

AspectJ language

Corso 3° livello



SoftEng
<http://softeng.polito.it>

Paolo Falcarin

2007

AspectJ

- **AspectJ is:**
 - ♦ an aspect-oriented extension to Java that supports general-purpose aspect-oriented programming
- **freely available implementation:**
 - ♦ the only full, stable and widely used one;
 - ♦ Open Source compiler that creates standard Java bytecode
- **It includes a set of tools**
 - ♦ Aspect-aware debugger and documentation tool
 - ♦ Visual aspect browser
 - ♦ Integration with popular IDEs (Eclipse, Jbuilder, Emacs, NetBeans)

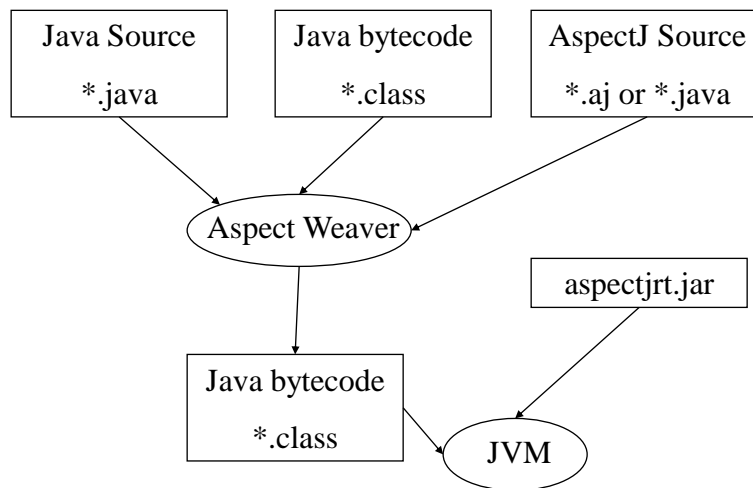
SoftEng
<http://softeng.polito.it>

Paolo Falcarin

Aspect Weaver (Compiler)

- Combines the various aspect fragments.
- Uses a set of rules to automate the process.
- Design reflected in the weaving rules.
- Optimizes the integration code.
- Simplifies debugging and testing.
- Errors about dangling or parallel aspects.

AspectJ Weaver



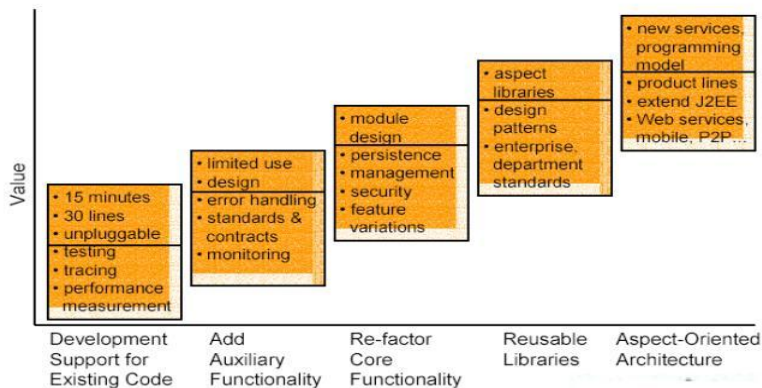
AspectJ adoption

- Rapid growth of AspectJ tool downloads
- User mailing list
 - ◆ more than 1000 active members
- Used as development support in different companies
 - ◆ IBM, Cigital Labs,...
- Companies research projects:
 - ◆ IBM, Siemens, SAP, SUN, ...
- Presentations at best conferences:
 - ◆ OOPSLA, SIGS, ICSE, TOOLS, ECOOP, AOSD

SoftEng
http://softeng.polito.it

Paolo Falcarin

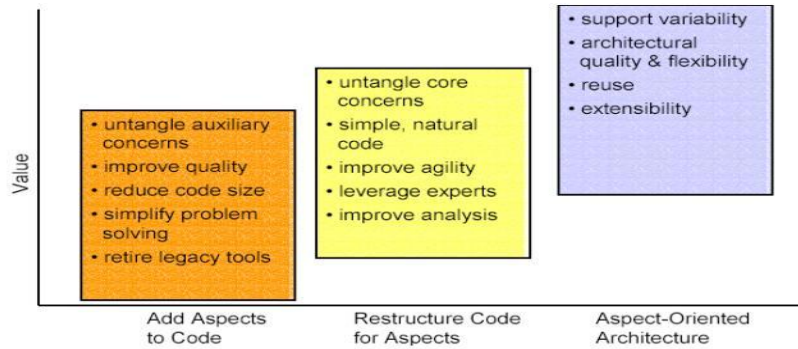
AspectJ adoption curve



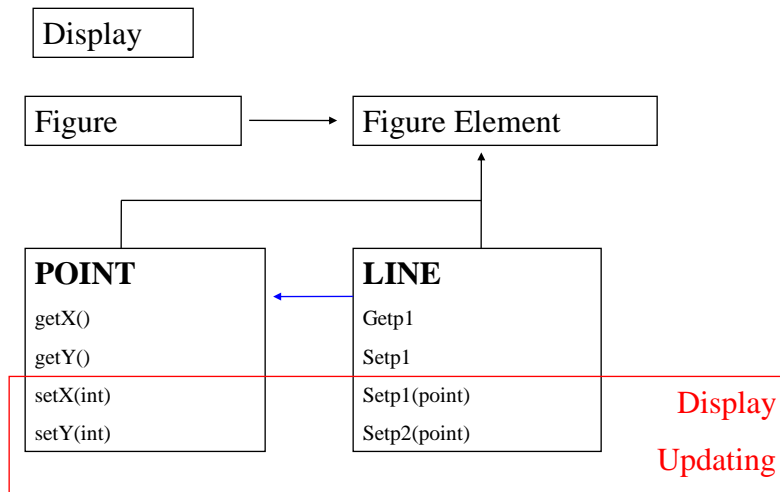
http://softeng

Paolo Falcarin

Expected benefits



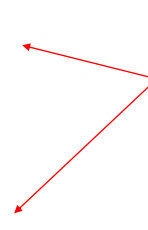
Basic example



Basic example without AspectJ

```
class Line {  
    private Point p1, p2;  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
    void setP1(Point p1) {  
        this.p1 = p1;  
        Display.update();  
    }  
    void setP2(Point p2) {  
        this.p2 = p2;  
        Display.update();  
    }  
}
```

update calls are tangled through the code =>
"what is going on" is less explicit



} SoftEng
http://softeng.polito.it

Paolo Falcarin

Basic mechanisms

Fundamental concept : **join point**

- identifiable points in the execution of Java programs

4 small additions to Java:

- **pointcuts**

- ♦ Language constructs to pick out join points and values at those points
- ♦ Primitive and user-defined pointcuts; composable

- **advice**

- ♦ additional action to take at join points in a pointcut

- **inter-class declarations** (= "open classes")

- **aspect**

- ♦ a modular unit of crosscutting behavior
- ♦ comprised of advice, inter-class, pointcut, field constructor and method declarations

SoftEng
http://softeng.polito.it

Paolo Falcarin

Join point

- Well-defined points in a program's execution
- AspectJ makes these join points available:
 - ♦ Method call and execution
 - ♦ Constructor call and execution
 - ♦ Read/write access to a field
 - ♦ Exception handler execution
 - ♦ Object and class initialization execution
- The **join-point model** is the set of available join-points used by the **AOP platform** (i.e. AspectJ)

AspectJ join-points

1. Method call
2. Method execution
3. Constructor call
4. Constructor execution
5. Initializer execution
6. Static initializer execution
7. Object pre-initialization
8. Object initialization
9. Field reference
10. Field assignment
11. Exception Handler execution

Pointcut designator

- Definition of a collection of join points
- Identifies particular join points by filtering out subset of all join points
- 2 ways:
 - ♦ It can filter out set of specific method calls
 - ♦ It can use wildcards to filter out all method calls with a common token

Primitive pointcut

- A pointcut is a means of identifying join-points
 - A pointcut is a kind of predicate on join points that:
 - ♦ can match or not match any given join point
 - ♦ optionally, can pull out some of the values at that join point
- call** (void Line.setP1(Point))
- ♦ matches if the join point is a method call with this signature

A simple aspect

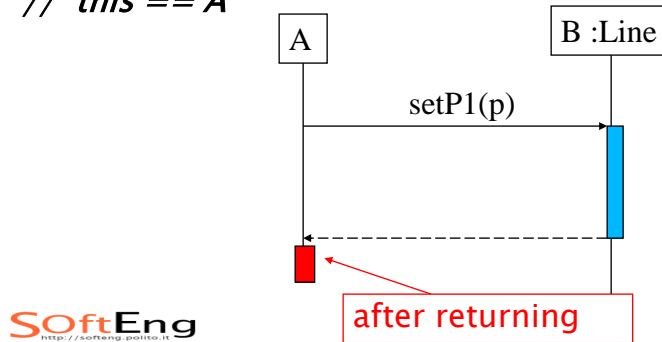
```
aspect DisplayUpdating {  
    pointcut move(): call(void Line.setP1(Point)) ||  
                    call(void Line.setP2(Point));  
    after returning: move() {  
        Display.update();  
    }  
}
```

Advice declarations

- Used to define additional code to run at join points
- **Before advice** – runs at the moment join point is reached, before method runs
- **After advice** – runs at the moment control returns through join point, just after method
- **Around advice** – runs when join point is reached and has control over whether method itself runs at all

after returning call

```
pointcut move(): call (void Line.setP1(Point)) ||  
              call (void Line.setP2(Point));  
after returning : move() {  
    <advice code: runs after each move>  
    // this == A  
}
```

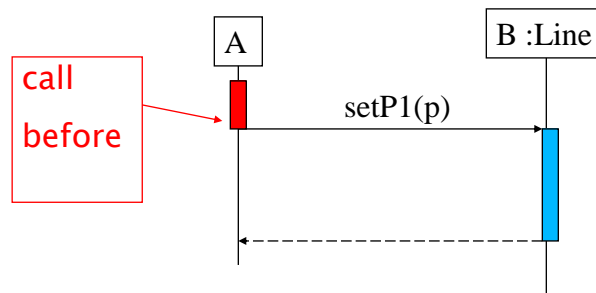


SoftEng
http://softeng.polito.it

Paolo Falcarin

before call

```
pointcut move(): call (void Line.setP1(Point)) ||  
              call (void Line.setP2(Point));  
before() : move() {  
    <advice code: runs after each move>  
    // this == A  
}
```

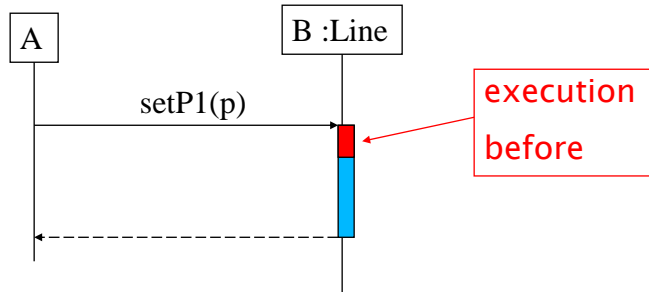


SoftEng
http://softeng.polito.it

Paolo Falcarin

before execution

```
pointcut move(): execution (void Line.setP1(Point)) ||  
                execution (void Line.setP2(Point));  
before() : move() {  
    <advice code: runs after each move>  
    // this == B  
}
```



After Advice types

After advice can be of three types

- **after returning** – runs when control returns to join point, just after method ends normally
- **after throwing** – runs when control returns to join point, after method ends throwing an exception
- **after** – runs when control returns to join point, after method ends, either normally or throwing an exception

Wildcards in pointcuts (1)

“*” is wild card

“..” is multi-part wild card

within(Point)

within(graphics.geom.Point)

within(graphics.geom.*) //any type in graphics.geom

within(graphics..*) //any type in any sub-package of graphics

call(void Point.setX(int))

call(public * Point.*(..)) //any public method on Point

call(public * *(..)) //any public method on any type

SoftEng
http://softeng.polito.it

Paolo Falcarin

Wildcards in pointcuts (2)

call(void Point.getX())

call(void Point.getY())

call(void Point.get*())

call(* get*()) //any getter method

call(new(..)) //any constructor

SoftEng
http://softeng.polito.it

Paolo Falcarin

Pointcuts

- Field access pointcuts
 - ♦ **get**(PrintStream System.out)
 - ♦ **set**(int MyClass.x)
- Exception handling pointcuts (*catch* execution)
 - ♦ **handler**(RemoteException)
 - ♦ handler(IOException+)
 - ♦ handler(CreditCard*)
- Object and class initialization pointcuts
 - ♦ **staticinitialization**(MyClass+)

Context-based Pointcuts

- **this**(<type name>)
 - // any join point at which currently executing object is an instance of <type name>
- **target**(MyClass) // match type of method call's target
- **args**(String, .. ,int) // match order & type of arguments
- **args**(RemoteException) // type of argument or handler
- You can expose context in pointcuts
 - pointcut publicCardAmount (CreditCard card, Money amount):
 - execution(public * CreditCardManager.*(..) && args(card, amount);

Pointcuts and lexical scope

Inside lexical scope of class or method:

- **within**(<type name>)
 - any join point at which currently executing code is defined within type name
- **withincode**(<method/constructor signature>)
 - any join point at which currently executing code is the specified method or constructor

▪ Example:

- ♦ **within**(MyClass) // of class
- ♦ **withincode**(* MyClass.myMethod(..)) // of method

SoftEng
http://softeng.polito.it

Paolo Falcarin

Dynamic Pointcuts

- Dynamic pointcuts matches different sets of joinpoints at different time during execution
- Control-flow based pointcuts
 - Note: If a() calls b(), then b() is inside a()'s control flow
 - ♦ **cflow**(call(* MyClass.myMethod(..)): picks all j.p. in the control flow originated by the inner call pointcut
 - ♦ **cflowbelow**(call(* MyClass.myMethod(..)) : picks all j.p. in the control flow originated by the inner call pointcut, excluding the initial call join-point
- Conditional test pointcuts
 - ♦ **if** (EventQueue.isDispatchThread()) : depend on condition which may depend on data

SoftEng
http://softeng.polito.it

Paolo Falcarin

The special value *thisJoinPoint*

- available in any advice
- introspective subset of reflection consistent with Java
- Extract signature of method
Signature s = *thisJoinPoint*.getSignature()
- Extract actual parameters values
Object[] args = *thisJoinPoint*.getArgs();
- Extract other info

ThisJoinPoint

A useful reflection-like feature, can provide:

- the kind of join point that was matched
- the source location of the current join point
- normal, short and long string representations of the current join point
- actual argument(s) to the method / field of the join point
- signature of the method or field of the current join point
- the target object
- the currently executing object
- a reference to the static portion of the object holding the join point; also available in *thisJoinPointStaticPart*

Advice

- Defines the code to run, and when to run it
- Based on pointcuts, for example `move()`:
after(): `move()` {
 System.out.println("A figure element moved."); }

- Before and After advice

```
before() : call(public * MyClass.*(..)) {  
    System.out.println("Before: " + thisJoinPoint + " " +  
        System.currentTimeMillis()); }  
after() : call(public * MyClass.*(..)) {  
    System.out.println("After: " + thisJoinPoint + " " +  
        System.currentTimeMillis()); }
```

Around Advice

- Around advice
 - ◆ Surrounds join point, can decide if computation will happen, can modify arguments

```
void around(Connection conn) :  
    call(Connection.close()) && target(conn) {  
        if (enablePooling) {  
            connectionPool.put(conn);  
        } else {  
            proceed();  
        }  
    }
```

Precedence rules

- Multiple pieces of advice may apply to the same join point. In such cases, the resolution order of the advice is based on advice precedence
- If the two pieces of advice are defined in
 - ♦ different aspects, then there are three cases:
 - dominates, sub-aspect, undefined.
 - ♦ in the same aspect, then there are two cases:
 - Textual order: After–reverse and other–normal.

Aspect with many advices

- If the two pieces of advice are defined in the same aspect, then there are two cases:
 - ♦ If either are after advice, then the one that appears later in the aspect has precedence over the one that appears earlier.
 - ♦ Otherwise, then the one that appears earlier in the aspect has precedence over the one that appears later.

Different aspect rules

- If aspect A is declared such that it **dominates** aspect B, then all advice defined in A has precedence over all advice defined in B.
- Otherwise, if aspect A is a sub–aspect of aspect B, then all advice defined in A has precedence over all advice defined in B.
 - ♦ So, unless otherwise specified with a **dominates** keyword, advice in a sub–aspect dominates advice in a super–aspect.
- Otherwise, if two pieces of advice are defined in two different aspects, it is undefined which one has precedence.

Circularity

These rules can lead to circularity, such as:

```
aspect A {  
    before(): execution(void main(String[] args)) {}  
    after(): execution(void main(String[] args)) {}  
    before(): execution(void main(String[] args)) {} }
```

Such circularities will result in errors signaled by the compiler.

AOP advantages

- Reuse
 - ◆ Aspects can often be reused in other programs with only slight modifications
- Adaptability
 - ◆ Easy to quickly adapt a change in an entire concern
- More comprehensible system
 - ◆ Separation of aspects makes it easier to quickly see and understand concerns

AOP advantages

- Explicit design decisions:
 - ◆ Visible at code level
 - ◆ Easier to modify
- Locality
 - ◆ Fixed part in abstract aspects
 - ◆ Variable part concrete aspects
- Reuse abstract aspects
- Aspects are easy to enable / disable

Applications of AOP

- Production aspects
 - ◆ Implement functionality intended to be included in shipping applications
 - Display refreshing, synchronization policies, security, etc...
- Development aspects
 - ◆ Facilitate debugging, testing, and performance tuning of applications
 - Tracing, error messages, etc...
- Optional aspects
 - ◆ Additional features included when needed
 - thread pooling, caching, ...

Aspect vs Class

| Question | Class | Aspect |
|-----------------------------|-------|--------|
| ▪ Can be instanciated? | Yes | No |
| ▪ Can extend other classes? | Yes | Yes |
| ▪ Can implement interfaces? | Yes | Yes |
| ▪ Can extend other aspect? | No | Yes * |

*: Only abstract aspects can be extended

Conclusions

- AOP is a strong complement to OOD
 - ♦ Separation of concerns for unrelated aspects
 - ♦ Less code, more modular, easier to modify
 - ♦ A lot of practical uses
 - ♦ A lot of hype
- AspectJ is the primary implementation today
 - ♦ Many features, good tools and active support
 - ♦ Yet the entire platform is still in 'beta version'
- A good tool, during development

Links

- Mailing list: users@aspectj.org
- Download tools and docs at: <http://eclipse.org/aspectj>
- Get the eclipse plug-in: <http://eclipse.org/ajdt>
- Find more information on AOP: <http://aosd.net>

Other resources on AOP :

- Communications of the ACM, October 2001 – Volume 44, Number 10.
- AOSD international conference every year on March.
- Transactions on AOSD (Springer) since 2006.