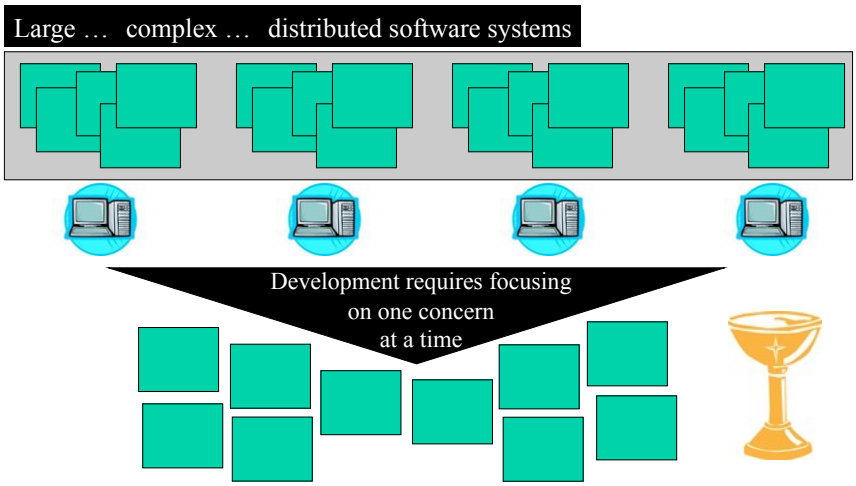
What is AOSD
(Aspect-Oriented Software Development)?

Ana Moreira
amm@di.fct.unl.pt

Need for Separation of Concerns



Separation of Concerns

This is what I mean by focusing one's attention upon a certain aspect; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extend that they are irrelevant for the current topic.

Such a separation, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of. I usually refer to it as "**a separation of concerns**" [...]."

E. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976, pp. 210

Edsger Dijkstra 1930-2002

The Problem of Crosscutting Concerns

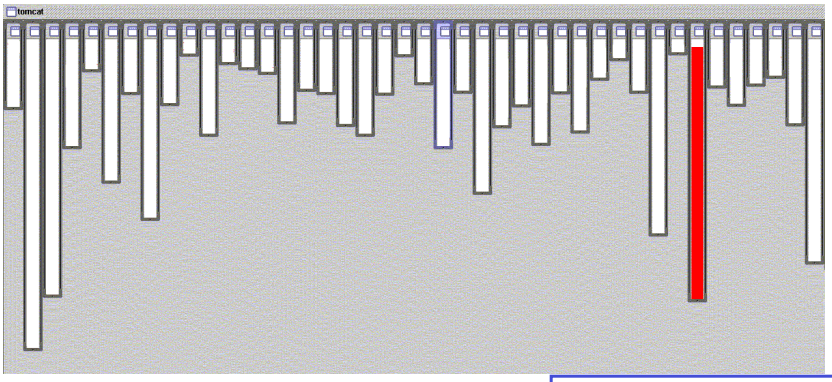
- Broadly-scoped concerns
  - Distribution, security, real-time constraints, etc.
  - Crosscutting in nature
  - Severely constrain quality attributes and separation of concerns

What are Crosscutting Concerns?

- A concern whose specification (and implementation) is scattered among several other concerns
- A concern that crosscuts various requirements sets or units in the specification
- A broadly scoped property that has an effect on multiple requirements with potential consequences to later development stages

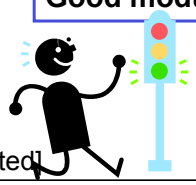
Good modularity

XML parsing



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

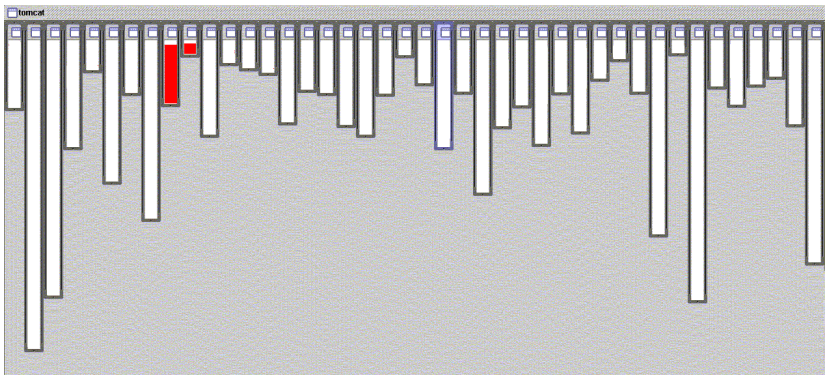
Good modularization



[1998-2002 Palo Alto Research Center Incroporated]

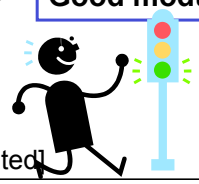
Good modularity

URL pattern matching



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

Good modularization

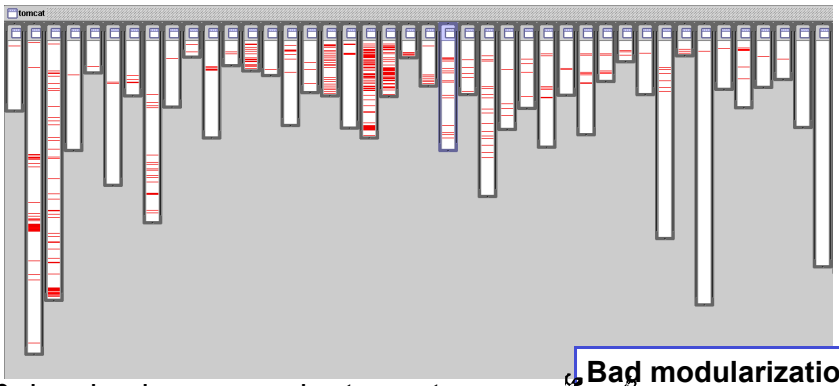


[1998-2002 Palo Alto Research Center Incroporated]

7

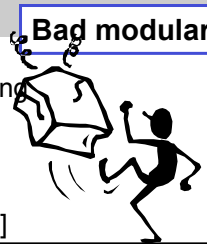
Crosscutting Concerns Affect Modularization

logging is not modularized



Bad modularization

- 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

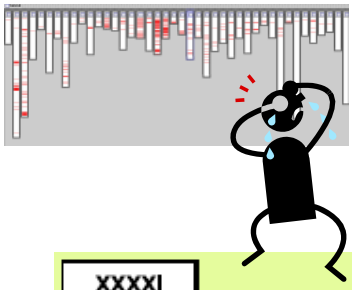


[1998-2002 Palo Alto Research Center Incroporated]

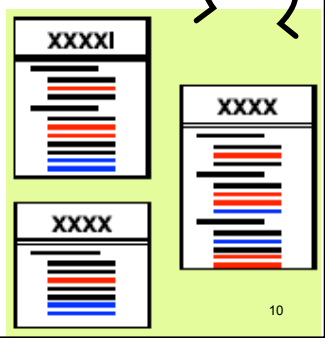
8

Resulting Problems

- **Scattering**
  - The specification of one property is **not encapsulated** in a single requirements unit, e.g., a viewpoint, a use case.



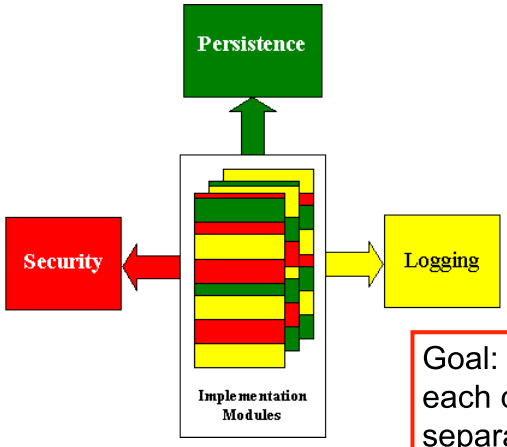
- **Tangling**
  - Each requirements unit contains descriptions of **several properties** or different functionalities



Consequences

- Redundancy
- Difficult to understand each concern and each module
- Difficult to evolve each module
- Reduced reuse
- Increased developing time and cost

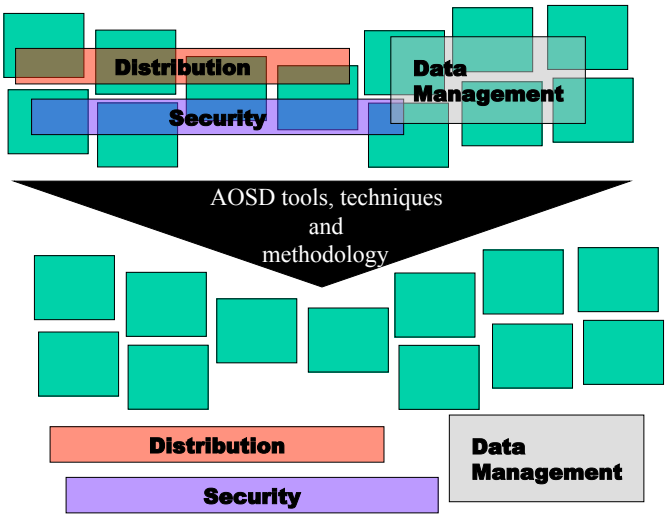
Crosscutting concerns



[R. Laddad (*JavaWorld*)**]**

Goal: to encapsulate each concern in a separate module (known as *aspect*)

Aspect-Oriented Software Development (AOSD)



A Definition of AOSD

- **AOSD**: systematic *identification*, *modularisation*, *representation* and *composition* of crosscutting concerns [1]

[1] Rashid, A., Moreira, A., Araujo, J. "Modularisation and Composition of Aspectual Requirements", Proceedings of 2nd International Conference on Aspect-Oriented Software Development, ACM, 2003.

Potential Benefits of AOSD

- Improved ability to reason about problem domain and corresponding solution
- Reduction in application code size, development costs and maintenance time
- Improved code reuse
- Requirements, architecture and design-level reuse
- Improved ability to engineer product lines
- Context-sensitive application adaptation
- Improved modelling methods

Crosscutting: The Tracing Concern

```
class A {
  // some attributes
  void m1( ) {
        System.out.println("Entering A.m1( )");
        // method code
        System.out.println("Leaving A.m1( )");
  }

  String m2( ) {
        System.out.println("Entering A.m2( )");
        // method code
        System.out.println("Leaving A.m2( )");
        // return a string
  }
```

```
class B {
  // some attributes
  void m2( ) {
        System.out.println("Entering B.m2( )");
        // method code
        System.out.println("Leaving B.m2( )");
        }

  int m3( ) {
        System.out.println("Entering B.m3( )");
        // method code
        System.out.println("Leaving B.m3( )");
        // return an integer
  }
```


Wouldn't it be Nice if …

```
class A {
  // some attributes
  void m1( ) {
        // method code
}

  String m2( ) {
        // method code
        // return a string
  }
```

```
class B {
  // some attributes
  void m2( ) {
        // method code
        }

  int m3( ) {
        // method code
        // return an integer
  }
```

aspect Tracing {

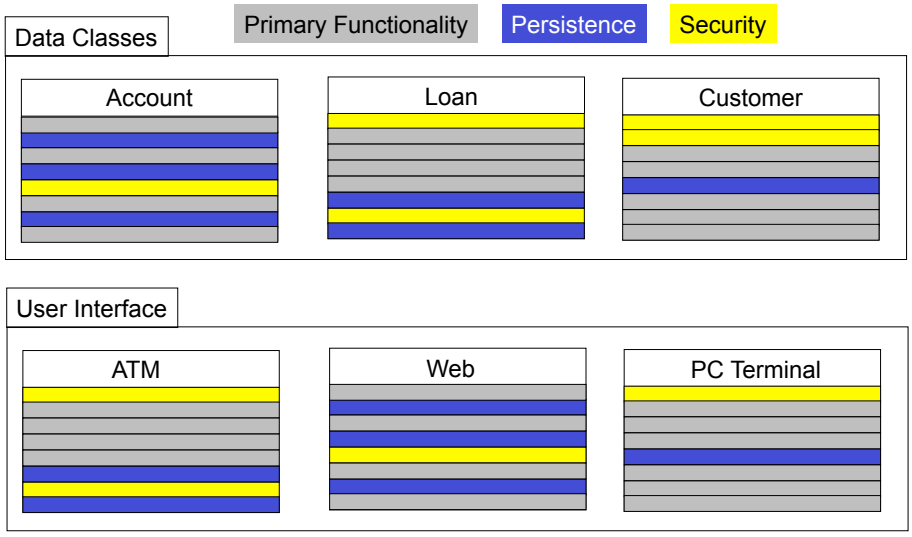
when someone ***calls*** these <u>methods</u>

before the call {System.out.println("Entering " + methodSignature);}

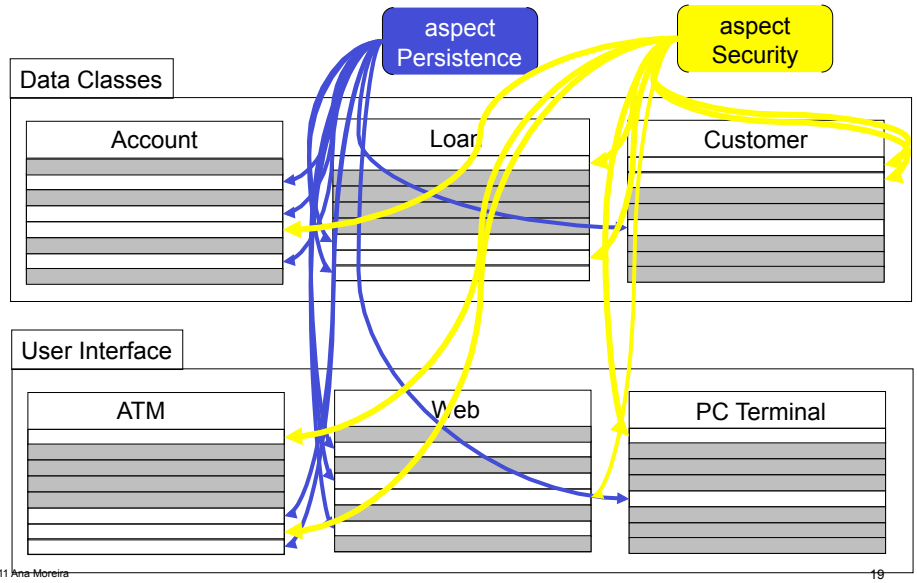
after the call {System.out.println("Leaving " + methodSignature);}

}

Tangling and Scattering



Wouldn't it be Nice if …



The Notion of a Join Point

```
class A {
  // some attributes
  void m1( ) {
        // me      ode
        // retu      ring
  }
```

```
class B {
  // some attributes
  void m2( ) {
        //
  int m
  }
```

Type of Join Point

Specific Join Points in this Program that we are Interested in

```
aspect Tracing {

  when someone calls these methods

  before the call {System.out.println("Entering " +  methodSignature);}

  after the call {System.out.println("Leaving " +  methodSignature);}
}
```


AOSD: main steps

1. **Decomposition**: Identify *crosscutting concerns*
2. **Specification/Implementation**: Specify/ implement each *concern* in a separate module
3. **Compostion** (*weaving*): Define composition rules by defining composition units

AOP

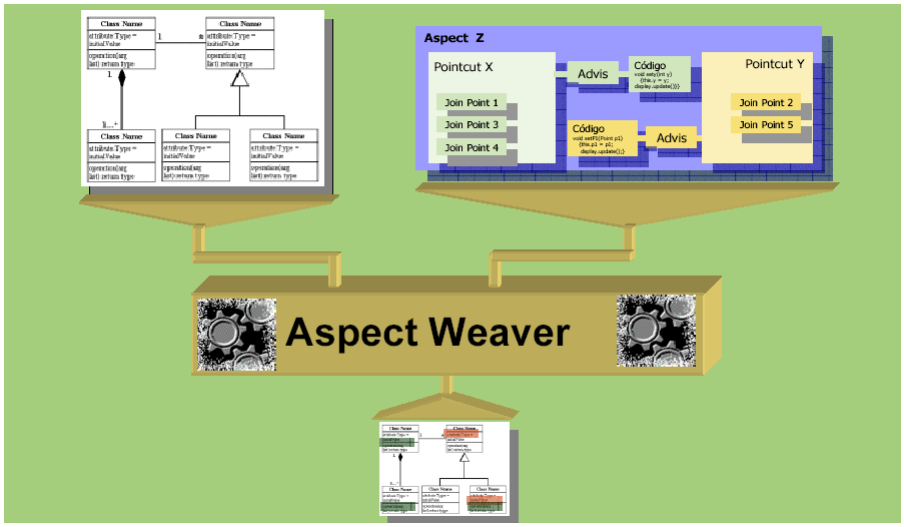
- **AspectJ** (1997) http://aspectj.org/
- **Composition filters** (1991) [Bergmans and Aksit]
- **DemeterJ/DJ** (1993) [Lieberherr, Orleans, and Ovlinger]
- **Hyper/J** (1999) [Ossher and Tarr]
- **CaesarJ** http://caesarj.org/
- **Apostle**, Aspect Programming em Smalltalk
- **AspectC,** uma extensão para C
- **AspectC++,** uma extensão para C++
- **JAC, Java Aspect Component** [Pawlak, L. Seinturier, L. Duchien, and G. Florin]

AspectJ

- Extension to Java
- Developed at Xerox Park por Gregor Kyczales
- Integrated in Eclipse since 2002 www.eclipse.org/aspectj

[Gregor Kizales]

AspectJ: code generation



AspectJ

- Join points
- Pointcuts
- Advices
- Aspects
- Aspect weaving

AspectJ

Joint points

- *Join points*: well-defined points in the execution of a program
  - Method call, Method execution
  - Constructor call, Constructor execution
  - Static initializer execution
  - Object pre-initialization, Object initialization
  - Field reference, Field set
  - Handler execution
  - Advice execution

AspectJ

Pointcuts

- A set of join point, plus, optionally, some of the values in the execution context of those join points.
- Can be composed using boolean operators || , &&
- Matched at runtime

Language

Advice

- Method-like mechanism used to declare that certain code should execute at each of the join points in the pointcut.
- Advices:
  - before
  - around
  - after

Aspect weaving

- Aspect weaving: makes sure that applicable advice runs at the appropriate join points.
- In AspectJ, almost all the weaving is done at compile-time to expose errors and avoid runtime overhead.

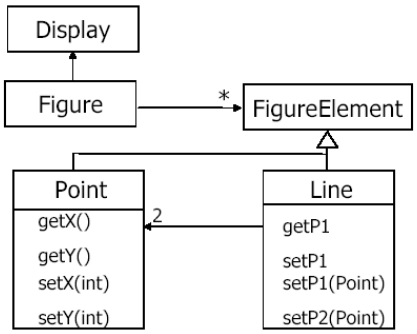
A guide tour of AspectJ

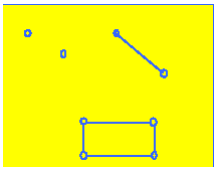
- C has “hello word”
- Lisp/Scheme have the “factorial” function
- Smalltalk has the “Counter” class
- Java has the “Observer” pattern
- AspectJ has the “figure editor” system

Figure editor example

- A *figure* consists of several *figure elements*. A figure element is either a *point* or a *line*. Figures are drawn on *Display*. A point includes X and Y coordinates. A line is defined as two points.

Crosscutting concern (1)



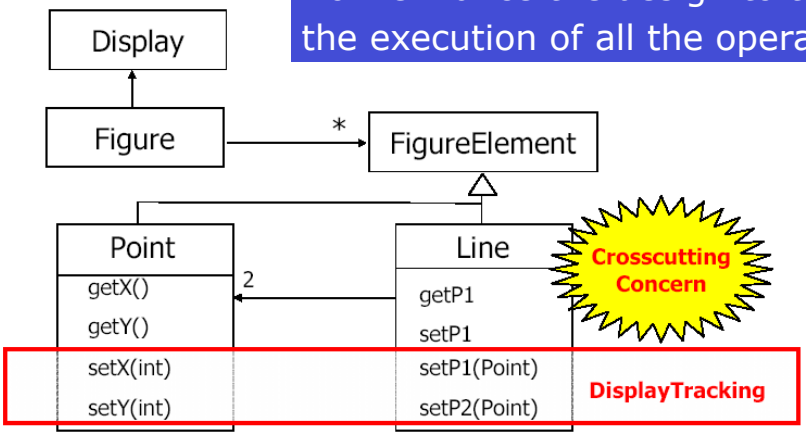


Components are
- Cohesive
- Loosely Coupled
- Have well-defined interfaces (abstraction, encapsulation)

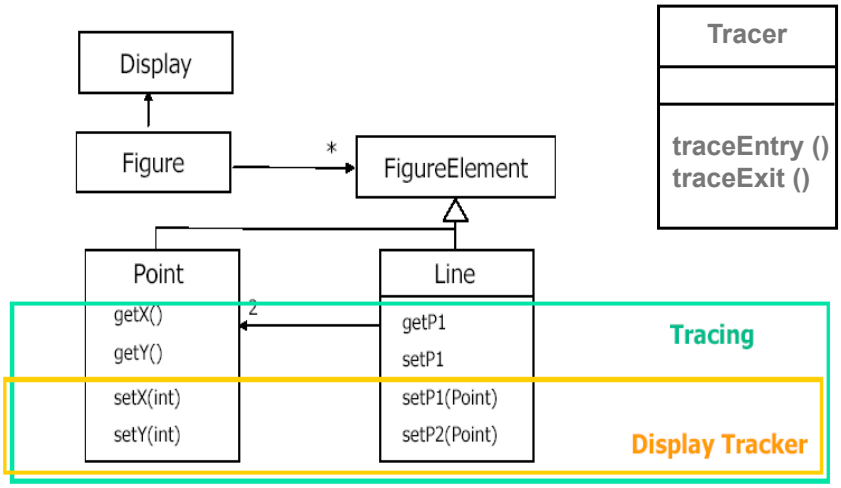
Well done!
Now I would like an extension. Notify *ScreenManager* if a *FigureElement* moves

Crosscutting concern (2)

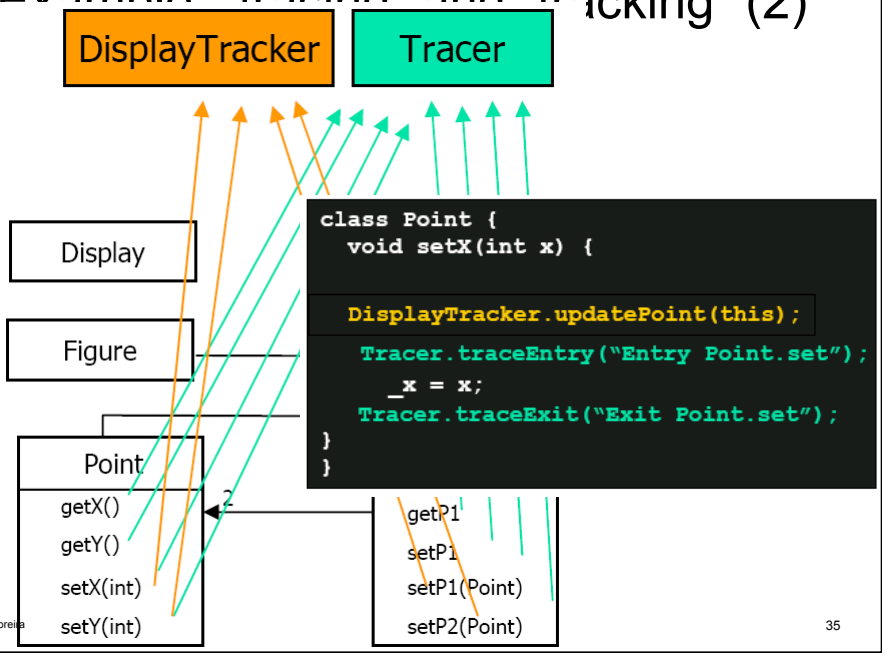
Now enhance the design to trace the execution of all the operations



Example: “tracing” and “tracking”(1)



Example: “tracing” and “tracking” (2)

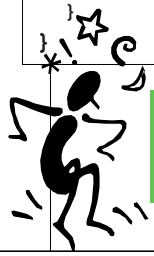


Without AOP

```
class Line {
  private Point _p1, _p2;

  Point getP1() { return _p1; }
  Point getP2() { return _p2; }

  void setP1(Point p1) {
    Tracer.traceEntry("entry setP1");
    _p1 = p1;
    Tracer.traceExit("exit setP1");
  }

 void setP2(Point p2) {
    Tracer.traceEntry("entry setP2");
    _p2 = p2;
    Tracer.traceExit("exit setP2");
  }
```

```
class Point {
  private int _x = 0, _y = 0;

  int getX() { return _x; }
  int getY() { return _y; }

  void setX(int x) {
   Tracer.traceEntry("entry setX");

    _x = x;
   Tracer.traceExit("exit setX")
  }
  void setY(int y) {
   Tracer.traceEntry("exit setY");

    _y = y;
   Tracer.traceExit("exit setY");
  }
```

```
class Tracer {

  static void traceEntry(String str)
  {
        System.out.println(str);
  }
  static void traceExit(String str)
  {
        System.out.println(str);
  }
}
```



Tangled code!

Scattered code!

With AOP

```
class Line {
  private Point _p1, _p2;

  Point getP1() { return _p1;
  Point getP2() { return _p2;

  void setP1(Point p1) {
    _p1 = p1;
  }
  void setP2(Point p2) {
    _p2 = p2;
  }
}
```

```
class Point {
  private int _x = 0, _y = 0;

  int getX() { return _x; }
  int getY() { return _y; }

  void setX(int x) {
    _x = x;
  }
  void setY(int y) {
    _y = y;
  }
}
```

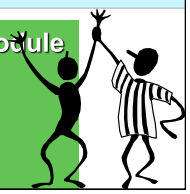
```
aspect Tracing {

 pointcut traced():
   call(* Line.*) ||
   call(* Point.*);

 before(): traced() {
  println("Enter " +
    thisJoinPointStaticPart.getSignature());
 }

 after(): traced() {
  println("Exit " +
    thisJoinPointStaticPart.getSignature());
 }
}
```

Aspect is defined in a separate module

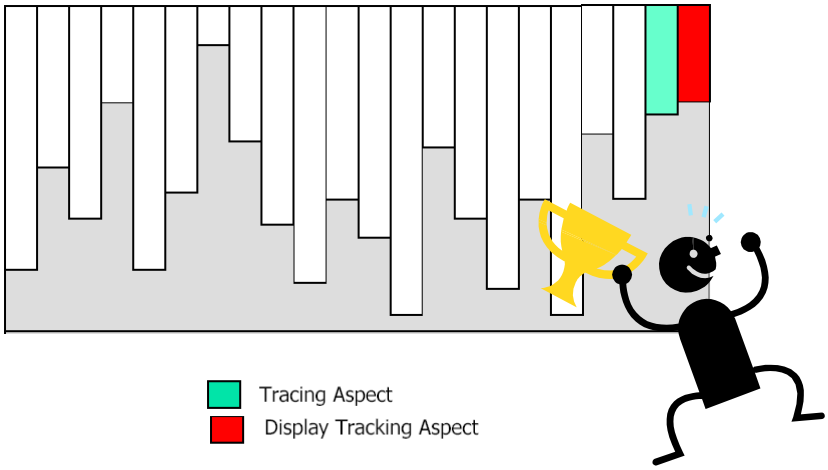
Crosscutting is localized

No scattering; No tangling

Improved modularity



Aspects modularized



Main Value of Aspect-Orientation

- **Abstraction**: abstract away from the details of how that crosscutting concern, or *aspect*, might be scattered and tangled with the functionality of other modules in the system
- **Modularization**: keep crosscutting concerns separated regardless of how they affect or influence various other modules in the system, so then we can reason about each module in isolation – **Modular Reasoning**
- **Composition**: the various modules need to relate to each other in a systematic and coherent fashion so that one may reason about the global or emergent properties of the system – **Compositional Reasoning**

Pointers to Further Reading

- AOSD Wiki at: *http://www.aosd.net*
- Introduction to AOSD White Paper and AOSD Ontology available at: *http://www.aosd-europe.net*
- Communications of the ACM, Special Section on AOP, *44(10), October 2001*
- IEEE Software, Special Section on AOP, *23(1), Jan/Feb 2006*
- Aspect-Oriented Software Development, *Filman, Elrad, Clarke, Aksit (eds), Addison-Wesley 2004*
- Discovering Early Aspects, *Baniassad, Clements, Araujo, Moreira, Rashid, Tekinerdogan, IEEE Software 23(1), Jan/Feb 2006*
- Special Issue on Early Aspects, *IEE proceedings - Software Engineering - Volume 151, Issue 04, August 2004, (Rashid, Moreira, Tekinerdogan (eds))*