AspectJ
Aspect Oriented Programming for Java

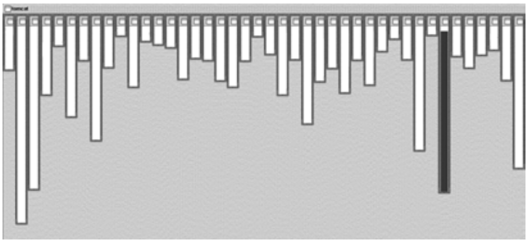
Tapani Ojanperä / Turku University of Applied Sciences

What is AspectJ

- A definition from the Wikipedia
- **”AspectJ** is an aspect-oriented extension to the Java programming language created at Xerox PARC by Chris Maeda, who originally coined the term "aspect-oriented programming" (no one remembers exactly when). Gregor Kiczales coined the term "crosscutting". The Xerox group's work was integrated into the Eclipse Foundation's Eclipse Java IDE in December 2002, abandoning support for users of the Netbeans IDE at this point. This helped AspectJ become one of the most widely-used aspect-oriented languages”.
- What is the need of AspectJ? Let us look at an example [PARC02].

good modularity

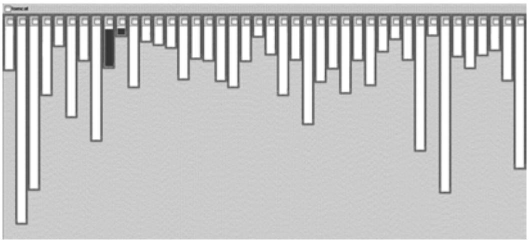
XML parsing



- XML parsing in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in one box

good modularity

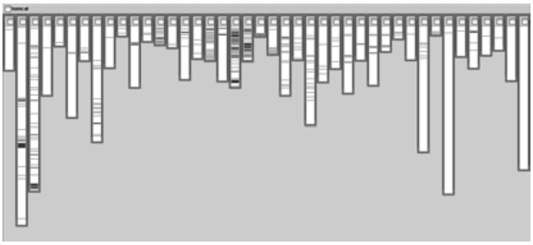
URL pattern matching



- URL pattern matching in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in two boxes (using inheritance)

problems like…

logging is not modularized



- logging in org.apache.tomcat
  - red shows lines of code that handle logging
  - not in just one place
  - not even in a small number of places

Background

- AOP is based on OOP.
- OOP is inflexible. When the class hierarchy is chosen, it is difficult to change.
- Class hierarchy is only one view or **aspect**.
- AOP gives the possibility to include different views.
- If we think that class hierarchy is vertical, we may consider aspects horizontal.

Background

- We define that **concern** is the target of interest.
- We may also say that concern is a way to modularize a group of classes.
- We have two approaches.
- First is **asymmetric**. Then the class hierarchy is the most important aspect and the other aspects (cross-cutting concerns) are less important.
- The **symmetric** approach provides that all aspects have equal importance.

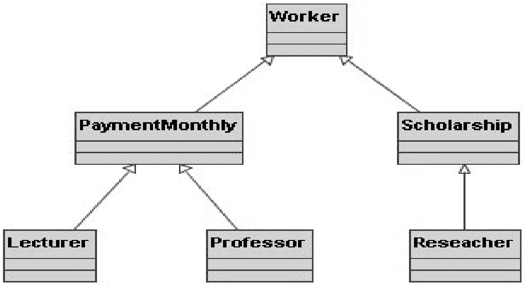
Background

- Many programs contain code fragments that are repetitive in some sense.
- Java has for example exceptions with try and catch blocks (exception aspect).
- Another example is threads, where we have to check that many threads cannot use the same resource at the same time (concurrency aspect).
- Programs need check the types of parameters (pre-conditions).
- Often the tracing and logging is necessary, too.

Background

- Aspect languages separate aspects into their own files.
- The code an aspect handle, is usually neither sequental nor in the same class.
- This kind of code is chosen according to some criteria.
- This resembles slicing, where code is sliced with some variables [Weiser84].

An example of workers



Worker

PaymentMonthly

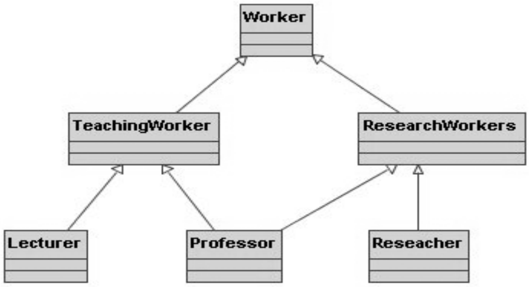
Scholarship

Lecturer

Professor

Reseacher

Another view of workers



Worker

TeachingWorker

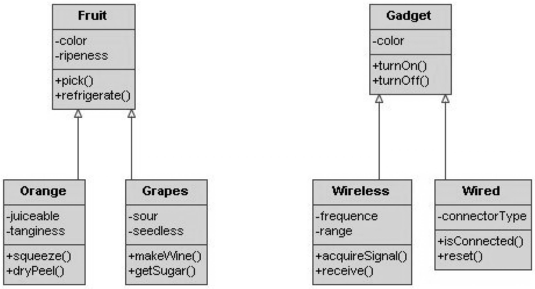
ResearchWorkers

Lecturer

Professor

Reseacher

A crosscutting example



Fruit
-color
-ripeness
+pick()
+refrigerate()

Gadget
-color
+turnOn()
+turnOff()

Orange
-juiceable
-tanginess
+squeeze()
+dryPeel()

Grapes
-sour
-seedless
+makeWine()
+getSugar()

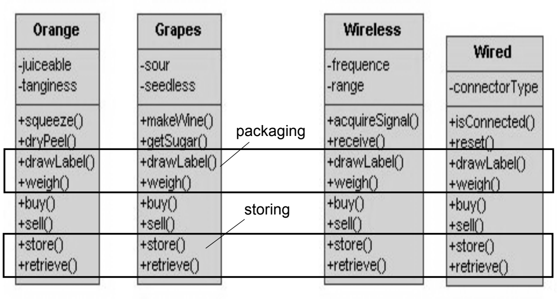
Wireless
-frequence
-range
+acquireSignal()
+receive()

Wired
-connectorType
+isConnected()
+reset()

A crosscutting example

- There seems to be two independent class hierarchies in the picture.
- However, if we think the fruits and gadgets such products we can sell, then the existing class hierarchies are not enough.
- Now we draw some **cross-cutting** aspects according to packaging and storing.

A crosscutting example



Aspect languages

- AspectJ is an aspect extension for Java and at the moment the most important aspect language.
- AspectL is an extension for Common Lisp. P. Costanza has been developing it for some time.
- There are also numerous other extensions: AspectC++, Aspect#, Pythius and PEAK (Python), Aspect module (Perl), AspectR (Ruby), AspectPHP, AspectXML, AspectS (Squeak), AOP Fun with JavaScript.

Concepts of AspectJ

- *Join point* is a well-defined point in the execution of a program.
- *Pointcut designator* detects the join point.
- For example the designator
  call(void Orange.store())
  detects the **call** to the method store in class Orange.
- Join points are usually related to method calls, object initialization, field references and exceptions.

Concepts of AspectJ

- When the join point has detected, the aspect code is performed.
- The designator has often a name, for example

```
pointcut storing() :                       or
    call ( void Orange.store()) ||
     call ( void Grapes.store()) ||
     call ( void Wireless.store()) ||
     call ( void Wired.store()))
```

Concepts of AspectJ

- A **pointcut** is a program element that picks out join points and exposes data from the execution context of those join points [eclipseOrg].
- Each join point has three pieces of state associated with it:
  - the currently executing object
  - the target object
  - an object array of arguments

Concepts of AspectJ

- Respectively, there are three state-exposing pointcuts:
  - this
  - target
  - args
- Pointcut designators can include wildcards.
- call(public * Wired.*(..))
- Every public method in the class Wired can be selected.

Advices

- When we have join points, we have to know, what to do in these points.
- An **advice** is a code, which is related to join points.
- There are three alternatives for advices:
  - **before** code is executed just before the point.
  - **after** code is executed after the point.
  - **around** code is executed instead of the point.

Comparing with CLOS

- There are the same type of methods in CLOS (Common Lisp Object System).
- CLOS is based on the Meta Object Protocol (MOP).
- G. Kiczales has planned MOP and has also been in the AspectJ team of PARC.
- Obviously some ideas have transported from CLOS to AspectJ [Seibel05].

Advices

- We had earlier the pointcut storing. So we may write for example
- after ( ): storing ( ) {
  System.out.println(”Products stored”);
  }
- When one of store methods has performed, the message is written.
- Compare this to advice functions in Scheme.

Aspects (tracing)

- An aspect is like a class. It can have attributes, contructors, methods, pointcuts and advices.
- The aspect for tracing is our next example.
- public aspect TraceStored {

```
    pointcut traced ():
            call ( void *.store ()) ||
            call ( void *.retrieve ());
    before (): traced () {
          debug (”Entering: ” + thisJoinPoint)
    }
    void debug (String str) { //write to stream
    }
}
```

Tracing

- The designator is traced and the join points are before calling store or retreive methods. Then the advice runs.
- The variable thisJoinPoint contains the exposed content of the join point.
- When the program is running, the messages e.g. ”Entering call (void Grapes.store()) ” are printed.

Checking

- Under developing programs there are generally situations, where we should add some checking or print some information as we saw in the previous example.
- The example below shows, how we can define **pre-conditions**.
- Note that the object thisJoinPoint has different methods that expose something about the content exposed by join point.

Checking

- Check the arguments of the methods of the classes in myPackage.
-
```
public aspect NullChecker {
      pointcut arguments (): execution (*
myPackage.*.* (..));
      before () : arguments () {
      for ( Object arg : thisJoinPoint.getArgs()) {
                if (null == arg) {
                          throw new IllegalArgumentException
                                    (”The arg is null”);
                }
      } }
}
```

Exceptions

- In Java and C++ the logic of the code, which refers to the exceptions, is possible to separate from other code with try catch blocks.
- The aspects go even further. We can add exceptions without touching an original code.
- The following example handles exceptions according to [LL00, 418-427].
- Breaking a contract is handled as an exception.

Exceptions

-
```
public class MyContract {
      static void require(boolean pre, Object c) throws
MyContractException { //defined
      if (!pre) throw new MyContractException
(”Precondition of ”+c+” violated”);
      }
      static void ensure(boolean post, Object c)
throws MyContractException {
      if (!post) throw new MyContractException
(”Postcondition of ”+c+” violated”);
      }
}
```

Exceptions

-
```
public class Account {
      private String owner;
      private int accNo;
      private double balance;
      public Account (String owner, int accNo, double
balance) {
                this.owner = owner;
                this.accNo = accNo;
                this.balance = balance;
      }
      public void deposit (double amount) {
                balance += amount;
      }
}
```

Contract and Account in Java

-
```
import Contract;
public class Account {… //attributes
      public Account (String owner, int accNo, double
balance) {
                Contract.require (owner != null, this);
                Contract.require (accNo > 0 && balance
>= 0, this);
                //this.param = param; statements
      }
      public void deposit (double amount) {
                Contract.require (amount > 0, this);
                balance += amount;
      }
}
```

Contract and Account in Java

- Unfortunately we had to change the original Account class, when we used the Contract class.
- Aspects give us an opportunity to write an aspect AccountContract, which AscpectJ compiler can include.
- The important notion is again that the original Account class is untouchable.
- The aspect contains two pointcuts and two before methods.

Contract and Account in AspectJ

constructor

```
• aspect AccountContract {
    pointcut consCheck (Account a, String s, int n,
  double bal) : call(Account.new(String,int)) &&
  target(a) && args(s,n,bal);
   pointcut depositCheck (Account a, double  x) : call(
  void Account.deposit(double)) && target(a) &&
  args(x);
    before (Account a, String s, int n, double bal) :
  consCheck(a,s,n) {
        Contract.require (s != null, a);
        Contract.require (n>0 && bal>=0, a); }
    before (Account a, double x) : depositCheck(a,x) {
        Contract.require (x>0, a); }
  }
```

Concurrency

- The code related to concurrency handles mutual excluding of processes and allocating and releasing common resources of these processes.
- Normally the code lines concerning concurrent processes is in the same place as the other code.
- We can separate these with aspects.

Concurrency

- The example below is taken from [WBM99].
- The logic is in one file. We have a library of books and two methods addBook and numBooks.
- The query, how many books there are in the library, cannot happen at the same time as adding a new book to the library.

Concurrency

```
• public class Query {
        Hashtable books;
        int bookCount = 0;
        public void addBook (Book b, Library lib) {
                if (!books.containsKey(b)) {
                        books.put(b,lib);
                        bookCount++;
                }
        }
        public long numBooks ( ) {
                return bookCount;
        }
  }
```

Concurrency

- Mutual excluding the methods can be presented by:
- coordinator Query {
      mutex { addBook, numBooks };
  }
- When AspectJ has been developing, the authors have dropped mutex and some other reserved words out.
- The more recent situation, see [HG04].

Profiling

-
  ```
  aspect MakeSugarCounting {
        private int count = 0;
        pointcut sugarCount () : withincode ( void
  makeWine()) && call (void makeSugar());
        after () returning () : sugarCount () {
              count++;
        }
  }
  ```

  returns normally
- Count the number of callings of sugarCount method, when we are making wine.

Error logging

-
  ```
  aspect PublicErrorLogging {
        Log log = new Log ();
        pointcut publicMethodCall () :
              call (public * Grapes.* (..));
        after () throwing (Error e) : publicMethodCall ()
  && !cflow (publicMethodCall ())  {
              log.write(e);
        }
  }
  ```

  parameter
- Write to log any errors caused by public method calls in Grapes class. Eliminate calls within other method calls.

pointcut (Exception)

- pointcut ioHandler () : (within (Orange) || within (Grapes)) && handler (IOException);
- Here the join points are picked out, where the code belongs to the Orange or Grapes classes and the IOException is caught inside the code.

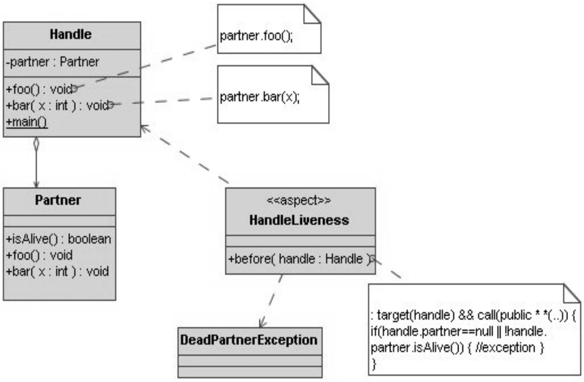
Around advice

- An around advice does not run before or after the join point but *instead* of it. The original action can be invoked by the proceed call, which is inside the around method.
- In CLOS the respective method is (call-next-method).

-
  ```
  void around (Grapes g) : target (g) && call (void
  makeSugar ()) {
  //make honey
  }
  ```
- Inside Grapes code instead of making sugar make honey.

Pointcut parameters

- From [eclipseOrg]:
- The example shows two classes Handle and Partner. Handle objects delegate their methods to their Partner objects.
- Our aspect HandleLiveness ensures that, before the delegations, the partner exists and is alive, or else it throws an exception DeadPartnerException.



```
: target(handle) && call(public * *(..)) {
if(handle.partner==null || !handle.
partner.isAlive()) { //exception }
}
```

Development and production aspects

- The aspects may be in the **development** phase. They includes then tracing and profiling. The aspects can be ignored easily, when the product is ready.
- The **production** aspects are such as extending the current class hierarchy or adding new methods to existing classes.
- The use of **inter-type declarations** in aspects makes it possible to design production aspects.
- Some speak also **reusable** aspects, which are aspects that can be applied quickly to many different situations.

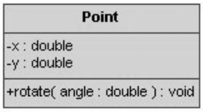
Inter-type declarations

- Aspects can declare **inter-type** features, such as declaring new attributes, contructors and methods.
- Besides they can implement new interfaces or extend new classes.
- aspect makeApple {

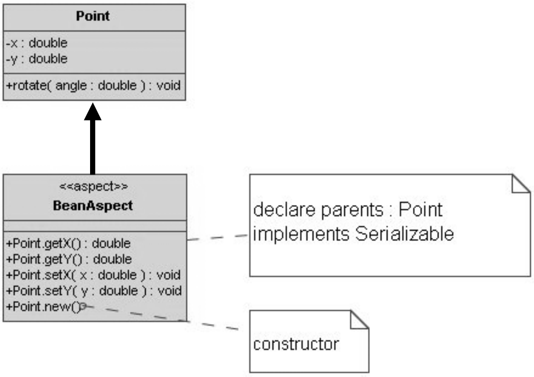
```
    private double Fruit.price; //new attribute
    declare parents: Apple extends Fruit;
    private Color Apple.color;
}
```

Bean aspect

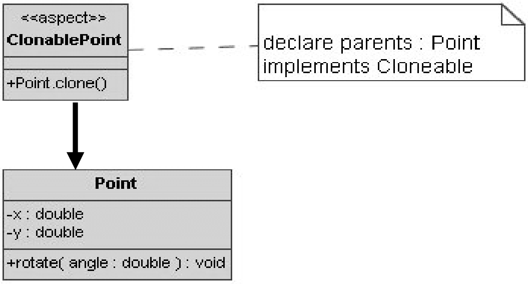
- A class is Point and it has two attributes.



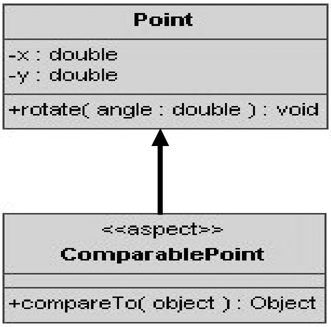
- If Point is a bean, it have to fulfill the conditions:
  - It has getter and setter methods.
  - It has a no-argument constructor.
  - It implements the interface Serializable.


Point
-x : double
-y : double
+rotate(angle : double) : void
<<aspect>>
BeanAspect
+Point.getX() : double
+Point.getY() : double
+Point.setX(x : double) : void
+Point.setY(y : double) : void
+Point.new()
declare parents : Point implements Serializable
constructor

Clonable aspect


<<aspect>>
ClonablePoint
+Point.clone()
declare parents : Point implements Cloneable
Point
-x : double
-y : double
+rotate(angle : double) : void

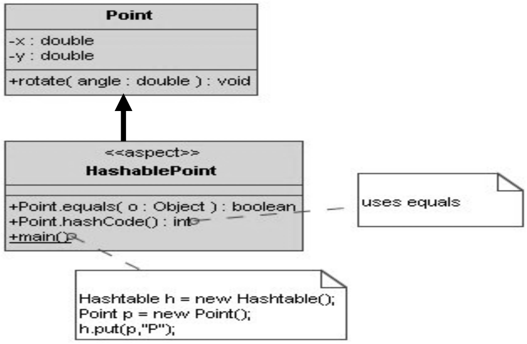
Comparable aspect


Point
-x : double
-y : double
+rotate(angle : double) : void
<<aspect>>
ComparablePoint
+compareTo(object) : Object

Comparable aspect

```
• public aspect ComparablePoint {
       declare parents : Point implements Comparable;
       public int Point.compareTo (Object object) {
              return Math.sqrt(x*x + y*y);
       }
  }
```

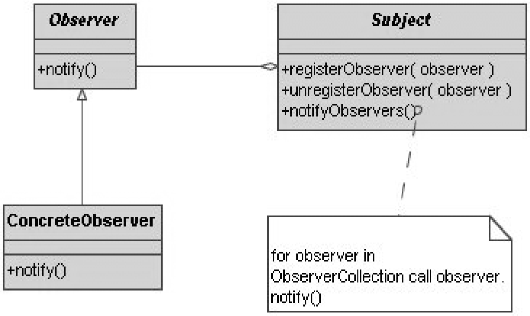
Hashable aspect



uses equals

```
Hashtable h = new Hashtable();
Point p = new Point();
h.put(p,"P");
```

Observer pattern

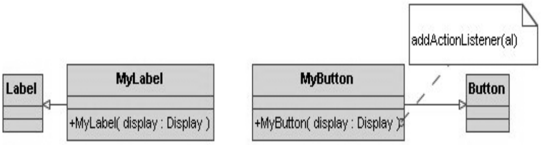


for observer in ObserverCollection call observer. notify()

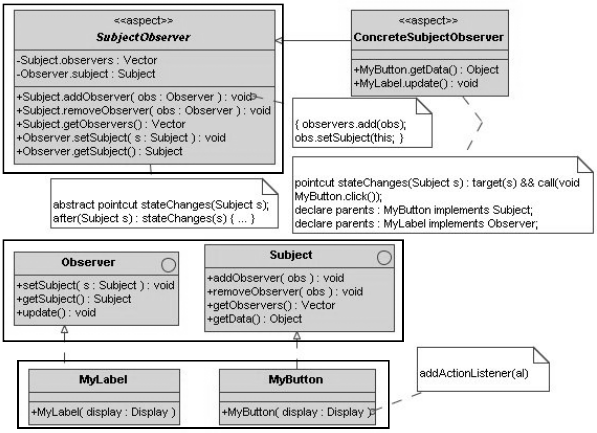
Observer pattern

- This example shows, how to apply an aspect to a design pattern.
- We have classes MyButton and MyLabel.
- When the aspect has been added, clicking a button changes a text in a label.
- MyButton and MyLabel have constructor that defines appearance of these components. Creating a button adds also listener to it.

Observer pattern



addActionListener(al)


<<aspect>>
SubjectObserver
-Subject.observers : Vector
-Observer.subject : Subject
+Subject.addObserver(obs : Observer) : void
+Subject.removeObserver(obs : Observer) : void
+Subject.getObservers() : Vector
+Observer.setSubject(s : Subject) : void
+Observer.getSubject() : Subject
<<aspect>>
ConcreteSubjectObserver
+MyButton.getData() : Object
+MyLabel.update() : void
Observer
+setSubject(s : Subject) : void
+getSubject() : Subject
+update() : void
Subject
+addObserver(obs) : void
+removeObserver(obs) : void
+getObservers() : Vector
+getData() : Object
MyLabel
+MyLabel(display : Display)
MyButton
+MyButton(display : Display)

```
{ observers.add(obs);
obs.setSubject(this; }
```

```
abstract pointcut stateChanges(Subject s);
after(Subject s) : stateChanges(s) { ... }
```

```
pointcut stateChanges(Subject s) : target(s) && call(void
MyButton.click());
declare parents : MyButton implements Subject;
declare parents : MyLabel implements Observer;
```

addActionListener(al)

Observer pattern

- Very important:
- Aspects can define interfaces, which have **non-static attributes** and **methods with code**.
- Also **abstract** aspects can have methods with bodies.

Problems with aspects

- Debugging can be difficult,because at run-time the code is not separated from the other code.
- Complex pointcuts can result in getting a loop.
- Join points with wildcards may cause unexpected results, because of creating a new method or renaming methods. We maybe do not want to apply wildcards to a new method. So we have to redefine pointcuts.

Compiling aspect files

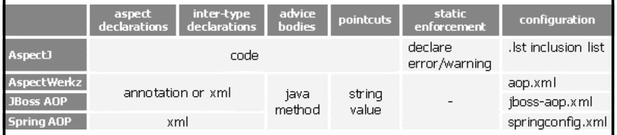
- The compiler ajc combines compilation and bytecode weaving.
- ajc HelloWorld.java Trace.java
- hello.lst
  HelloWorld.java
  Trace.java
- ajc –argfile hello.lst
- Running
- java –classpath ”.;installDir/lib/aspectjrt.jar” hello

Comparing tools

- The excellent comparing of AOP tools (by Mik Kersten) for Java can be found from the link:
- http://www-128.ibm.com/developerworks/library/j-aopwork1/
- There are four primary tools at the moment: AspectJ, AspectWerkz, JBoss AOP and Spring AOP.
- According to the latest news AspectJ and AspectWerkz projects are merging.
- The tools do not yet support refactoring nor UML views.
- I have selected several tables for the summary of these tools.

Comparing tools

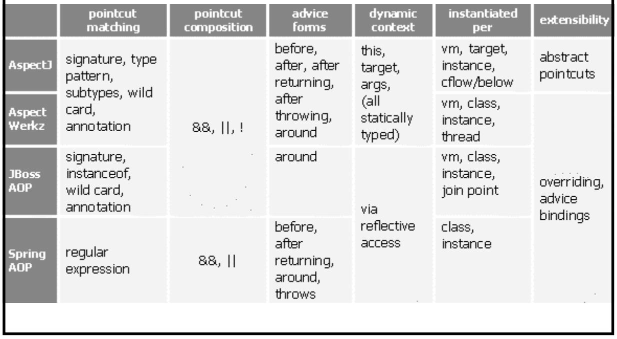
Table 1. Comparing syntax among the leading AOP tools



	aspect declarations	inter-type declarations	advice bodies	pointcuts	static enforcement	configuration
AspectJ	code				declare error/warning	.lst inclusion list
AspectWerkz	annotation or xml		java method	string value	-	aop.xml
JBoss AOP						jboss-aop.xml
Spring AOP	xml					springconfig.xml

Comparing tools

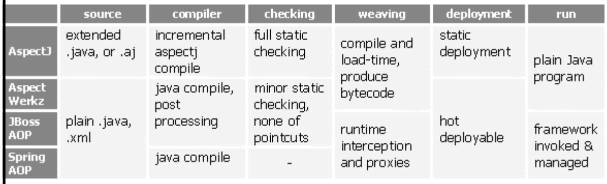
Table 2. Semantics overview of the leading AOP tools



	pointcut matching	pointcut composition	advice forms	dynamic context	instantiated per	extensibility
AspectJ	signature, type pattern, subtypes, wild card, annotation	&&, ||, !	before, after, after returning, after throwing, around	this, target, args, (all statically typed)	vm, target, instance, cflow/below	abstract pointcuts
Aspect Werkz					vm, class, instance, thread	overriding, advice bindings
JBoss AOP	signature, instanceof, wild card, annotation		around	via reflective access	vm, class, instance, join point	
Spring AOP	regular expression	&&, ||	before, after returning, around, throws		class, instance	

Comparing tools

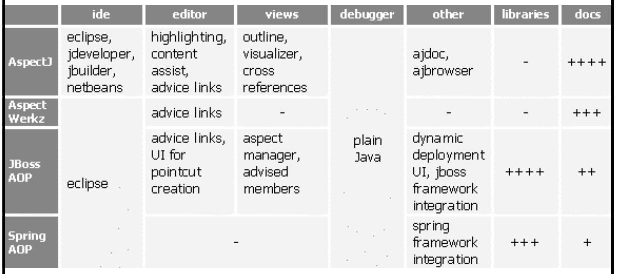
Table 3. AOP tools comparison: development environment integration



	source	compiler	checking	weaving	deployment	run
AspectJ	extended .java, or .aj	incremental aspectj compile	full static checking	compile and load-time, produce bytecode	static deployment	plain Java program
Aspect Werkz	plain .java, .xml	java compile, post processing	minor static checking, none of pointcuts		hot deployable	
JBoss AOP				runtime interception and proxies		framework invoked & managed
Spring AOP		java compile	-			

Comparing tools

Table 4. IDE support, libraries, and documentation



	ide	editor	views	debugger	other	libraries	docs
AspectJ	eclipse, jdeveloper, jbuilder, netbeans	highlighting, content assist, advice links	outline, visualizer, cross references	plain Java	ajdoc, ajbrowser	-	++++
Aspect Werkz	eclipse	advice links	-		-	-	+++
JBoss AOP		advice links, UI for pointcut creation	aspect manager, advised members		dynamic deployment UI, jboss framework integration	++++	++
Spring AOP		-			spring framework integration	+++	+

Recommended books

- **AspectJ in Action**: Practical Aspect-Oriented Programming (Paperback)
- by Ramnivas Laddad
- Paperback: 512 pages
- Publisher: Manning Publications (July 1, 2003)
- Language: English
- ISBN: 1930110936

- **Eclipse AspectJ** : Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools (Paperback)
- by Adrian Colyer, Andy Clement, George Harley, Matthew Webster
- Paperback: 504 pages
- Publisher: Addison-Wesley Professional (December 14, 2004)
- Language: English
- ISBN: 0321245873

- **AspectJ Cookbook** (Paperback)
- by Russell Miles
- Paperback: 354 pages
- Publisher: O'Reilly Media, Inc.; 1 edition (December 20, 2004)
- Language: English
- ISBN: 0596006543

AspectJ.org

AspectJ.org is a PARC project

(partially funded by DARPA under contract F30602-97-C0246)

Erik Hilsdale, Jim Hugunin, Wes Isberg,
Mik Kersten and Gregor Kiczales

AJDT Team: Adrian Colyer, Mik Kersten, Andy Clement, Julie Waterhouse Park

download the tools and docs at: http://aspectj.org

get the eclipse plug-in: http://eclipse.org/ajdt

email the team: support@aspectj.org

find more information on AOP: http://aosd.net

References

- [PARC02] OOPSLA 2002, November 4-8, 2002 Tutorial: Aspect-Oriented Programming with AspectJ™ (1.0.6).
- [Weiser84] M.Weiser. Program slicing. IEEE. Transactions on Software Engineering, SE-10(4): 352-357, 1984.
- [eclipseOrg] www.eclipse.org/aspectJ.
- [Seibel05] Seibel Peter, Practical Common Lisp, Apress (April 11, 2005), ISBN**:** 1590592395

References

- [LL00, 418-427] M.Lippert and C.V.Lopes. A study on exception detection and handling using aspect-oriented programming. In 22nd International Conference on Software Engineering (IC-SE 2000), 418-427, Limerick, Ireland, June 2000.
- [WBM99] R. J. Walker, E. L. A. Baniassad and G. C. Murphy. An initial assessment of aspect-oriented programming. In 21th International Conference on Software Engineering (ICSE´99): 120-130, Los Angeles, California, May 1999.