Aspect-Oriented Programming and the AspectJ language

Tamás Kozsik

Managing complexity

- Driving force
  - Methodologies
  - Programming languages
- Abstraction and modularity

Increase reusability

- Nice dream
- Only partially achieved

- Modularity and abstraction

Modularity

- Methodology
  - decomposition
  - hierarchy
  - cohesion within components
  - narrow interface between components
- Language
  - encapsulation, information hiding
  - subprograms, classes, packages
  - compilation units, libraries

Decomposition

- Separation of concerns (Dijkstra)
- Based on
  - functionality (features)
  - data structures (objects)
  - control flow (concurrency)
  - services, technologies (framework)
  - …

Example by [OT'99]

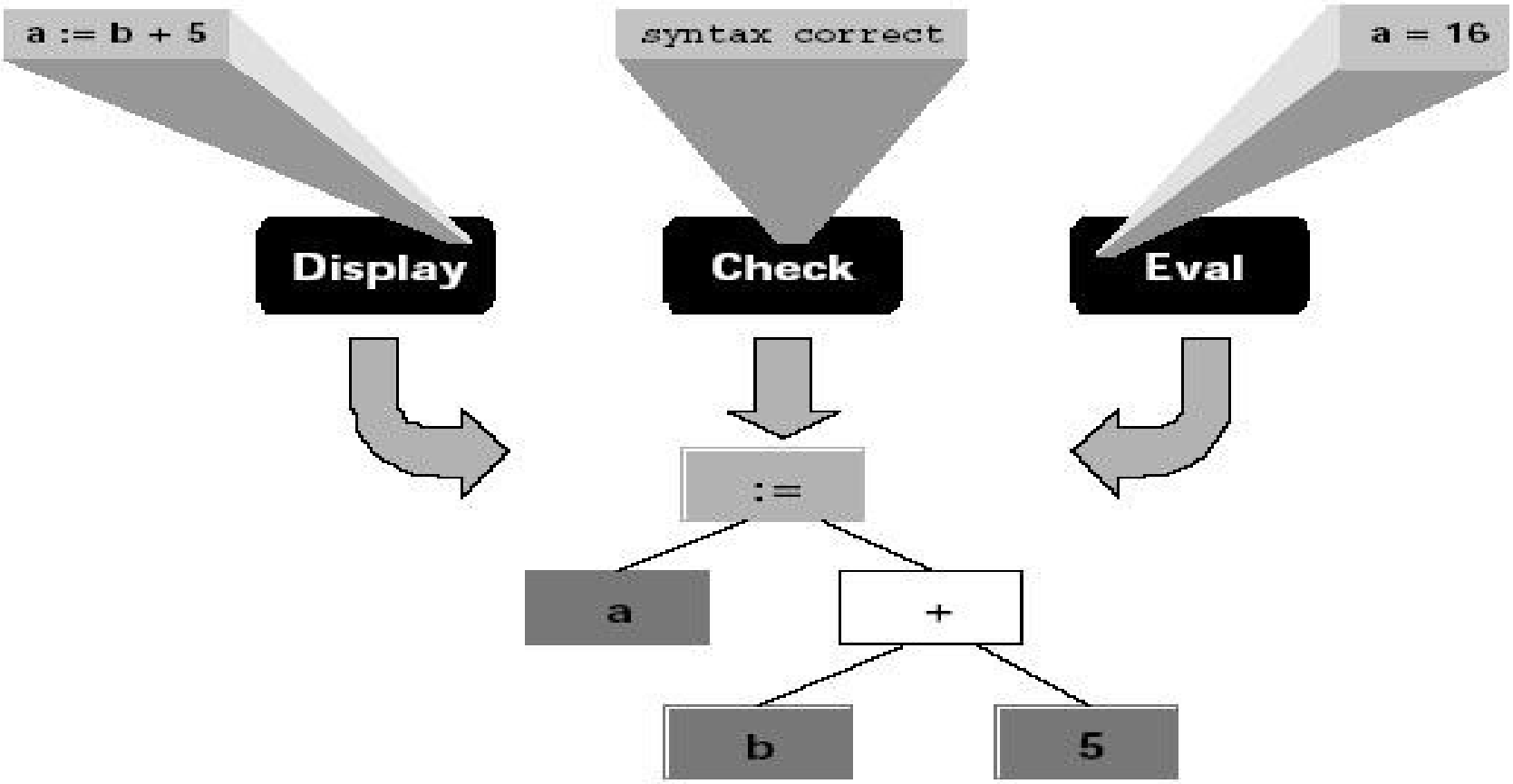


Figure 1. Tools and Shared AST in the Expression SEE.

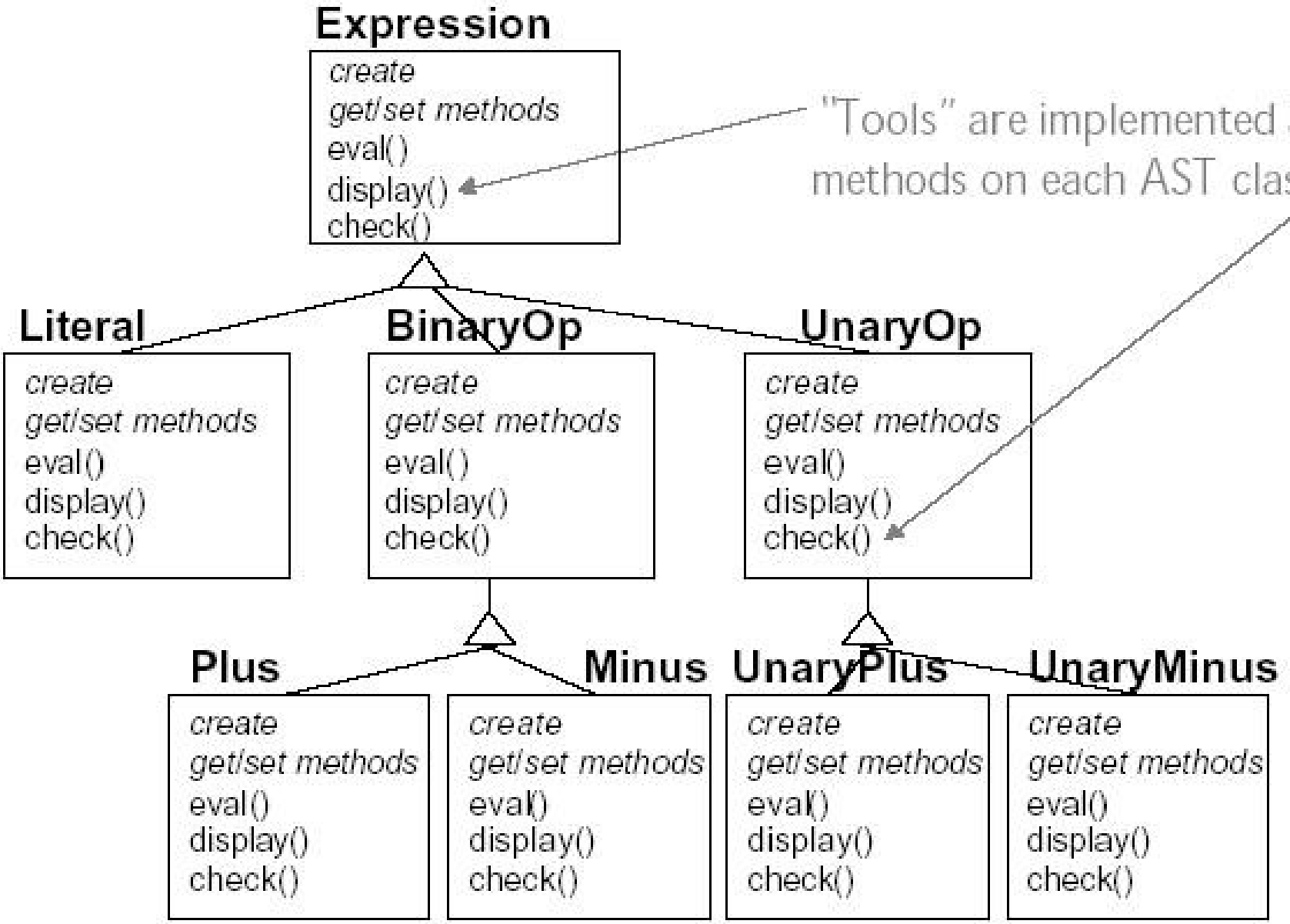


Figure 3. Partial UML Design for the Expression SEE

Problems with this design

- Consider e.g. the concern "display"
- Scattering and tangling
- Bad decomposition? Need another one?

- No!

Tyranny of the Dominant Decomposition [OT'99]

- Arbitrariness of the decomposition hierarchy [MO'05]
- Current methodologies/languages have DD
- Need to overthrow the tyranny
- Need for decomposition in multiple dimensions simultaneously
- Several approaches…

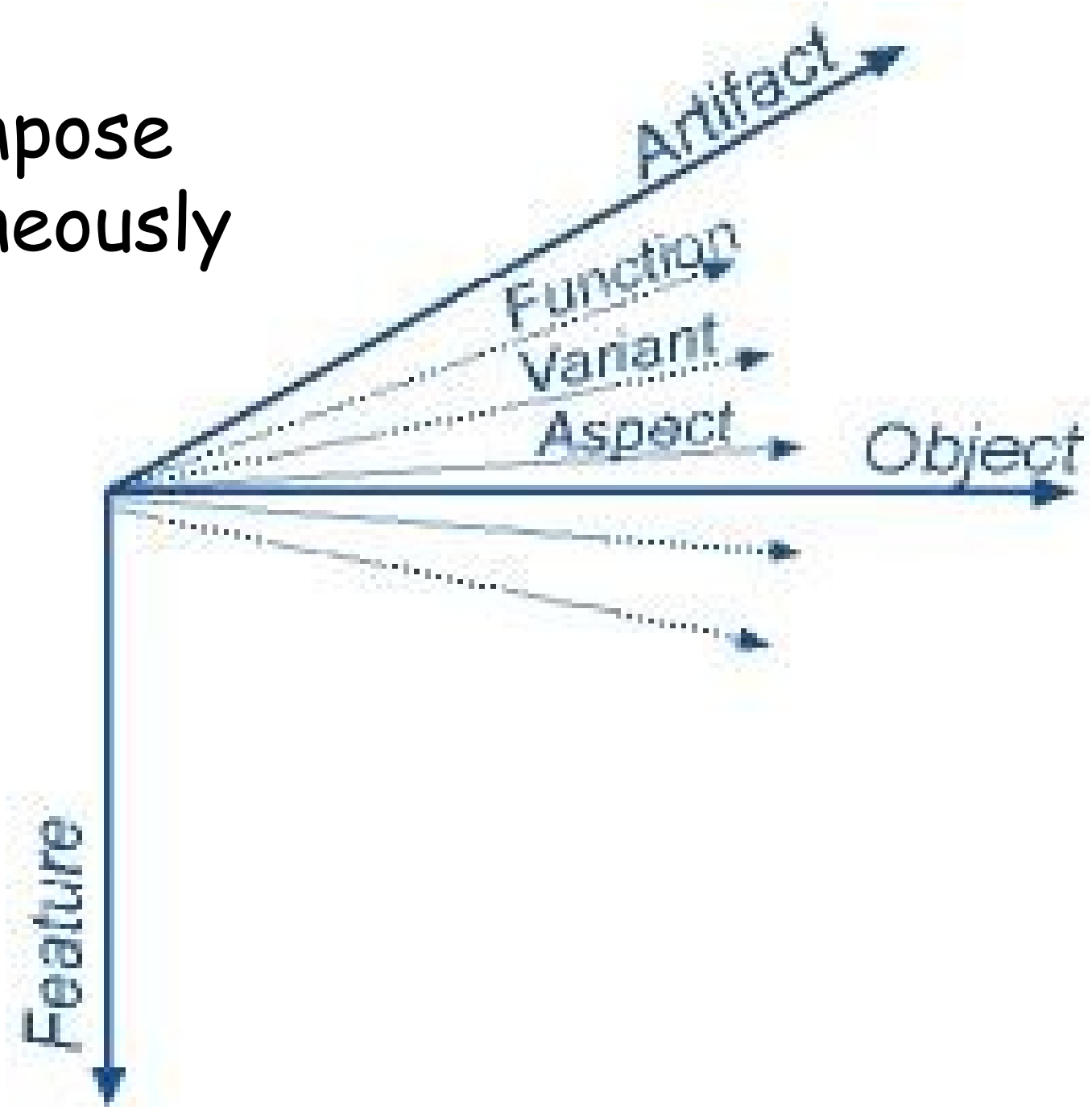
Multi-Dimensional Separation of Concerns

- IBM Research, Ossher, Tarr
- General solution
- For every artifact in software development
- A realization: Hyperslices
- An implementation: Hyper/J

Concerns

- A dimension of concerns
  - Decomposition is based on ~
  - Functionality, data structures, control flow, etc.
- Concern
  - Obtained by decomposition
  - An element of a dimension of concerns
  - A program feature, a class, a process
  - BinaryOp, Plus, Display, Evaluation

Decompose simultaneously



Dimensions

- Artifacts
  - Specification, Design docs, Code, Test suite
- Functionality (features)
  - Evaluation, Display, Persistence
- Data structures (classes)
  - Expression, UnaryOp, Plus
- Variants
  - For different configurations (e.g. style checks)
- Units of change

The overall system

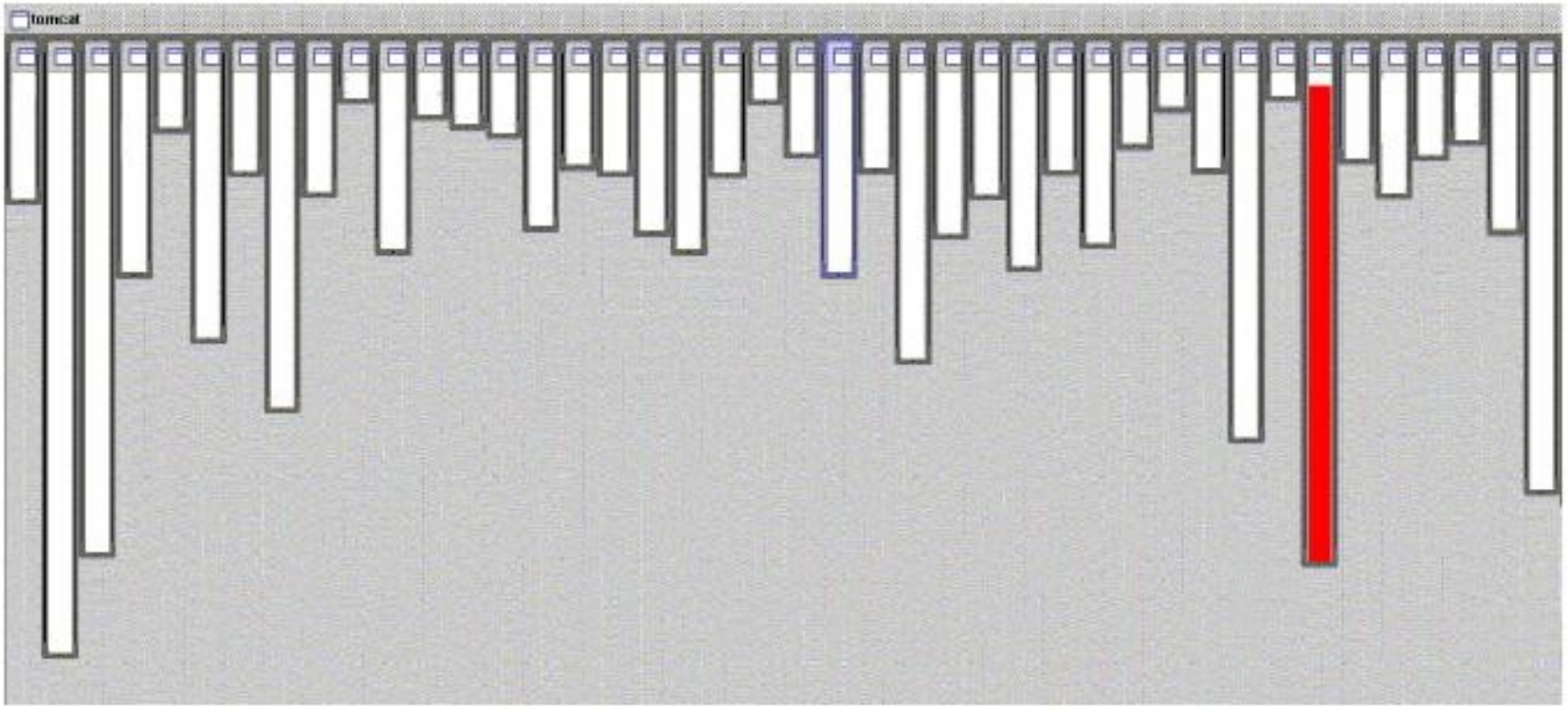
- Can be viewed and modified in different dimensions of concerns
  - Different developers
  - Same developer at different times
- The dimensions and the decompositions
  - are coequal
  - can evolve

Aspect-Oriented Software Development

- A less general solution
- Base functionality + crosscutting concerns
- Simple and powerful
- Became popular and wide-spread
- Many approaches, many implementations
- Aspect-Oriented Programming
- Most famous: AspectJ

good modularity

XML parsing

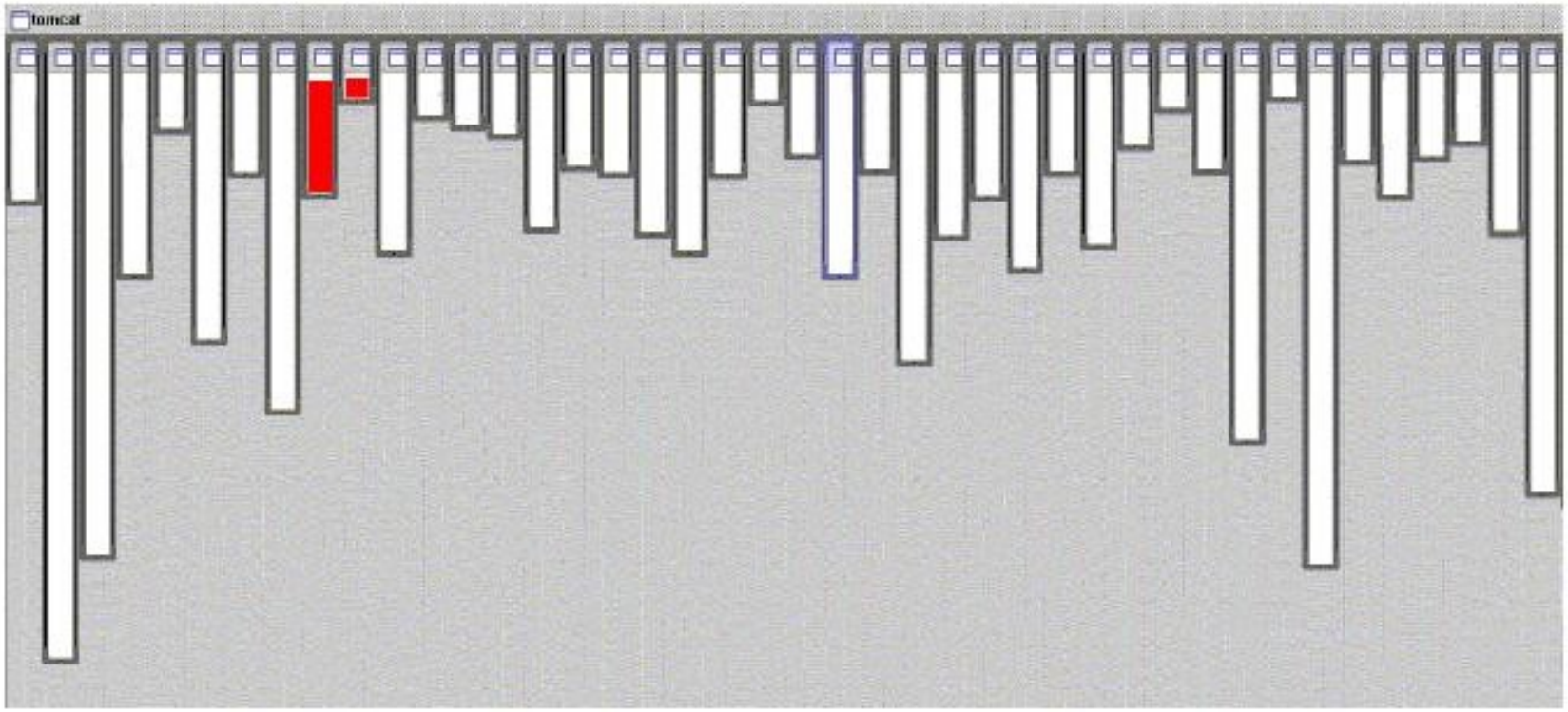


- **XML parsing in org.apache.tomcat**
  - red shows relevant lines of code
  - nicely fits in one box

aspectj.org

good modularity

URL pattern matching

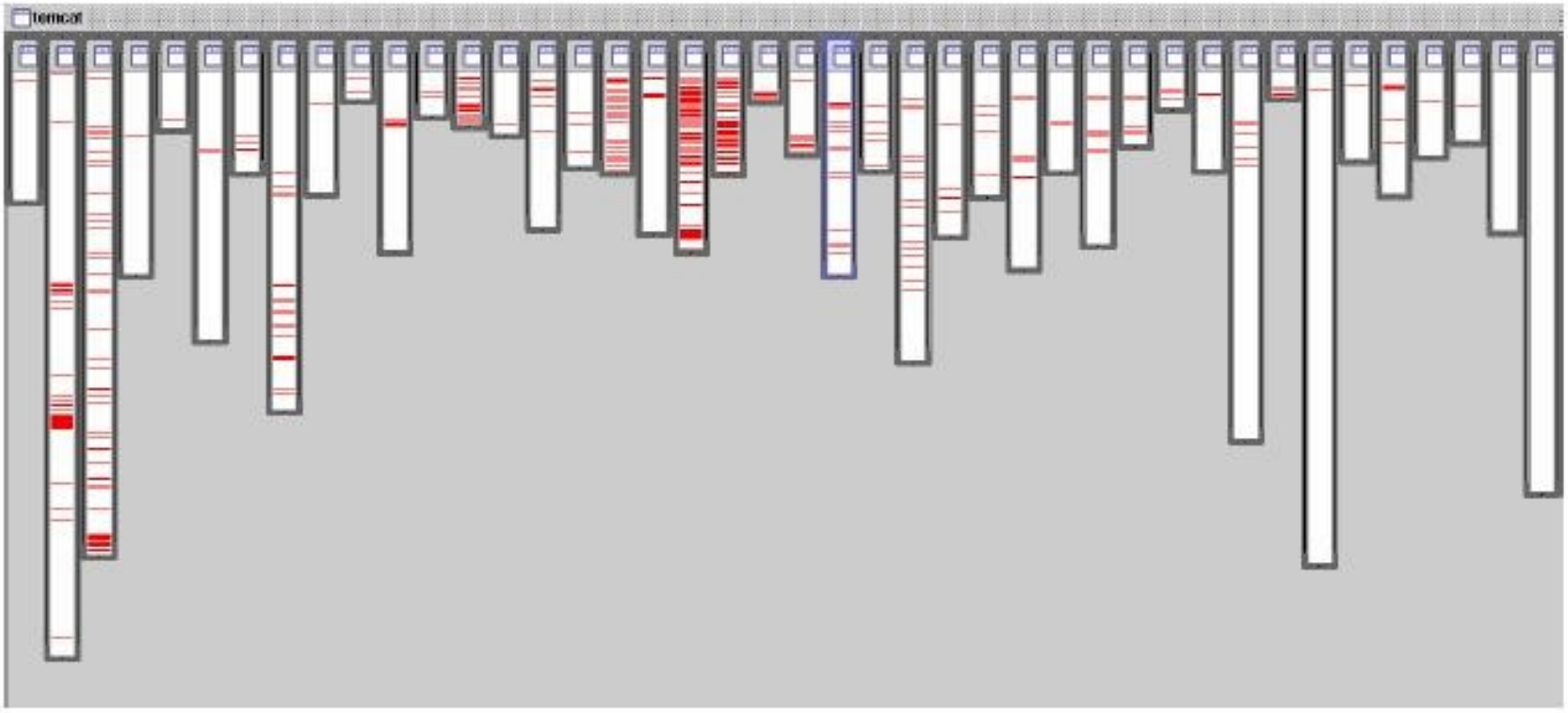


- **URL pattern matching in org.apache.tomcat**
  - red shows relevant lines of code
  - nicely fits in two boxes (using inheritance)

aspectj.org

problems like…

logging is not modularized

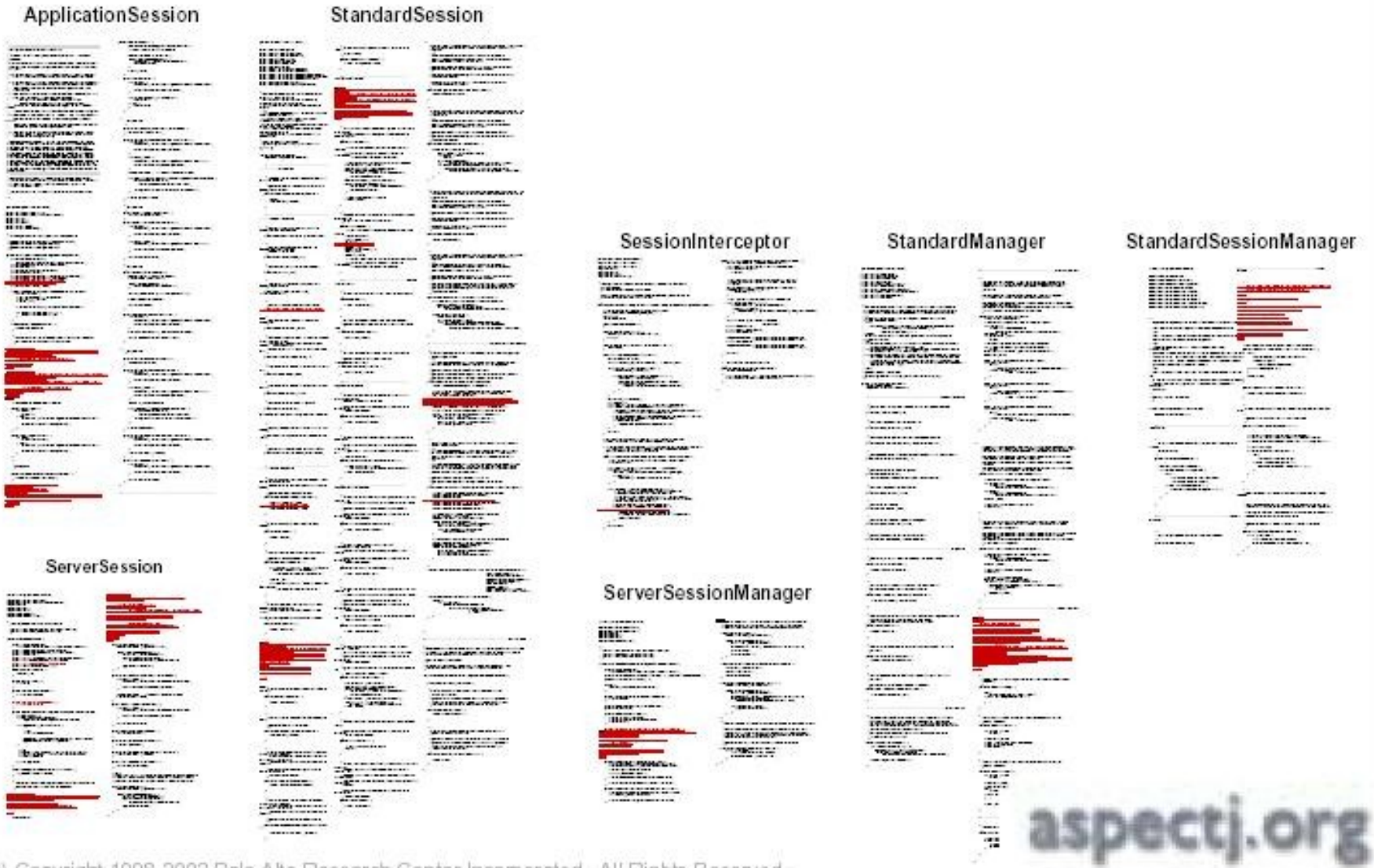


- **where is logging in org.apache.tomcat**
  - red shows lines of code that handle logging
  - not in just one place
  - not even in a small number of places

aspectj.org

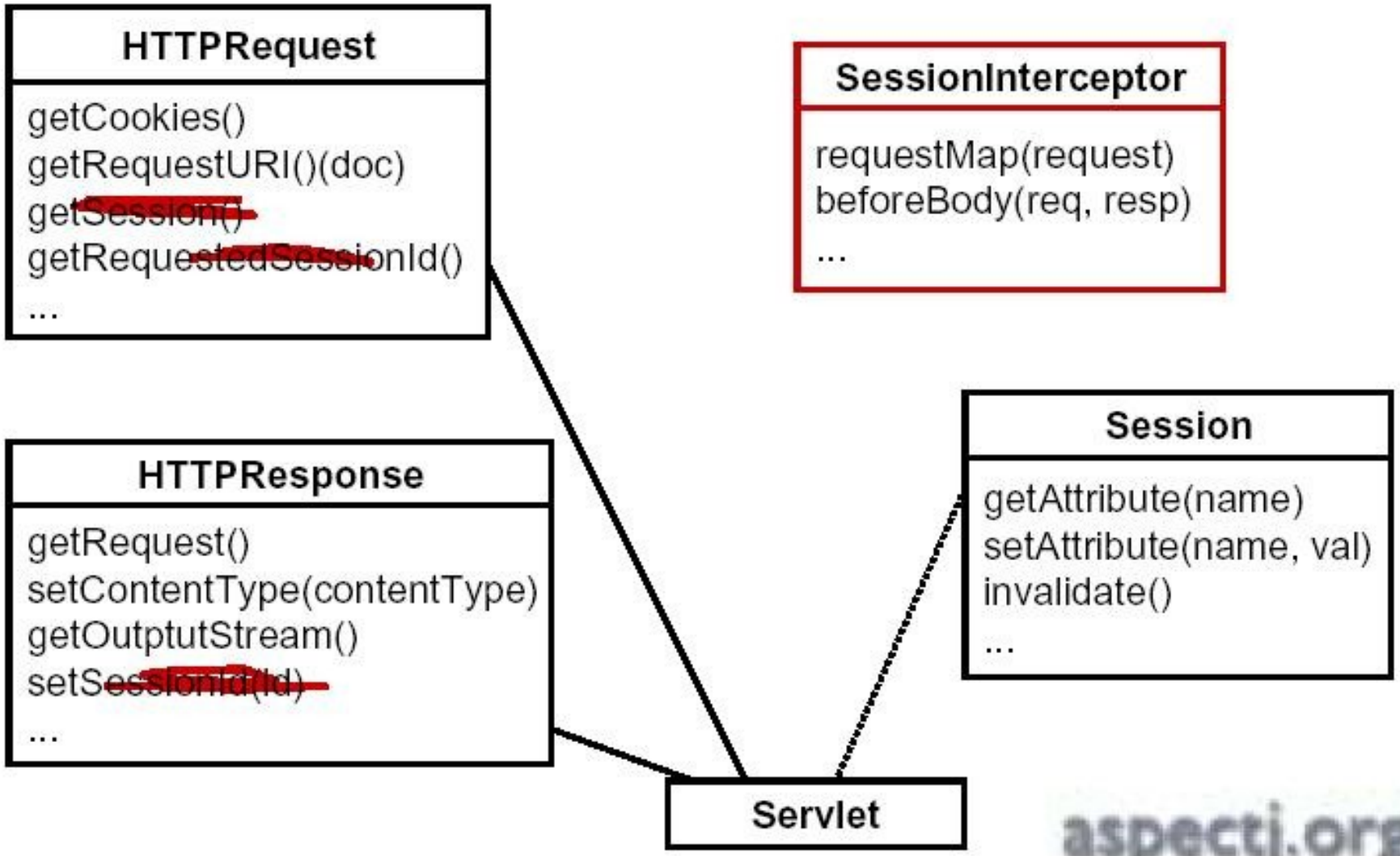
problems like…

session expiration is not modularized



problems like…

session tracking is not modularized



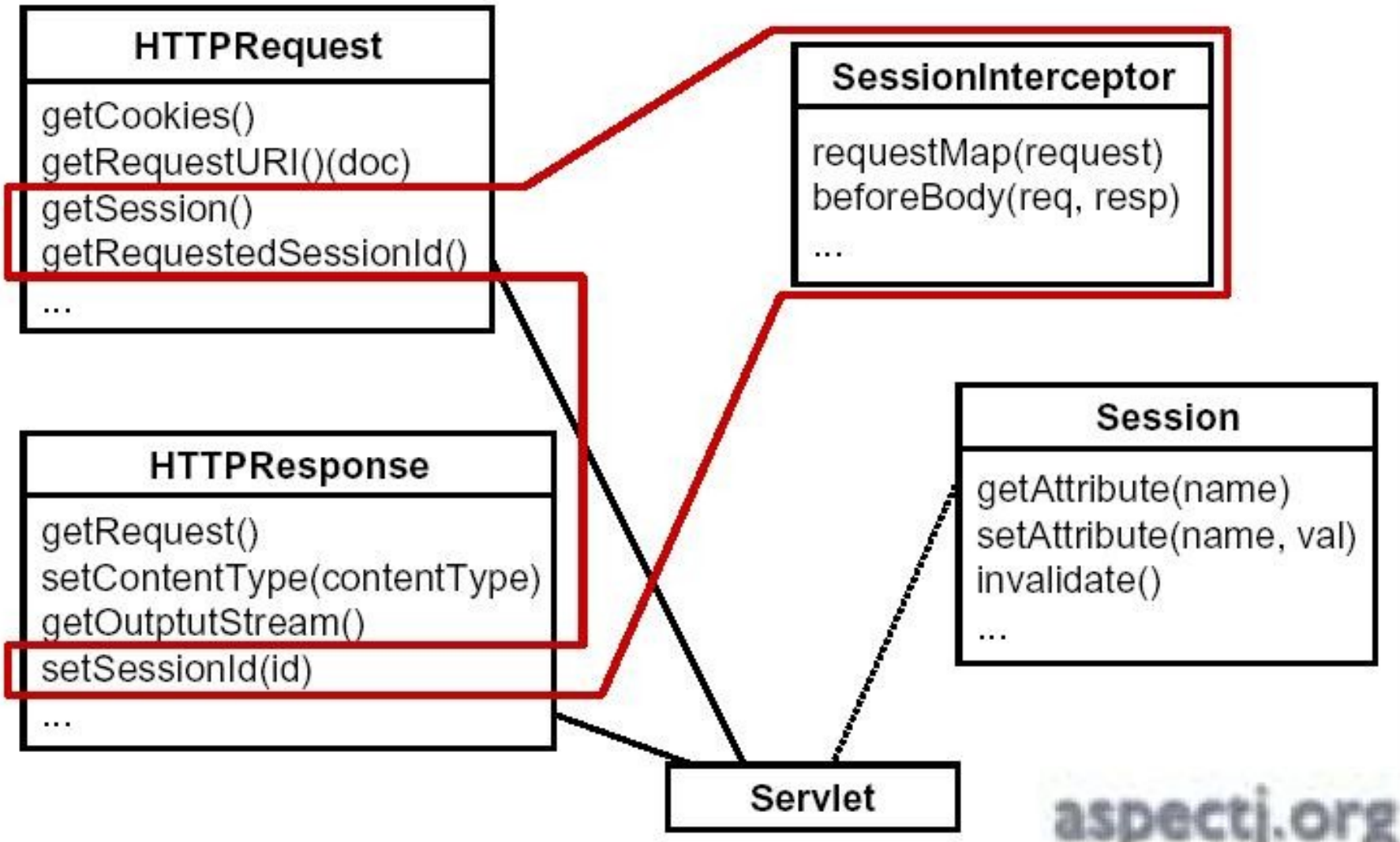
aspectj.org

Crosscutting concern

- A concern that appears at many different places in the program
- Scattering
- Tangling

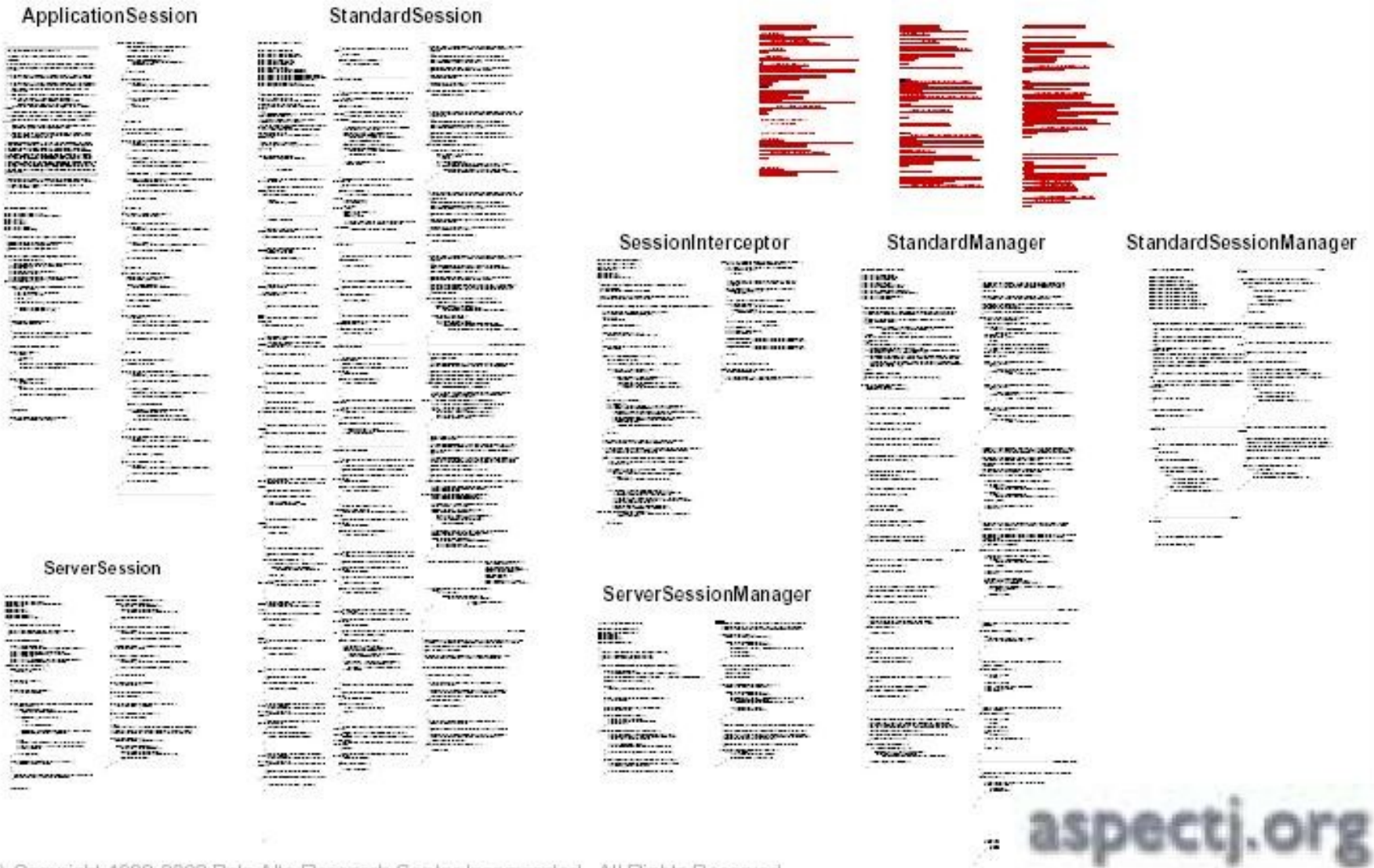
- Physical separation: in an aspect
- Pluggable

crosscutting concerns



aspectj.org

language support to…



Examples of crosscutting

- Tracing and profiling
- Logging
- Configuration management
- Exception handling
- Security
- Synchronization
- Verifying correctness

Example of aspect

```
import java.io.*;
aspect SimpleLogging {
   pointcut loggedCalls():
          call(public * *.*(..));

   after() throwing (IOException e): loggedCalls() {
          System.err.println(e);
   }
}
```

AOP terminology

- Crosscut: affect more modules
- Aspects: modules implementing crosscutting behaviour
- Obliviousness: base code not referring to aspects
- Join points: where composition happens
- Quantification: reference to more join points
- Aspect-weaver: composes aspects with base code

The beginning of AOP [Lopes'05]

- Much related research from early 90'
- Special purpose AOP languages [K…'97]
  - Concurrency: D framework [Lopes'97]
    - Cool: coordination
    - Ridl: remote access
  - Performance
    - RG: image processing [MKL'97]
    - AML: sparse matrix manipulation [I…'97]
- DJ, DJava, AspectJ
- AspectJ (general purpose) [LK'98]
- Further general purpose AOP languages & technologies

My story of AOP

- (Multi-agent) simulations
- Concerns
  - Modelling
  - Observation
- OOP: tangling and scattering
  - Conceptual separation
  - Publishing models
  - Multiple observers
- Multi-Agent Modelling Language (MAML), 1999. [GK'99]

Some examples [AOSD]

- AspectC++
- AspectC
- Aspect#, AspectDNG, LOOM.NET, AspectC#, EOS
- Aspect-Oriented Perl, Aspect.pm
- Aspects, Pythius (Python)
- AspectR (Ruby)
- AspectS, Apostle, MetaclassTalk (Squeak/Smalltalk)
- AspectXML
- AOPHP, AspectPHP
- UMLAUT

Some examples for Java

- Spring AOP
- JBoss-AOP
- AspectWerkz
- Object Teams
- Caesar
- Java Aspect Components
- JMangler, JOIE, JMunger
- DJ
- ComposeJ, ConcernJ, JCFF
- Java Layers
- JPiccola
- Pragma
- Lasagne/J

Reminder

- Managing complexity: abstraction and modularization
- Better separation of concerns is needed
- Problems with DD: scattering and tangling
- Popular solution: AOSD, AOP
- Crosscutting concerns turned into aspects
- Obliviousness, join points, aspect-weaver
- AspectJ, among others…

The AspectJ language

- Join points (concept)

- Pointcuts
- Advice (constructs)
- Inter-type declarations
- Aspects

The first program: a class

```
public class Hello {

     void greeting(){
      System.out.println("Hello!");
     }

     public static void main( String[] args ){
      new Hello().greeting();
     }

}
```

The first program: an aspect

```
public aspect With {
     before() : call( void Hello.greeting() ) {
      System.out.print("> ");
     }
}
```

The first program: compile & run

- Source file for aspects: `.java` or `.aj`
- **`PATH`** includes <aspectj>`/bin`
- **`CLASSPATH`** includes <aspectj>`/lib/aspectjrt.jar`

```
ajc Hello.java With.aj
java Hello
```

ajc

- Aspect weaver
- Compiles Java and AspectJ
- Produces efficient code
- Incremental compilation
- Accepts bytecode

The first program: after weaving (Simplified view!!!)

```
public class Hello {
  void greeting(){ System.out.println("Hello!"); }
  public static void main( String[] args ){
     Hello dummy = new Hello();
     System.out.print("> ");
     dummy.greeting();
  }
}
```

Join points

- New concept
- Well-defined points in the program flow
  - call of a method or constructor
  - execution of a method or constructor
  - execution of a **`catch`**
  - getting/setting a field
  - initialization of a class, object or aspect
  - execution of advice

Pointcut

- A language construct
- Picks out certain join points (and values): quantification
- Composition

```
call( void Hello.greeting() )
call( * Hello.*(..) )
call( void Hello.greeting() )  &&  target(f)
call(void Point.setX(int)) ||
                call (void Point.setXY(int,int))
```

Advice

- A language construct
- Code to be executed at certain join points
  - **before**, **after** or **around**

```
before() : call( void Hello.greeting() ) {
                System.out.print("> ");
}
```

Inter-type declaration

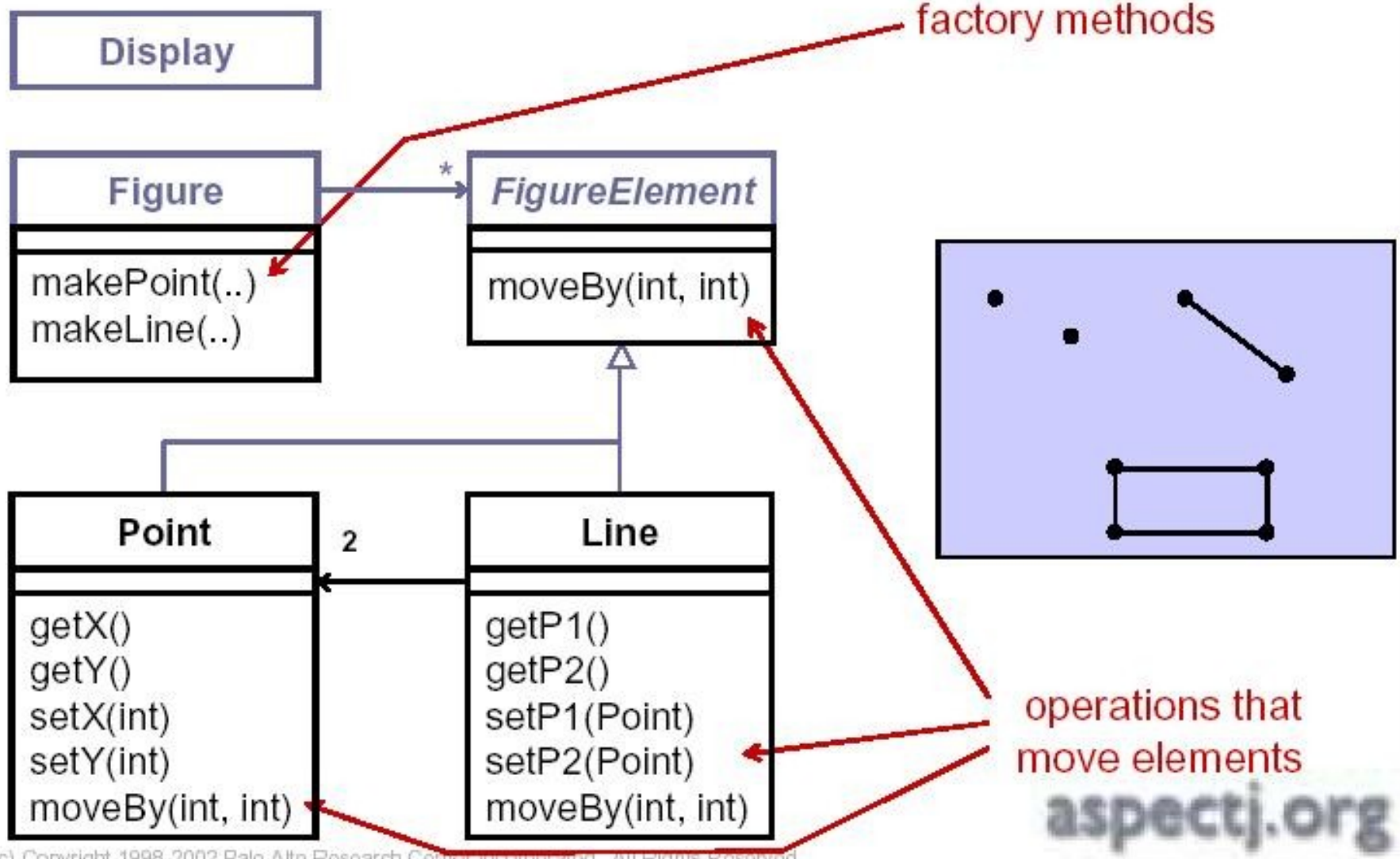
- A language construct
- Modify the static structure of a program
  - introduce new members
  - change relationship between classes

Aspect

- A language construct
- The unit of modularity for crosscutting concerns
- May contain pointcuts, advice and inter-type declarations

```
public aspect With {
        before() : call( void Hello.greeting() ) {
                System.out.print("> ");
        }
}
```

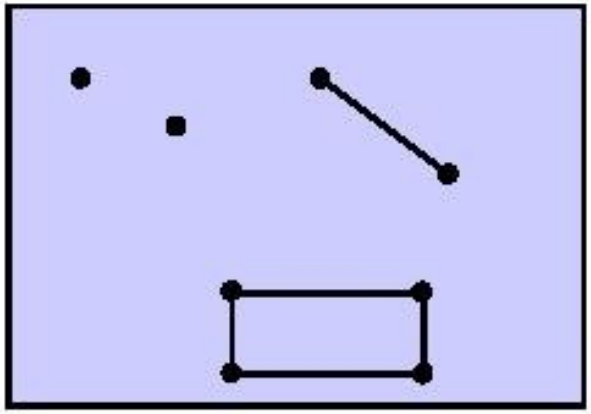
a simple figure editor



a simple figure editor

```
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) { ... }
}
```



aspectj.org

Composing pointcuts

- Pointcut: pick up joint points
- Composition operators: **&&** **||** **!**
- The result is a pointcut

```
call(void Point.setX(int)) ||
call(void Point.setY(int))
```

(Name-based) crosscutting

```
   call(void FigureElement.moveBy(int,int))
|| call(void Point.setX(int))
|| call(void Point.setY(int))
|| call(void Line.setP1(Point))
|| call(void Line.setP2(Point))
```

- Affects multiple types

Named pointcuts

```
pointcut move():
        call(void FigureElement.moveBy(int,int))
    || call(void Point.setX(int))
    || call(void Point.setY(int))
    || call(void Line.setP1(Point))
    || call(void Line.setP2(Point));
```

- Declares a pointcut named **move**
- May be used many times

Using named pointcuts

```
pointcut move():
        call(void FigureElement.moveBy(int,int))
    || call(void Point.setX(int))
    || call(void Point.setY(int))
    || call(void Line.setP1(Point))
    || call(void Line.setP2(Point));

before() : move(){
    System.out.println("moving something");
}
```

Property-based crosscutting

```
call( public * Figure.* (..) )
call( * Figure.make*(..) )
```

- Wildcards in the signature
- Not only syntactical, but also lexical match

Dynamic joint point model

- Joint point model – classification of AOP languages
- AspectJ: dynamic j.p.m.
- Run-time events
- Containment
  - Dynamic context of a joint point

Dynamic context of a joint point

```
cflow( move() )
cflowbelow( move() )
```

- Join point selection based on dynamic semantics

```
before() : move() && (! cflowbelow(move())) {
        System.out.println("moving something");
        ++counter;
}
```

Exercise

Give a pointcut expression

- for calling a public method of any class returning an int
- for calling a setter method in the control flow of a make* in Figure

Solution of exercise

```
call( public int *.*(..) )

cflow( call(* Figure.make*(..)) )
&& call( void *.set*(..) )
```

Advice

- Provide code to execute at a join point

- before
- after
  - if succeeds
  - if fails
- around

before advice

- Right before the join point

```
before() : call( * *.set*(..) ) {
      System.out.println("about to set");
}
```

- Before method call
- After the arguments are evaluated

after advice

- Right after the join point

```
after() : call( * *.set*(..) ) {
    System.out.println("after setting");
}
```

- Variants for testing success

After success

```
after() returning : call(* *.set*(..)) {
    System.out.println("setting OK");
}
```

- When method exited normally

After failure

```
after() throwing : call( * *.set*(..) ) {
    System.out.println("setting failed");
}
```

- When method exited with an exception

around advice

- Instead of, or around, join points

```
void around() : call(void Figure.moveBy(..)) {
    System.out.print("Press Y to really move figure:");
    try {
          if (System.in.read() == 'Y')    proceed();
    } catch (java.io.IOException e) {}
}
```

Parametrized advice

- Formal parameter list in advice
- Bound by the pointcut

```
before( Figure f ) :
call(* Figure.moveBy(..)) && target(f) {
    System.out.println("before move: " + f);
}
```

Exposing context in pointcuts

- With three primitive pointcuts:

  this **target** **args**

```
before( Figure f, FigureElement fe, int x, int y ):
call( void FigureElement.moveBy(int,int) )
&& this(f) && target(fe) && args(x,y)
{
    …
}
```

Exercise

- Print which FigureElement and with which vector

```
void around() : call(void FigureElement.moveBy(..)) {
     System.out.print("Press Y to really move FE: ");
     try {
          if (System.in.read() == 'Y')   proceed();
     } catch (java.io.IOException e) {}
}
```

Solution of exercise

```
void around( FigureElement fe, int dx, int dy) :
target(fe)  &&  args(dx,dy)  &&  call(void FigureElement.moveBy(..))
{
   System.out.print("About to move "+fe+" with "+dx+", "+dy);
   System.out.print(". Press Y to really move figureElement: ");
   try {
       if (System.in.read() == 'Y')   proceed();
   } catch (java.io.IOException e) {}
}
```

Inter-type declarations

- Modify the static structure of the program
- Compile-time effect
- Addition of fields, methods or constructors to a class
- … or to multiple classes (crosscutting)
- Change the inheritance hierarchy

Introducing line labels

- Point, Line: geometrical properties of FigureElements
  - translate, rotate, reflect, etc.
- Labels for lines
  - relevant for displaying lines
  - another aspect (part of displaying aspect)

Labeling aspect

```
public aspect Labeling {
      private String Line.label;
      public void Line.setLabel(String s){
        label = s;
      }
      public String Line.getLabel(){
           return label;
      }
      …
}
```

Mixin

```
public aspect Labeling {
    public static class Labelled
    extends FigureElement {
        private String label;
        public void setLabel(String s){…}
        public String getLabel(){…}
    }

    declare parents:
        (Point || Line) extends Labelled;
}
```

Interface-based extension

```
public aspect Labeling {
    interface Labelled {}

    private String Labelled.label;
    public void Labelled.setLabel(String s){
   label = s;
   }
    public String Labelled.getLabel(){
   return label;
   }

    declare parents: (Point||Line||Figure)
    implements Labelled;
}
```

Concerns to implement as aspects

- Development aspects
  - Tracing, Profiling, Logging
  - Pre- and postconditions, Contract enforcement
  - Configuration management
- Production aspects
  - Change monitoring
  - Context passing
  - Providing consistent behavior
  - Collaboration-based design
- Reusable aspects

Profiling

- Flexible: programmatically

```
aspect SetsInRotateCounting {
   int rotateCount = 0;
   int setCount = 0;
   before(): call(void Line.rotate(double)) {
      rotateCount++;
   }
   before(): call(void Point.set*(int)) &&
                  cflow(call(void Line.rotate(double))){
      setCount++;
   }
}
```

Contract enforcement

`withincode` **pointcut**

```
aspect RegistrationProtection {
   pointcut register():
      call(void Registry.register(FigureElement));
   pointcut canRegister():
      withincode(* Figure.make*(..));
   before(): register() && !canRegister() {
      throw new IllegalStateException(…);
   }
}
```

Static contract enforcement

`declare error`, based on static information

```
aspect RegistrationProtection {
   pointcut register():
     call(void Registry.register(FigureElement));
   pointcut canRegister():
     withincode(* Figure.make*(..));
   declare error:
     register() && !canRegister() :
                              "Illegal call";
}
```

Concept checking

- **`declare error`**
- **`declare warning`**

- Have the compiler issue (programmed) compilation errors/warnings
- Extend the compiler with additional grammatical and static semantical rules

Production aspects

- Used in production builds
- Add real functionality to applications
- E.g. the Labeling aspect
- Further examples
  - Change monitoring
  - Context passing
  - Providing consistent behaviour
  - Security
  - Resource pooling, caching
  - Synchronization of threads

Change monitoring

- Indicate whether any of the FigureElement has moved since last display
- Dirty flag introduced
- Setting the dirty flag at each method that moves a figure element

Implementation in an aspect

```
aspect MoveTracking {
    private static boolean dirty = false;
    public static boolean testAndClear(){
        boolean result = dirty;
        dirty = false;
        return result;
    }
    …
    after() returning: move() {
        dirty = true;
    }
}
```

Advantages

- The structure of the concern is made explicit
- Evolution is easier [ECOOP 01]
- Plug in or out
- More stable implementation

Context passing

- Set the color of FigureElements upon creation
- Pass a color (or a color factory) to **`make*`**
- This may influence many methods:
  on the control flow from client to **`make*`**
- Non-AOP solution: additional arg to those methods
- AOP solution: pass information between far-away code fragments

Passing context with aspect

```
aspect ColorControl {

   after (ColorControllingClient c)
   returning (FigureElement fe):
   call(* Figure.make*(..)) &&
   cflow( call(* * (..)) && this(c) )
   {
      fe.setColor(c.colorFor(fe));
   }

}
```

A fragment of ajc

```
aspect ContextFilling {

   pointcut parse( JavaParser jp ):
   call(* JavaParser.parse*(..)) && target(jp)
   && !call(Stmt parseVarDec(boolean));

   around(JavaParser jp) returning ASTObject: parse(jp) {
   Token beginToken = jp.peekToken();
   ASTObject ret = proceed(jp);
   if (ret != null) jp.addContext(ret, beginToken);
   return ret;
   }

}
```

Consistent behaviour

- Advising 35 methods

  `parseMethodDec`, **`parseThrows`**, **`parseExpr`** etc.
- Explicit exclusion of **`parseVarDec`**
- Clear expression of a clean crosscutting modularity
- Java → AspectJ refactoring revealed two bugs

Subject/Observer design pattern

```
public abstract aspect ObserverProtocol {
   protected interface Observer {}
   protected interface Subject {}
   private List<Observer> Subject.observers
                              = new LinkedList<Observer>();
   public void Subject.addObserver(Observer o){
      observers.add(o);
   }
   public void Subject.removeObserver(Observer o){
      observers.remove(o);
   }
   protected abstract void notifyObserver
                                    (Observer o, Subject s);
   protected abstract pointcut observedEvent(Subject s);
   after(Subject s) returning: observedEvent(s){
      for(Observer o: s.observers) notifyObserver(o,s);
   }
}
```

Binding to Point

```
aspect Binding extends ObserverProtocol {

  declare parents: Point implements Subject;
  declare parents: Point implements Observer;

  protected pointcut observedEvent(Subject s):
      set(int Point.x) && target(s);

  protected void notifyObserver( Observer o,
                                 Subject s ){
      ((Point)o).x = ((Point)s).x;
  }

}
```

Problems to solve

- With ObserverProtocol
  - If a class is a Subject in two independent bindings...
- With Binding
  - If two different bindings interfere (notification in one of them triggers notification in the other one and vice versa)

Parent/child relationship (1)

```
public abstract aspect
ParentChildRelationship <Parent,Child> {

    public interface
    ParentHasChildren <C extends ChildHasParent> {
        Set<C> getChildren();
        void addChild(C child);
        void removeChild(C child);
     }

    public interface
    ChildHasParent <P extends ParentHasChildren> {
        P getParent();
        void setParent(P parent);
    }

    ...
}
```

Parent/child relationship (2)

```
public abstract aspect
ParentChildRelationship <Parent,Child> {
    public interface
    ParentHasChildren <C extends ChildHasParent>
    { ... }

    public interface
    ChildHasParent <P extends ParentHasChildren>
    { ... }

    declare parents:
      Parent implements ParentHasChildren<Child>;

    declare parents:
      Child implements ChildHasParent<Parent>;

    ...
}
```

Parent/child relationship (3)

```
private Set<C> ParentHasChildren<C>.children
                                    = new HashSet<C>();

private P ChildHasParent<P>.parent;

public Set<C> ParentHasChildren<C>.getChildren() {
   return Collections.unmodifiableSet(children);
}

public P ChildHasParent<P>.getParent() {
   return parent;
}
```

Parent/child relationship (4)

```
public void ParentHasChildren<C>.addChild(C child) {
   if (child.parent != null)
      child.parent.removeChild(child);
   children.add(child);
   child.parent = this;
}

public void ParentHasChildren<C>.removeChild(C child) {
   if (children.remove(child))  child.parent = null;
}

public void ChildHasParent<P>.setParent(P parent) {
   parent.addChild(this);
}
```

Picking out further join points: *kinded pointcuts* describe events

`call(`*MethodPattern*`)`
`call(`*ConstructorPattern***`)`**
`execution(`*MethodPattern***`)`**
`execution(`*ConstructorPattern***`)`**
`set(`*FieldPattern*`)`
`get(`*FieldPattern***`)`**
`initialization(`*ConstructorPattern***`)`**
`preinitialization(`*ConstructorPattern***`)`**
`staticinitialization(`*TypePattern***`)`**
`handler(`*TypePattern***`)`**
`adviceexecution()`

References (Aspect-Oriented Software Development)

[FECA'05] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, Mehmet Akşit: "***Aspect-Oriented Software Development***". Addison-Wesley, 2005.

[AOSD] ***Aspect-Oriented Software Development***. http://www.aosd.net/

[Lopes'97] C. V. Lopes.: ***"D: A Language Framework for Distributed Programming"***. Ph.D. Thesis, Graduate School of College of Computer Science, Northeastern University, Boston, MA, 1997.

[K…'97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: "Aspect-oriented programming". In ***ECOOP'97 Object-Oriented Programming, 11th European Conference***, M. Akşit and S. Matsuoka, Eds. LNCS 1241. Springer-Verlag, Berlin, 1997. pp. 220-242.

References (Aspect-Oriented Software Development)

[MKL'97] Mendhekar A., Kiczales G. and Lamping J.: "***RG: A Case-Study for Aspect-Oriented Programming***". Xerox PARC, Palo Alto, CA. Technical report SPL97-009 P9710044, February 1997.

[I…'97] Irwin J., Loingtier J-M., Gilbert J. R., Kiczales G., Lamping J., Mendhekar A., Shpeisman T.: ***"Aspect-Oriented Programming of Sparse Matrix Code"***. Proc. Int'l Scientific Computing in Object-Oriented Parallel Environments. LNCS 1343. Springer-Verlag, 1997.

References (AspectJ)

[AspectJ] ***aspectj project***. http://www.eclipse.org/aspectj/

[AJDT] AspectJ Development Tools. http://www.eclipse.org/ajdt/

[Col'05] Adrian Colyer: ***"AspectJ"***. In [FECA'05]

[LK'98] Lopes C.V., Kiczales G.: Recent Developments in AspectJ. Workshop on Aspect-Oriented Programming (ECOOP), http://trese.cs.utwente.nl/aop-ecoop98/papers/Lopes.pdf 1998.

[Lopes'05] Cristina Videira Lopes: "AOP: A Historical Perspective (What's in a Name?)". In [FECA'05]

[Kis'02] Ivan Kiselev: "***Aspect-Oriented Programming with AspectJ***". Sams, 2002. ISBN 978-0672324109

References
(Related technologies - 1)

[OT'99] Harold Ossher, Peri Tarr: ***Multi-Dimensional Separation of Concerns and The Hyperspace Approach***. IBM Research Report 21452, April, 1999. IBM T.J. Watson Research Center. http://www.research.ibm.com/hyperspace/Papers/tr21452.ps

[Hyper] ***Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces***. http://www.research.ibm.com/hyperspace/

[Hyper/J] ***Hyper/J***. http://www.alphaworks.ibm.com/tech/hyperj/

[SOP] ***Subject-oriented programming***. http://www.research.ibm.com/sop/

[CF] ***Composition Filters***. http://trese.cs.utwente.nl/composition_filters/

[AP] ***Demeter: Adaptive Object-Oriented Software***. http://www.ccs.neu.edu/research/demeter/

References
(Related technologies - 2)

[MO'05] Mira Mezini, Klaus Ostermann: ***Untangling Crosscutting Models with Caesar***. In [FECA'05].

[IP] ***Intentional Programming Frequently Asked Questions***. 2003. http://www.omniscium.com/?page=IntentionalFaq

[CE'00] K. Czarnecki, U. Eisenecker: ***Generative Programming: Methods, Tools and Applications***. Addison-Wesley, Reading, Mass., 2000.

[Alex'01] Andrei Alexandrescu: ***Modern C++ Design (Generic Programming and Design Patterns Applied)***. Addison-Wesley, 2001.

References
(Some own stuff)

[GK'99] Gulyás L., Kozsik T.: "The Use of Aspect-Oriented Programming in Scientific Simulations". ***Software Technology, Fenno-Ugric Symposium Proceedings***, Technical Report CS 104/99, 1999. pp. 17-28.

[FKV'02] Frohner Á., Kozsik T., Varga L.: "Design and implementation of synchronisation patterns using an extension to UML: A case study". ***Pure Mathematics and Applications*** (PU.M.A.) Volume 13 (2002), No. 1-2, pp. 133-158, SAAS Ltd.-SAAS Publishing, Budapest, Hungary

[ZPK'03] Zólyomi I., Porkoláb Z., Kozsik T.: "An Extension to the Subtype Relationship in C++ Implemented with Template Metaprogramming". ***LNCS 2830***, pp. 209-227. Springer-Verlag Berlin Heidelberg 2003.

[Alt'04] Altrichter F.: "Extending PL/SQL with AOP features". Thesis for National Student Competition (OTDK). 2004. (in HU)