Aspects

J.Hřivnáč*, LAL, Orsay, France

Abstract

Aspect-Oriented Programming (AOP) is a new paradigm promising to allow further modularization of large software frameworks, like those developed in HEP. Such frameworks often manifest several orthogonal axes of contracts (Crosscutting Concerns - CC) leading to complex multidepenencies. Currently used programing languages and development methodologies don't allow to easily identify and encapsulate such CC. AOP offers ways to solve CC problems by identifying places where they appear (Join Points) and specifying actions to be applied at those places (Advices). While Aspects can be added in principle to any programming paradigm, they are mostly used in Object-Oriented environments. Thanks to wide acceptance and rich object model, most Aspect-Oriented toolkits have been developed for Java language. Probably the most used AOP language is AspectJ.

WHAT'S WRONG ?

OOP Limitations

In OOP, an Object is the only fundamental abstraction. In real life, however, other abstractions are needed, e.g. Before-after, Cause-effect or State. In OOP, Hierarchies (is_a) and Collections (has_a) are the only relations. In real life, however, other relations are needed, e.g. Master-slave, NxM, Component-container, Interval or Element-metadata. OOP solves this limitation using work-arounds (Patterns, Hooks, Wrappers,...). Aspects can be the first step of a more organic solution.

Crosscutting Concerns

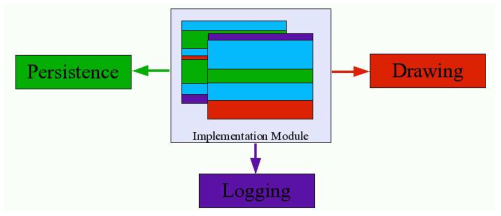


Figure 1: Crosscutting Concerns.

As shown in Figure 1, besides its own Mission, classes have to fulfill other (unrelated) tasks, like Logging/Tracing, Authentication, Persistency, Exception handling, Contract Enforcing, Distribution, Self-testing, etc. Those tasks are spread over classes from different domains. OOP doesn't give natural tools to modularize them.

* Julius.Hrivnac@cern.ch

Crosscutting Concerns have serious impact on source code:

- Code Tangling
- Code Scattering (Duplicated Code and Complementary Code)

With consequences on software quality:

- Poor Traceability
- Low Reuse
- Hard Evolution

Traditional OOP (abstract interfaces,...) can't modularize Crosscutting Concerns because:

- Using interfaces, implementation should be defined for each class.
- Interface can't define which classes it should act on.
- Hooks (Publish/Subscribe, Visitor,...) must be placed before affected class.
- Wrappers can be circumvented.

ASPECT ORIENTED PROGRAMMING [1]

Aspect Definition

Lets separate Crosscutting Concerns from the Core Concern, move them from the Class into other entities, and re-introduce them later. Lets call them Aspects. We have just introduced a new level of Modularization (an Aspect) and a new kind of Relationship (is_an_aspect_of). We have used Aspect-Oriented Methodology, which consist of Aspectual Decomposition, Concern Implementation and Aspectual Recomposition.

Aspect Structure

An Aspect consist in general of four components:

- Join Point, which is identifiable point, formally described by a PointCut. It can be Method (either call or execution), Constructor (call or execution), Field Access (read or write), Exception throwing, Initialization (of a class or an object) or Advice Execution itself.
- Advice, which is the code to be executed at Join Point. It can be executed before, after (returning, throwing or always) or around its Joint Point.

- Introduction, which is a modification of class code itself.

- Compile-time Declaration of warnings or errors.

Aspect can extend class, implement interface, extend another aspect and contain methods and data. An Aspect is analogical to OOP Class, where PointCut corresponds to Method declaration and Advice corresponds to Method implementation.

ASPECT SYNTAX

AspectJ [2]

AspectJ uses extensions to Java, so Aspects have to be compiled by special tools. AspectJ Weaving rules are defined inside Aspects. Following is as simple example of an AspectJ Aspect:

```
public aspect TracingAspect {

  /** Choose all calls to methods in hep package
    * issued from HelloWorld. */
  pointcut callAnyMessage() : within(hep.*.HelloWorld) &&
                              call(* hep.*.*(..));

  /** Choose all executions of
    * HelloWorld.say(String) method, pass argument
    * to advice. */
  pointcut executeSayMessage(String s) :
    execution(public * hep.*.HelloWorld.say(String)) &&
    args(s);

  /** Trace calls before. */
  before() : callAnyMessage() {
    System.out.println("before " + thisJoinPoint);
    }

  /** Trace executes after. */
  after() : executeSayMessage(String s) {
    System.out.println("after saying " + s);
    }

  /** Modify execution. */
  Object around() : callAnyMessage() {
    ...
    Object obj = proceed();
    ...
    return obj;
    }

  /** Modify class hierarchy,
    * declare Track PersistenceCapable. */
  declare parents : Track implements PersistenceCapable;

  /** Add Vertex to Track. */
  private Vertex Track._vertex;

 /** Issue error, if user wants to create Track
   * directly instead of by TrackFactory subclass. */
  declare Error :
    call(Track.new(..)) &&
    ! within(TrackFactory+):
      "Only TrackFactory can create Tracks !";

  }
```

AspectWerkz [3]

AspectWerkz shows two major differences to AspectJ:

- Aspect is a normal Java class so it can be compiled by a standard compiler and distributed as a standard jar library.

- Weaving Rules can be external (in XML) so they can be applied independently, later. Weaving Rules can be also expressed using Java 1.5 Annotations.

Other Systems

Three constructs of Aspects (PointCut, Advice and Weaving instructions) can be written in different language:

- Target language itself
- Extension of a Target language
- XML
- (Embedded) Annotations
- Special language
- Framework/GUI

They can be all in the same unit or in different units for different Constructs.

Following is the list of the most popular Aspect systems (with their GoogleMark values as of 1st October 2004):

- **Java (195k)** which can be AspectJ (125k), AspectWerkz (40k), Java Aspect Components [4] (20k) or JBoss AOP [5] (10k).

- **C/C++ (4k)** which can be AspectC [6] (2k) or AspectC++[7] (2k).

- Others, like Pythius [8]/Pythonic [9] (0.5k) for Python, Aspect for Perl [10], AspectR [11] (3k) for Ruby or AspectC# [12] (2k) for C#.

- Some languages, like Lisp or Smalltalk, have a basic support for Aspect-like programming already included.

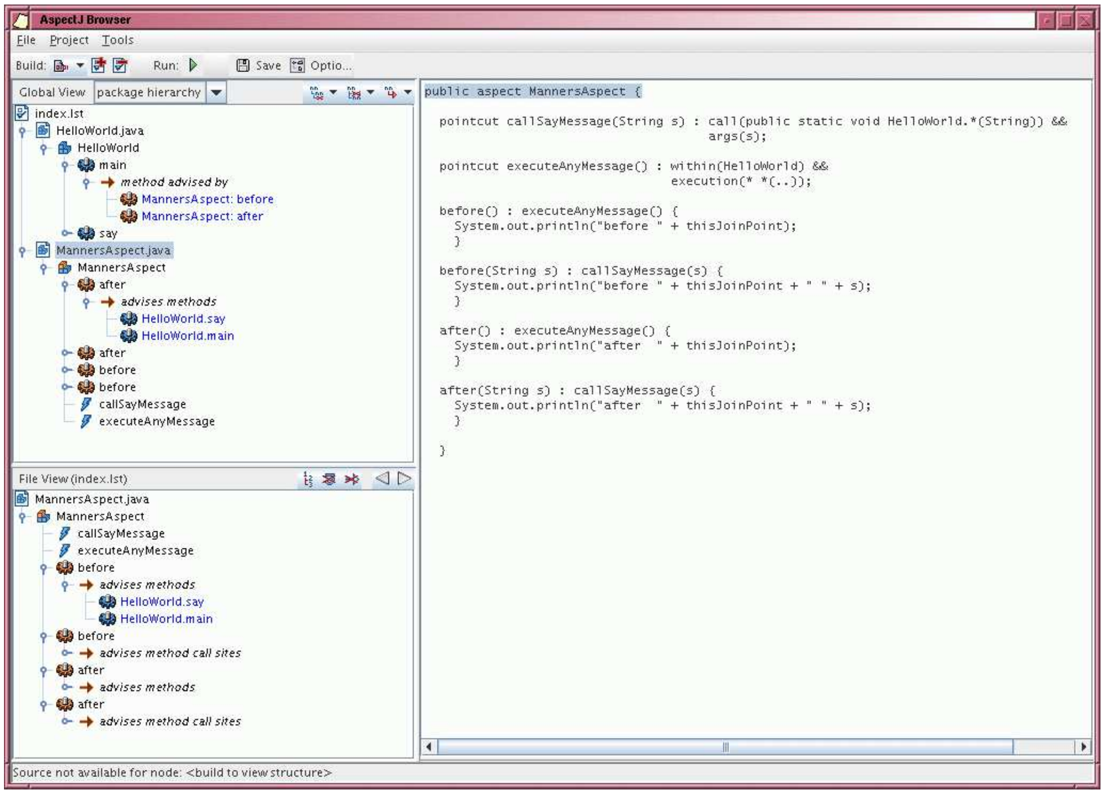
Level of language support in AOP systems varies; Java is generally well supported, C++ support is still very incomplete.

Weaving

Weaving (i.e. introducing of Aspects into the code) can be done in several phases (as is shown on Figure 2). More dynamic (load/run-time) weaving method's are supported only by some implementations.

GUI

Graphical User Interface is often needed (and supported) to visualize the weaving process. GUI studios (like AspectJ Browser shown in Figure 3) may support incremental compilation and weaving.



```
public aspect MannersAspect {

  pointcut callSayMessage(String s) : call(public static void HelloWorld.*(String)) &&
                                      args(s);

  pointcut executeAnyMessage() : within(HelloWorld) &&
                                 execution(* *(..));

  before() : executeAnyMessage() {
    System.out.println("before " + thisJoinPoint);
    }

  before(String s) : callSayMessage(s) {
    System.out.println("before " + thisJoinPoint + " " + s);
    }

  after() : executeAnyMessage() {
    System.out.println("after  " + thisJoinPoint);
    }

  after(String s) : callSayMessage(s) {
    System.out.println("after  " + thisJoinPoint + " " + s);
    }

}
```

Source not available for node: <build to view structure>

Figure 3: AspectJ Browser.

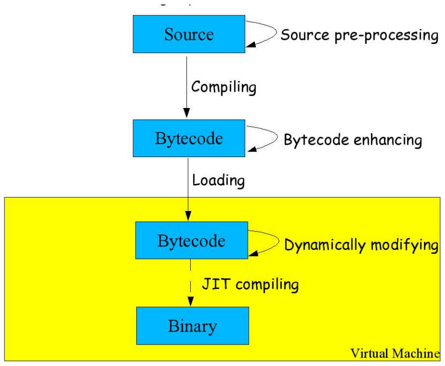


Figure 2: Weaving.

ASPECTS APPLICATION

The list of domains, where AOP can be used, is very long. Let's name just some: Logging/Tracing, Exception Handling, Monitoring/profiling, Unit testing, Const (for Java) / Final (for C++), Cache Management, Connection Pool, Contracts Enforcing, Security/Authentication/Authorization, Distribution, Grid, Coding Conventions Checking, Web Service, Graphics, Multiple Inheritance (for Java), Mixin, Persistence, or Fine-grained access (besides public, package, protected, private and friend).

Patterns have a special role in AOP. They are often used to correct problems in OO languages. In AOP, some Patterns disappear (like Observer or Visitor), other simplify (like Factory, MVC or Entity-Model-Representation).

Aspects in HEP

There are several HEP domains, where AOP can be very useful:

- **Graphics:** Aspect uses core class and performs all graphical actions for it. This approach is prototyped to connect GraXML [13] display to external framework.
- **Fine-Grained Access Control:** Aspect checks that only allowed relations are used.
- **Cache:** Around advice stores all results in a cache. Cached result is returned if it exists
- **Connection Pool** is analogical to Cache.
- **Web Service:** Aspect wraps serving class in a Web Service. Around advice forwards service request through Web Service.
- **Persistence:** AOP can be used in several ways to provide persistence to Objects. Aspect introduces read/write functions. Field access advice performs reading/writing when necessary. Aspect makes class JDO [14] PersistenceCapable (this is used in JOnAS Speedo [15]). (JDO) PersistenceCapable aspect connects to a core class and handles its persistence (it has

been prototyped for AIDA [16] FreeHEP [17] JDO-based persistence).

- **Factory:** Around advice returns unique Object on all Constructor calls. Compile-time declaration checks that objects are not created directly, but by a Factory.

- **Singleton:** Around advice on Constructors returns single Object, if it already exists; creates it otherwise

DEVIL'S ARGUMENTS

The are certainly problems with AOP, some just reflect its immaturity, others are more fundamental.

Immaturity

- Aspect syntax is not standardized, there are several incompatible approaches.

- AOP Theoretical Foundation is not yet very solid.

- AOP Methodology is still very primitive. UML syntax for Aspects is not yet standardized

- PointCuts rely on naming conventions, they use just (a bit better) regular expressions and pattern matching (with weak grammar).

Fundamental problems

- AOP breaks encapsulation. *(Yes, but in a controlled way. Otherwise, equivalent functionality would require more serious break.)*

- AOP improves locality of Concerns, but destroy locality of Control Flow. Control Flow of program with Aspects is difficult to understand. Tools are necessary. *(But that is true for Object Oriented Program compared with Procedural Program too.)*

- Aspects can change program behavior without original author being aware (and what about copyright ?). *(But this is what we want.)*

- AOP programs can be hard to evolve as they rely on (coding) conventions. Objects depend on Aspects, but Aspects depend on Objects' structure. *(This is not much more serious that pre-AOP dependencies in OOP.)*

HEPASPECTS [18]

HEPAspects is a package hosting some Reusable Aspects (incl. examples from this talk). It's not yet very big, but its is growing. It contains also Ant tasks for AspectJ management and scripts to allow a simple Aspects weaving into user jar-libraries.

CONCLUSION

Object Oriented Programming abstractions are not rich enough to capture actual Use Cases. In particular, Crosscutting Concerns can't be expressed. Various ways have been invented to fix that problem (OO Patterns, etc.). Those solutions are too complex and fragile as they are not native to existing (OO) languages (they manifest an Abstraction Leak). Aspect Oriented Programming offers organic way of modularizing Crosscutting Concerns. AOP (in Java) is solid, easy to use and powerful (maybe too). It is, however, in a rapid evolution and its impact on Architecture is not yet clear. There are several fully functional AOP systems, the most popular is AspectJ. Many HEP Crosscutting Concerns can be easily separated with AOP. **AOP (in Java) is ready for Development and optional Production Aspects.**

REFERENCES

[1] Aspect Oriented Software Development (http://aosd.net)

[2] AspectJ Project (http://eclipse.org/aspectj)

[3] AspectWerkz - Dynamic AOP for Java (http://aspectwerkz.codehaus.org/)

[4] JAC - A Framework for Aspect-Oriented Programming in Java (http://jac.objectweb.org)

[5] JBoss Aspect Oriented Programming (http://www.jboss.org/products/aop)

[6] AspectC (http://www.cs.ubc.ca/labs/spl/projects/aspectc.html)

[7] AspecC++ (http://www.aspectc.org)

[8] Pythius (http://pythius.sourceforge.net)

[9] Pythonic CherryPy (http://www.cherrypy.org)

[10] AOP for Perl (http://search.cpan.org/ eilara/Aspect-0.10/lib/Aspect.pm)

[11] AspectR - Simple aspect-oriented programming in Ruby (http://aspectr.sourceforge.net)

[12] AspectC# (http://www.dsg.cs.tcd.ie/index.php?category_id=169)

[13] GraXML - Framework for manipulation and visualization of geometrical objects in space (http://home.cern.ch/hrivnac/Activities/Packages/GraXML)

[14] Java Data Objects (http://access1.sun.com/jdo, http://www.jdocentral.com)

[15] Speedo - ObjectWeb Implementation of JDO (http://speedo.objectweb.org)

[16] AIDA - Abstract Interfaces for Data Analysis (http://aida.freehep.org)

[17] FreeHEP - HEP Components and Tools for Java (http://java.freehep.org)

[18] AspectJ HEP Aspects (http://home.cern.ch/hrivnac/Activities/Packages/HEPAspects)