Programming in AspectJ

COMP6209: Automated Code Generation

Julian Rathke
jr2@ecs.soton.ac.uk
Background Information

• The AspectJ Team: *Getting Started with AspectJ.*
• The AspectJ Team: *The AspectJ Language.*
  – original AspectJ documentation, both available via
    http://eclipse.org/aspectj/docs.php


Counting Method Calls

Implement the following concern:

*The FigureEditor shall count the number of times its methods are called and print this before termination.*

```java
package FigureEditor;
...

class Line implements FigureElement {
    private Point p1, p2;
    Point getP1() { return p1; };
    Point getP2() { return p2; };
    void setP1(Point p1) { this.p1=p1 ];
    void setP2(Point p2) { this.p2=p2 };
    void moveBy(int dx, int dy) {...};
};

class Point implements FigureElement {
    private int x=0, y=0;
    int getX() { return x; };
    int getY() { return y; };
    void setX(int x) { this.x=x };;
    void setY(int y) { this.y=y };
    void moveBy(int dx, int dy) {...};
};
```
Counting Method Calls

Implement the following concern:

The FigureEditor shall count the number of times its methods are called and print this before termination.

```java
public aspect CountCalls {
    int count = 0;

    before(): call(* FigureEditor..*()) {
        count++;
    }

    after(): execution(public static * main(..)) {
        System.out.println("count = " + count);
    }
}
```
Counting Method Calls

Implement the following concern:

The FigureEditor shall count the number of times its methods are called and print this before termination.

```java
public aspect CountCalls {
    int count = 0;

    before(): call(* FigureEditor..*(..)) {
        count++;
    };

    after(): execution(public static * main(..)) {
        System.out.println("count = " + count);
    };
}
```
List of basic pointcut kinds

- method execution
- method call
- constructor execution
- constructor call
- object pre-initialization
- object initialization
- static initialization
- field set
- field get
- handler execution
- advice execution
Method and field pointcuts allow signature-based access.

Methods and Constructors

\texttt{call}(Signature) \quad \text{every call to any method or constructor matching \textit{Signature} at the call site.}

\texttt{execution}(Signature) \quad \text{every execution of any method or constructor matching \textit{Signature}.}

Fields

\texttt{get}(Signature) \quad \text{every reference to any field matching \textit{Signature}.}

\texttt{set}(Signature) \quad \text{every assignment to any field matching \textit{Signature}.}

(www.eclipse.org/aspectj/doc/released/progguide/quick.html)
Method and field pointcuts allow wildcards in Java signatures.

- method signature grammar (similar for fields)
  \[
  \text{Signature} \rightarrow \text{Modifier}^* \text{ Type}? \text{ Class}? \text{ Name} \\
  \quad \text{“(“ Parameters “)“} (\text{“throws”} \text{ Exception})?
  \]

- each syntax element can be
  - the corresponding Java syntax element
    \[
    \text{Modifier} \rightarrow \text{public} | \text{private} | \ldots
    \]
  - a wildcard
    \[
    \text{Modifier} \rightarrow \ast
    \]
Method and field pointcuts allow wildcards in Java signatures. (cont’d)

- AspectJ uses different wildcards:
  - plain (*)
    - matches any Java syntax elements in the same context
    - `public * *(int)` - any public method with one integer argument
    - `** **(*, *)` - any method with two arguments
  - lexical (*)
    - matches names as in regular expressions
    - `* *set*(int)` - any method with one integer argument and a name that contains “set”
    - `* *.set*(int)` - any toplevel method with one integer argument and a name that starts with “set”
Method and field pointcuts allow wildcards in Java signatures. (cont’d)

• AspectJ uses different wildcards:
  – list ellipsis (..)
    ▶ matches arbitrary parameter lists
    ▶* * *(int, ..)  – any method with any number of arguments (but at least one int)
    ▶ matches arbitrary package / class name lists
    ▶* swing..*(_:..)  – any method anywhere in the swing-package, with any number of arguments
  – sub-type (+)
    ▶ matches arbitrary sub-types of the given base type
    ▶ Figure+.new()  – any constructor for Figure or any of its sub-classes
Pop Quiz

1. Describe in words what is matched by the pointcut descriptor
   \texttt{swing..*add*Listener(EventListener+)}
2. What are the differences (if any) between these class patterns?
   \texttt{swing..*Model}
   \texttt{swing.*.*Model}
   \texttt{swing..*.Model}
call and execution can behave differently with inheritance.

- **call** and **execution** use different contexts:
  - **call** uses the call site
  - **execution** uses the declaration site

```java
class A {
    void m() {...}
}
class B extends A {}
class Main {
    void main() {
        A a = new B();
        B b = new B();
        a.m();
        b.m();
    }
}
```

```java
aspect Executions {
    after(): execution(* A.m())
    matches {...}
    after(): execution(* B.m())
    never matches
    pattern refers to class where m is declared
}
```
call and execution can behave differently with inheritance. (cont’d)

- **call** and **execution** use different contexts:
  - **call** uses the call site
  - **execution** uses the declaration site

```java
class A {
    void m() {...}
}
class B extends A {}
class Main {
    void main() {
        A a = new B();
        B b = new B();
        a.m();
        b.m();
    }
}

aspect Calls {
    after(): call(* A.m()) {
    {...}
    }
    after(): call(* B.m()) {
    {...}
    }
}
```

invocation `a.m()` is matched only by `call A.m()`
Call and execution can behave differently with inheritance. (cont’d)

- **call** and **execution** use different contexts:
  - **call** uses the call site
  - **execution** uses the declaration site

```java
class A {
    void m() {...};
}
class B extends A {}

class Main {
    void main() {
        A a = new B();
        B b = new B();
        a.m();
        b.m();
    }
}

aspect Calls {
    after(): call(* A.m()) {...}
    after(): call(* B.m()) {...}
}
```

Invocation b.m() is matched by both call B.m() and call A.m()
call and execution can behave differently with inheritance. (cont’d)

- **call** and **execution** use different contexts:
  - **call** uses the call site
  - **execution** uses the declaration site

```java
class A {
    void m() {...}
}
class B extends A {}
class Main {
    void main() {
        A a = new B();
        B b = new B();
        a.m();
        b.m();
        // call(* B.m()) generates a warning that a.m() invocation will not match
    }
}
```

```java
aspect Calls {
    after(): call(* A.m())
    {...}
    after(): call(* B.m())
    {...}
}
```
call and execution can behave differently with inheritance. (cont’d)

• call and execution use different contexts:
  – call uses the call site
  – execution uses the declaration site

• Rule of Thumb:
  “If you want to pick a join point that runs when an actual piece of code runs, use execution, but if you want to pick one that runs when a particular signature is called, use call.”
List of basic pointcut kinds

- method execution
- method call
- constructor execution
- constructor call
- object pre-initialization
- object initialization
- static initialization
- field set
- field get
- handler execution
- advice execution

read up on these in the guide [www.eclipse.org/aspectj](http://www.eclipse.org/aspectj)
Counting Method Calls per Object

Implement the following concern:

The FigureEditor shall count the number of times its methods are called for each FigureElement-object and print this before termination.

```java
public aspect CountCallsPerObject {
    int FigureElement.count = 0;

    before(FigureElement fe): FigureElement class. call(* FigureEditor..*(..)) && target(fe) {
        fe.count++;
    }
    ...
}
```
Counting Method Calls per Object

Implement the following concern: 

*The FigureEditor shall count the number of times its methods are called for each FigureElement-object and print this before termination.*

```java
public aspect CountCallsPerObject {
    int FigureElement.count = 0;

    before(FigureElement fe):
        call(* FigureEditor..*(...)) && target(fe) {
            fe.count++;
        }

    ... // advice parameters
    ...
        • bound by pointcut descriptor
        • pass information into advice code
```
State-based pointcuts allow instance checks and context access.

**Instanceof checks**

`this(TypeOrID)` matches when the current object (i.e., the receiver of the currently executing method) is an instance of the specified type.

`target(TypeOrID)` matches when the target object is an instance of the specified type.

`args(TypeOrID, …)` matches when the arguments are instances of the specified types.

Note: `target` is the object to which the current join point transfers control; it may be different to `this` at a method call join point.
State-based pointcuts allow instance checks and context access. (cont’d)

Instanceof checks

\textbf{this}(TypeOrID) \quad \text{matches when the current object (i.e., the receiver of the currently executing method) is an instance of the specified type.}

\textbf{target}(TypeOrID) \quad \text{matches when the target object is an instance of the specified type.}

\textbf{this} \ vs. \ \textbf{target}:

\begin{align*}
\text{before}(&\text{Line l}): \text{call}(\ast \text{ Line.}\ast((..)) \&\& \text{this(1)} \\
\Rightarrow & \text{“calls to Line-methods sent from a Line-object”} \\
\text{before}(&\text{Line l}): \text{call}(\ast \text{ Line.}\ast((..)) \\
\Rightarrow & \text{“calls to Line-methods”}
\end{align*}
State-based pointcuts allow instance checks and context access. (cont’d)

- *TypeOrId* argument must have defined type
  - type pattern
    ▶ same wildcards as in method signatures
    ▶ `target(FigureElement+)`
  - identifier introduced by advice
    ▶ “normal” Java parameter declaration
      - subtyping rules apply: implicit instanceof-test
      - special handling for `Object`: also matches base types
        ▶ `before(FigureElement fe): target(fe)`
  - identifiers are bound to actual values at joinpoint
    - values accessible in advice
Return values can be accessed via \texttt{after} / \texttt{returning}.

- return values are not yet available when identifiers are bound at joinpoint…
- … but only after method execution
- special syntax form in \texttt{after}-advice

\begin{verbatim}
  after(Line l) returning(Point p):
    call__(* *.get*(*)) && args(l) {
      System.out.println(
        "returning Point " + p.toString()
        " for Line " + l.toString());
    }
\end{verbatim}

- similar for exceptions (\texttt{throwing})
- arguments optional – check for status only
Tracing Method Calls

Implement an aspect that traces all calls to user-defined methods (i.e., not from java..) and indicates the nesting level.

public class Test {
    static int fib(int n) {
        if (n<=1)
            return n;
        else
            return fib(n-1) + fib(n-2);
    }
    static void report(int n,int r) {
        write("fib(\"+n+\")=\"+r);
    }
    public static void main(String[] a) {
        report(4,fib(4));
    }
    ...
}

void Test.main(String [])
>void Test.report(int, int)
>>int Test.fib(int)
>>>>int Test.fib(int)
>>>>>int Test.fib(int)
>>>>>>int Test.fib(int)
>>>>>>int Test.fib(int)
>>>>>>int Test.fib(int)
>>>void Test.write(String)
fib(4)=3
Tracing Method Calls

Implement an aspect that traces all calls to user-defined methods (i.e., not from java..) and indicates the nesting level.

```java
public aspect Trace {
    String prefix = "";

    before(): call( * *(..)) && !within(java..*) {
        write(prefix+thisJoinPoint.getSignature());
        prefix = "\" + prefix;
    }

    after(): call( * *(..)) && !within(java..*) {
        prefix = prefix.substring(1);
    }
}
```

- program text-based pointcut
- predefined variable of type `org.aspectj.lang.JoinPoint`
- joinpoint combinators
Program text-based pointcuts expose static program scopes.

Program text-based pointcuts

within(\textit{Type}) matches when the code executing is defined within the declaration of one of the matching types.

withincode(\textit{Method}) matches when the code executing is defined within the declaration of any method matching the signature.

Note: \textit{within} and \textit{withincode} also match any join points that are associated with code in nested classes, or a method’s local or anonymous types.
org.aspectj.lang contains meta-classes to expose the pointcut.

package org.aspectj.lang;

public interface JoinPoint {
    String toString();
    Object getThis();       // current object
    Object getTarget();     // receiver of call, field access
    Object[] getArgs();     // arguments
    StaticPart getStaticPart();
}

interface StaticPart {
    Signature getSignature();  // type of method, field
    SourceLocation getSourceLocation();
    String getKind();          // 11 different kinds
}
Expression pointcuts allow arbitrary context checks.

- can check for every boolean expression
  \[
  \text{if} (\text{Expression})
  \]
  - \text{Expression} should be side-effect free
- often combined with \text{JoinPoint} interface
- allow to modify or complement AspectJ behavior:
  - modified (name) matching
    \[
    \text{if} (\text{test(thisJoinPoint.getTarget().}
    \backslash
    \text{getClass().getName())})
    \]
  - check arguments
    \[
    \text{args(l)} \&\& \text{if(l.length > 10)}
    \]
  - check properties of the code via Java reflection
Tracing Method Calls (revisited)

Implement an aspect that traces all calls to user-defined methods (i.e., not from java..) and indicates the nesting level.

```java
public aspect Trace {
    String prefix = "";

    before(): call(* *(..)) && !within(java..*) {
        write(prefix+thisJoinPoint.getSignature());
        prefix = ">" + prefix;
    }

    Do we need before and after with the same pointcut descriptors?

    after(): call(* *(..)) && !within(java..*) {
        prefix = prefix.substring(1);
    }
}
```
Tracing Method Calls (revisited)

Implement an aspect that traces all calls to user-defined methods (i.e., not from java..) and indicates the nesting level.

public aspect Trace {
    String prefix = "";

    around(): call(* *(..))
        && !within(java..*) {
            write(prefix+thisJoinPoint.getSignature());
            prefix = ">" + prefix;

            prefix = prefix.substring(1);
        }
}
Tracing Method Calls (revisited)

Implement an aspect that traces all calls to user-defined methods (i.e., not from java..) and indicates the nesting level.

```java
public aspect Trace {
    String prefix = "";
    Object around(): call(* *(..))
        return value && !within(java..*) {
            write(prefix+thisJoinPoint.getSignature());
            prefix = "\>" + prefix;
            Object result = proceed();
            prefix = prefix.substring(1);
            return result;
        }
}
```

advice code runs instead of joinpoint! Return value can call original joinpoint
around-advice allows to change the base program’s behavior completely.

- **before / after** only add behavior to joinpoint
  - *incremental* changes

- **around** replaces original joinpoint behavior
  - *invasive* changes
    - only very weak correctness guarantees
      - advice return type must be compatible with joinpoint return type
      - AspectJ boxes / unboxes primitive types
        (i.e., will match against **object**)
proceed allows around to invoke the original joinpoint.

- **proceed()** invokes original joinpoint behavior
  - uses original arguments to joinpoint
  - returns original joinpoint result
  - under control of advice code

- **proceed()** can have arguments
  - same argument types as advice
  - can be used to change joinpoint context

```java
aspect A {
    int around(int i): call(int c.*(*, int))
      && args(Object, i) {
        int j = proceed(i*2)
        return j/2;
    }
}
```
Example: failure handling

Implement an aspect that retries `getReply()` requests `N` times after a `RemoteException` has occurred.

```java
import java.rmi.RemoteException;
public class RemoteService {
    public static int getReply() throws RemoteException {
        if (Math.random() > 0.25)
            throw new RemoteException("failure");
        System.out.println("Replied");
        return 5;
    }
}
```
Example: failure handling (cont’d)

```java
aspect FailureHandling {
    final int N = 3;
    int around() throws RemoteException:
        call(int *.*.getReply()
            throws RemoteException) {
    int retry = 0;
    while (true) {
        try {return proceed();}
        catch(RemoteException ex) {
            System.out.println("Encountered " + ex);
            retry++;
            if (retry == N) throw ex;
            System.out.println("Retrying...");
        }
    }
}
```
Counting Method Calls (again)

Implement an aspect that counts for each outer quicksort-call the number of internal method calls.

```java
public class QuickSort {
    public static void quicksort(double[] a, int left, int right) {
        if (right <= left) return;
        int i = partition(a, left, right);
        quicksort(a, left, i-1);
        quicksort(a, i+1, right);
    }

    private static int partition(double[] a, int left, int right) {
        int i = left - 1; int j = right;
        while (true) {
            while (a[++i] < a[right]);
            while (a[right] < a[--j]) if (j == left) break;
            if i >= j) break;
            My.swap(a, i, j);
        }
        My.swap(a, i, right);
        return i;
    }
}
```

partitions=18
swaps=85
Counting Method Calls (again) (cont’d)

What is wrong with this solution?

• what happens the second time quicksort is called?
• what happens in the recursion?
• what happens if somebody else calls My.swap()?

⇒ need to take control flow into account!

```java
public aspect QSProfile {
    private int p=0; private int s=0;
    before(): call(* partition(..)) {p++;}
    before(): call(* My.swap(..)) {s++;}
    before(): call(* quicksort(..)) {p=0; s=0;}
    after(): call(* quicksort(..)) {
        write("partitions="+p+"\nswaps="+s+"\n");
    }
}
```
public aspect QSProfile {
    private int p=0; private int s=0;
    pointcut qs(): call(* quicksort(..));
    pointcut init(): qs() && !cflowbelow(qs());
    before(): call(* partition(..)) {p++;}
    before(): call(* My.swap(..)) && cflow(qs()) {s++;}
    before(): init() {p=0; s=0;}
    after() : init() {
        write("partitions="+p+"\nswaps="+s+"\n");
    }
}
Control flow-based pointcuts expose dynamic program scopes.

Control flow-based pointcuts

cflow($PCD$) matches all joinpoints in the control flow of any joinpoint $P$ that matches the given pointcut descriptor $PCD$.

cflowbelow($PCD$) matches all joinpoints in the control flow of any joinpoint $P$ that matches the given pointcut descriptor $PCD$, except $P$ itself.
Control flow-based pointcuts expose dynamic program scopes. (cont’d)

pointcut qs(): call(* quicksort(..));
before(): qs() {
    write("call-type: arbitrary call");
}

before(): qs() && cflow(qs()) {
    write("call-type: arbitrary call");
}

before(): qs() && !cflowbelow(qs()) {
    write("call-type: top-level call");
}

before(): qs() && cflowbelow(qs()) {
    write("call-type: recursive call");
}
Control flow-based pointcuts expose dynamic program scopes. (cont’d)

- combination of program text-based and control flow-based pointcuts can express fine-grained control

```java
public class A { match public aspect Test {
    void c() {
        always pointcut w(): call(write(*));
        write("c");
    }
    void b() {
        c(); write("b");
    }
    void a() {
        b(); write("a");
    }
} }
```
Control flow-based pointcuts expose dynamic program scopes. (cont’d)

- combination of program text-based and control flow-based pointcuts can express fine-grained control

```java
public class A {
    void c() {
        write("c");
    }
    void b() {
        c(); write("b");
    }
    void a() {
        b(); write("a");
    }
}

public aspect Test {
    pointcut w() : call(write(*));
    after(): w() {...}
    after(): w() && withincode(a()) {...}
}
```
Control flow-based pointcuts expose dynamic program scopes. (cont’d)

• combination of program text-based and control flow-based pointcuts can express fine-grained control

```java
public class A {
    void c() {
        write("c");
    }
    void b() {
        c();
        write("b");
    }
    void a() {
        b();
        write("a");
    }
}

public aspect Test {
    pointcut w(): call(write(*));
    after(): w() {...}
    after(): w() && withincode(a())
        {...}
    after(): w() && cflow(call(a()))
        {...}
}
```

matches always
match only via calls to a
Control flow-based pointcuts expose dynamic program scopes. (cont’d)

• combination of program text-based and control flow-based pointcuts can express fine-grained control
Control flow-based pointcuts expose dynamic program scopes. (cont’d)

• combination of program text-based and control flow-based pointcuts can express fine-grained control

• `cflow` does not distribute over `&&`:

\[ cflow(\text{call}(a()) && \text{call}(b())) \neq \emptyset \neq cflow(\text{call}(a())) && cflow(\text{call}(b())) \{b^*\} \]

• `cflow` distributes over `||`

```java
class A {
    void c() {
        write("c");
    }
    void b() {
        c(); write("b");
    }
    void a() {
        b(); write("a");
    }
}
```
Pop Quiz

1. Write an aspect that calls the `redraw`-method on `FigureEditor.Display` after each top-level call to `moveBy`.

```java
class Line implements FigureElement {
    ...
    public void moveBy(int dx, int dy) {
        getP1().moveBy(dx, dy);
        getP2().moveBy(dx, dy);
    }
}
```

don’t redraw after these calls!

a) use *dynamic* pointcuts to exclude nested calls
b) use *static* pointcuts to exclude nested calls
Control flow-based pointcuts access the most recent call context.

```java
public class Fact {
    static int fact(int x) {
        if (x == 0) { write("done"); return 1; }
        else return x * fact(x - 1);
    }
}

public aspect Test {
    pointcut fcall(int i):
        call(int fact(int)) && args(i);
    before(int i): call(* write(*)) && !within(A)
        && cflow(fcall(i)) {
        write("Current arg is " + i);
    }
}
```

Current arg is 0
done
Pop Quiz

1. Write an aspect that traces the calls to `fact`, and writes the current and the original argument before each call.

   - Current arg is 7, original arg was 7
   - Current arg is 6, original arg was 7
   - Current arg is 5, original arg was 7
   - Current arg is 4, original arg was 7
   - Current arg is 3, original arg was 7
   - Current arg is 2, original arg was 7
   - Current arg is 1, original arg was 7
   - Current arg is 0, original arg was 7
   done

How do you pass the original argument along?
Counting Method Calls per Object (revisited)

Implement the following concern:
*The FigureEditor shall count the number of times its methods are called for each FigureElement-object and print this before termination.*

```java
public aspect CountCallsPerObject {
    int FigureElement.count = 0;
    before(FigureElement fe):
        call(* FigureEditor..*(..)) && target(fe)
    {
        fe.count++;
    }
    ...
}
```

Inter-type declaration adds `count` field to `FigureElement` class.
Inter-type declarations modify the static program structure.

• can add members to existing classes / interfaces
  – fields / methods / constructors
• can introduce new classes / interfaces
• can extend inheritance relation
• can declare warnings / errors
  ⇒ compile-time effects
Adding Members

• declaration similar to “normal” members
  – specify target class
    ▶ must be uniquely defined
  – introduced in target class
    ▶ called *introductions* in early AspectJ literature

• can be **private** or **public**

```java
public aspect LineLabeling {
  private String Line.label = "";
  public void Line.setLabel(String s) {
    label = s;
  }
  public String Line.getLabel() {
    return label;
  }
}
```

no qualification required – inter-type member is in scope
Visibility Rules for Inter-type Member Declarations

- **private** members visible only in declaring aspect
  - no name clash with existing members in target
    - name clashes between inter-type members introduced by different aspects resolved by aspect precedence

- **public** visibility relative to aspect’s package
  - *not* target class package
  - same for other access modifiers

- inter-type code...
  - can see aspect methods
    - target methods take precedence
  - can *not* see aspect fields
  - can *only* see public interface of target class
    - privileged aspects can see private / protected
Visibility Rules for Inter-type Member Declarations (cont’d)

• AspectJ does not distinguish between inter-type members and original members
  – pointcut descriptors catch introduced members
    * Line.get*() && !(* Line.getLabel())
Extending the Inheritance-Relation

- aspect can introduce new class
- and add new inheritance links

```java
public aspect Labeling {
    public class Labels extends FigureElement {
        private String label = "";
        public void setLabel(String s) {
            label = s;
        }
        public String getLabel() { return label; }
    }
    declare parents
        (Line || Point) extends Labels;
}
```
Extending the Inheritance-Relation (cont’d)

• implements mixins
Extending the Inheritance-Relation (cont’d)

• aspect can introduce new marker interface
• and populate it
• and add new inheritance links

```java
public aspect Labeling {
    interface Labels {} // marker interface
    private String Labels.label = "";
    public void Labels.setLabel(String s) {
        label = s;
    }
}
```

classes don’t need to implement the same interface(s)

`declare parents` (Line || Point || Compound) `extends Labels`
Error advice can enforce domain-specific language restrictions.

Ensure that all object creations use factory methods.

class FigureElementFactory {
    static Line mkLine(Point p1, Point p2) {
        return new Line(p1, p2);
    }
    ...
}

public aspect FactoryCheck {
    pointcut illegalNew():
        call(FigureElement+.new(..)) &&
        !withinCode(* FigureElementFactory.mk*(..));
    before(): illegalNew() {
        throw new Error("Use factory method");
        dynamic check (at runtime)
Error advice can enforce domain-specific language restrictions.

Ensure that all object creations use factory methods.

```java
class FigureElementFactory {
    static Line mkLine(Point p1, Point p2) {
        return new Line(p1, p2);
    }
    ...
}

global aspect FactoryCheck {
    pointcut illegalNew():
        call(FigureElement+.new(..)) &&
        !withincode(* FigureElementFactory.mk*(..));
    declare error: illegalNew() {
        "Use factory method";
    }
}
```
Static vs. Dynamic Joinpoints

Static joinpoints:
• primitive joinpoint kinds
• within()
• withincode()

Dynamic joinpoints:
• args()
• this()
• target()
• if()
• cflow()
• cflowbelow()
Things not covered...

- additional joinpoints
  - exception handling
  - object / class initialization
- aspect precedence
- abstract aspects
- aspect instantiation
- annotations

read up on these in the guide [www.eclipse.org/aspectj](http://www.eclipse.org/aspectj)