JAsCo connectors, the `TwinCombinationStrategy` is added to the traversal connector using the `addCombinationStrategy` keyword.

```
1   class TwinCombinationStrategy implements CombinationStrategy
2   {
3       private Object hookA, hookB;
4       TwinCombinationStrategy (Object a,Object b) {
5            hookA = a;
6            hookB = b;
7       }
8
9       HookList validateCombinations(Hooklist hlist) {
10
11           if (!hlist.contains(hookA)) {
12                hlist.remove(hookB);
13           return hlist;
14      }
15  }
```

Code Fragment 48: Twin combination strategy that makes sure that hookB is only triggered when hookA is also triggered.

```
1   traversalconnector BackupTraversal("from system.Root to *") {
2
3     DataPersistenceAspectBean.Backup backup = new
4       DataPersistenceAspectBean.Backup(visiting  DataStore) {
5        public void getDataMethod(Object context) {
6          DataStore store = (DataStore) context;
7          return store.getData();
8        }
9      };
10
11     Logger.FileLogger logger = new
12         Logger.FileLogger(visiting *);
13
14     logger.before();
15     backup.before();
16
17     TwinCombinationStrategy twin = new
18          TwinCombinationStrategy(backup,logger);
19     addCombinationStrategy(twin);
20  }
```

Code Fragment 49: Specifying a combination strategy in a traversal connector.

3.3.2 Traversal Strategies

The traversal strategies specified in a traversal connector are passed directly to the DJ library when the traversal connector is compiled. As such, the traversal strategies supported by JAsCo traversal connectors are identical to those supported by the DJ library. Examples of more advanced keywords for traversal strategies include:

- `bypassing:` The bypassing keyword allows to denote classes that may not be visited during the traversal. For example, in the traversal strategy *"from A bypassing C to B"*, an instance of the class C can not be visited in order to traverse starting from an instance of class A and reaching an instance of class B. In other words, when considering the class hierarchy as a class graph, all paths from class A to B that contain the class C are not visited. An exception is thrown if such a traversal is not possible.
- `via:` The via keyword denotes the opposite of the bypassing keyword, namely that the class corresponding to that keyword has to be visited during the traversal. For example, in the traversal strategy *"from A via C to B"*, an instance of class C has to be visited in order to traverse from an instance of class A to an instance of class B. In other words, when considering the class hierarchy as a class graph, all paths from class A to B that do not contain the class C are not visited. An exception is thrown if such a traversal is not possible.
- `->Start,field,End:` This expression can be combined with the `via` or `bypassing` keywords to specify a relation between two classes instead of a single class. In other words, when considering the class hierarchy as a graph, the pattern specifies the edge labeled `field` from class `Start` to class `End`. For example, in the traversal strategy *"from A via ->C,f,* to B"*, an instance of class C has to be visited in order to traverse from an instance of class A to an instance of class B. In addition, after visiting class C, the traversal can only continue by following the field f. The wildcard denotes that by following the field f any target class may be reached.

3.4 Concluding remarks

Employing independent aspect beans and expressive combinations improves on current Adaptive Programming implementations in the context of component-based software engineering. Adaptive visitors implemented as aspect beans are now truly reusable as no context information is hard coded. Aspect beans can even be reused both as aspects instantiated in regular JAsCo connectors and as adaptive visitors instantiated in traversal connectors. In addition, adaptive visitors implemented as aspect beans can easily be combined in traversal connectors in order to visit the same traversal as specified by the common traversal strategy. The traversal connector syntax is very close to the original JAsCo syntax making the Adaptive Programming extension to JAsCo quite logical and easy to learn from the JAsCo point of view. However, a drawback of this approach is that the JAsCo syntax and keywords are not always intuitive from the viewpoint of Adaptive Programming. Indeed, keywords often have a different semantics when the hook is used as an adaptive visitor. The `thisJoinPointObject` keyword for example refers to the visited object instead of a called object when the hook is used as an adaptive visitor, making the keyword less intuitive. In addition, the constructor body of a hook is in fact ignored when employing this hook as an adaptive visitor. This could be solved by changing the semantics of the hook constructor keywords to be useful in a traversal specification. For example, *Cflow* could denote that the corresponding class has to be visited prior to the currently visited class. Likewise, *withincode* can be used to specify that the corresponding class is the previously visited class. Changing the semantics of a hook constructor is not a completely satisfying solution as it causes confusion. As such, aspect beans become somewhat less readable and maintainable.

4 Stateful Aspects

(see the stateful aspects in JAsCo paper at the website for an extended explanation of stateful aspects)

Most aspect-oriented approaches currently offer conditions about which methods are in the current control flow for determining whether an aspect needs to be triggered or not. However, it is often required to specify conditions about events that have occurred before the current joinpoint, regardless of whether an activation record for that event is still on the stack. These kind of general protocol history conditions are rarely supported in current aspect-oriented approaches. Douence et al. [8] propose a formal model for aspects with general protocol based triggering conditions, named stateful aspects. The JAsCo language includes a realization of this formal model. To illustrate this stateful aspects feature in JAsCo, suppose that the access control aspect introduced in Code Fragment 1 has to be extended so that it is only triggered when a user is actually logged in into the system. Hence, the access control aspect should only start checking access permissions when a user logs in and stop checking access permissions when a user logs out. As a result, a stateful triggering condition is required. Code Fragment 50 shows an improved version of the access control aspect bean which takes into account a starting and stopping condition.

```
1  class StatefullAccessManager {
2
3     ...
4
5     hook AccessControl {
6
7       AccessControl(startm(..a1),runningm(..a2),stopm(..a3))) {
8         start>p1;
9         p1: execution(startm) > p3||p2;
10        p2: execution(runningm) > p3||p2;
11        p3: execution(stopm) > p1;
12      }
13
14      isApplicable p2() {
15        return !p_db.isRoot(currentUser);
16      }
17
18      around p2() {
19        if(p_db.check(currentuser,thisJoinPointObject)) {
20          return proceed(); }
21        else {
22          throw new AccessException(); }
23      }
24    }
25 }
```

Code Fragment 50: The stateful access control aspect bean.

The implementation of the `AccessControl` hook is quite similar to its original version. The constructor (lines 7 till 12) takes three abstract method parameters (`startm`, `runningm` and `stopm`) as input. The constructor body specifies the stateful triggering condition by enumerating a set of transitions from which the first one is highlighted using the start keyword. Each transition is denoted by a label and contains a JAsCo compatible pointcut definition and an enumeration of subsequent transitions. When a transition pointcut evaluates to true, the state of the stateful aspect bean is altered and the subsequent transition enumeration is processed for the consecutive encountered joinpoints. The access control aspect of Code

Fragment 50 starts executing its behavior in transition `p1` where it waits for its starting method to be executed in order to move on to transition `p2` (line 9). The aspect remains in state `p2` (line 10) while the running method is being executed and will only move on to transition `p3` if the stopping method is executed (line 11). Likewise to normal aspects, stateful aspects are able to contain an `isApplicable` method that specifies an additional triggering condition. This method can be applicable onto all transitions or can be delimited to a specific one. In this case, the `isApplicable` method is only valid for transition `p2` as the condition whether a user is an administrator must only be checked at the moment a user is actually logged in. Similarly, advices can be applicable for all transitions or limited to one or more specific transitions. In this case, the around advice is again only applicable for transition `p2`.

```
1  static connector PrintAccessControl {
2
3    AccessManager.AccessControl control =
4      new AccessManager.AccessControl(
5       void System.Login(User), * *.*(*), void System.Logout(User)
6      );
7
8  }
```

Code Fragment 51: The JAsCo connector for stateful access control.

Code Fragment 51 illustrates how the stateful access control aspect is deployed within the system. The `Login` and `Logout` methods are bound to the start and stop method of the access control aspect. Hence, access control will start at the moment a user logs in and will stop at the moment a user logs out. At that moment, the aspect waits again for the execution of the `Login` method to start its access control behavior.

Explicitly supporting stateful conditions in the constructor allows capturing them cleanly. Otherwise, state information needs to be maintained in the advices, which pollutes the advice code. In addition, explicitly specifying the stateful conditions allows optimizing the aspects performance-wise as the system knows which joinpoints the aspect is currently interested.