On Modular Reasoning, Information Hiding and Aspect-Oriented Programming

http://www.st.informatik.tu-darmstadt.de/

Prof. Dr. Mira Mezini
Technische Universität Darmstadt
Fachbereich Informatik

AOP is …

- AOP and modular reasoning
- AOP and information hiding
- Speculations on the future

OOP Is…

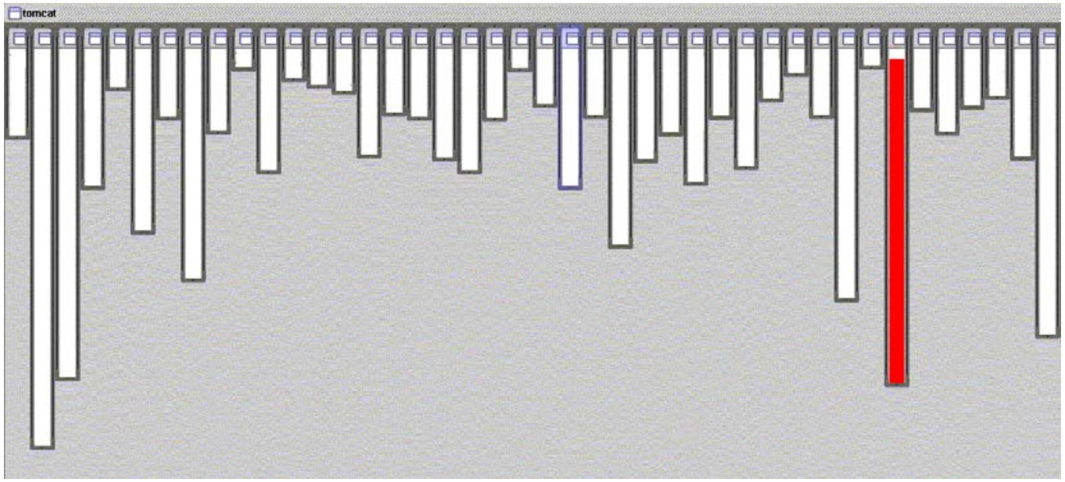
- a “way of thinking”
  - objects, classification hierarchies
- supporting mechanisms
  - classes, encapsulation, polymorphism…
- allows us to
  - make code look like the design
  - improves design and code modularity

many other benefits build on these

- many possible implementations
  - style, library, ad-hoc PL extension, integrated in PL

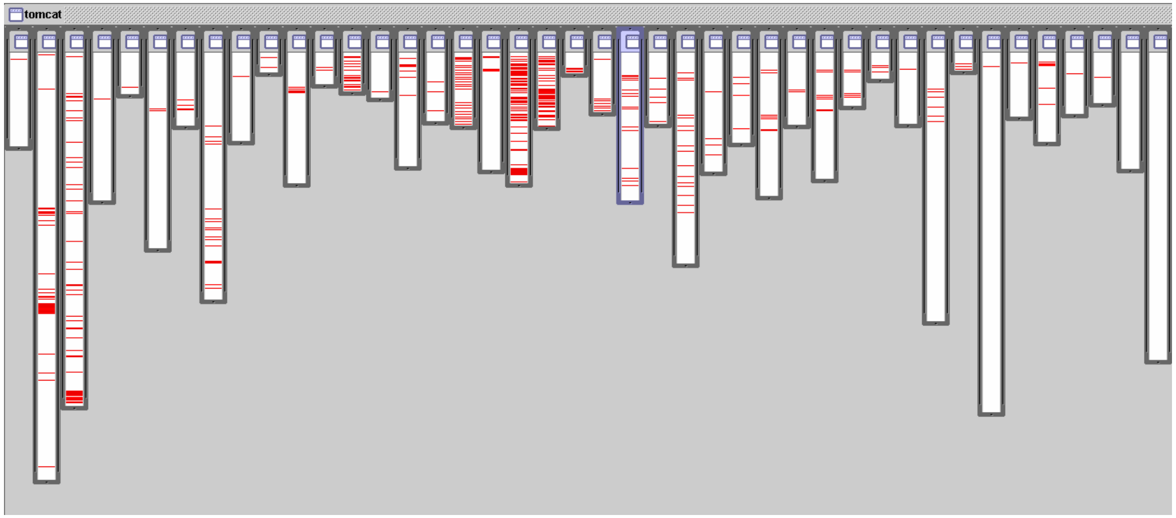
- Code implementing a concern is *modular* if:
  - it is textually local and not tangled with other concerns
  - there is a well-defined interface
  - the interface is an abstraction of the implementation
  - the interface is enforced

XML parsing in apache.tomcat is modular



Some Concerns “don’t fit” with OOP

session expiration in apache.tomcat

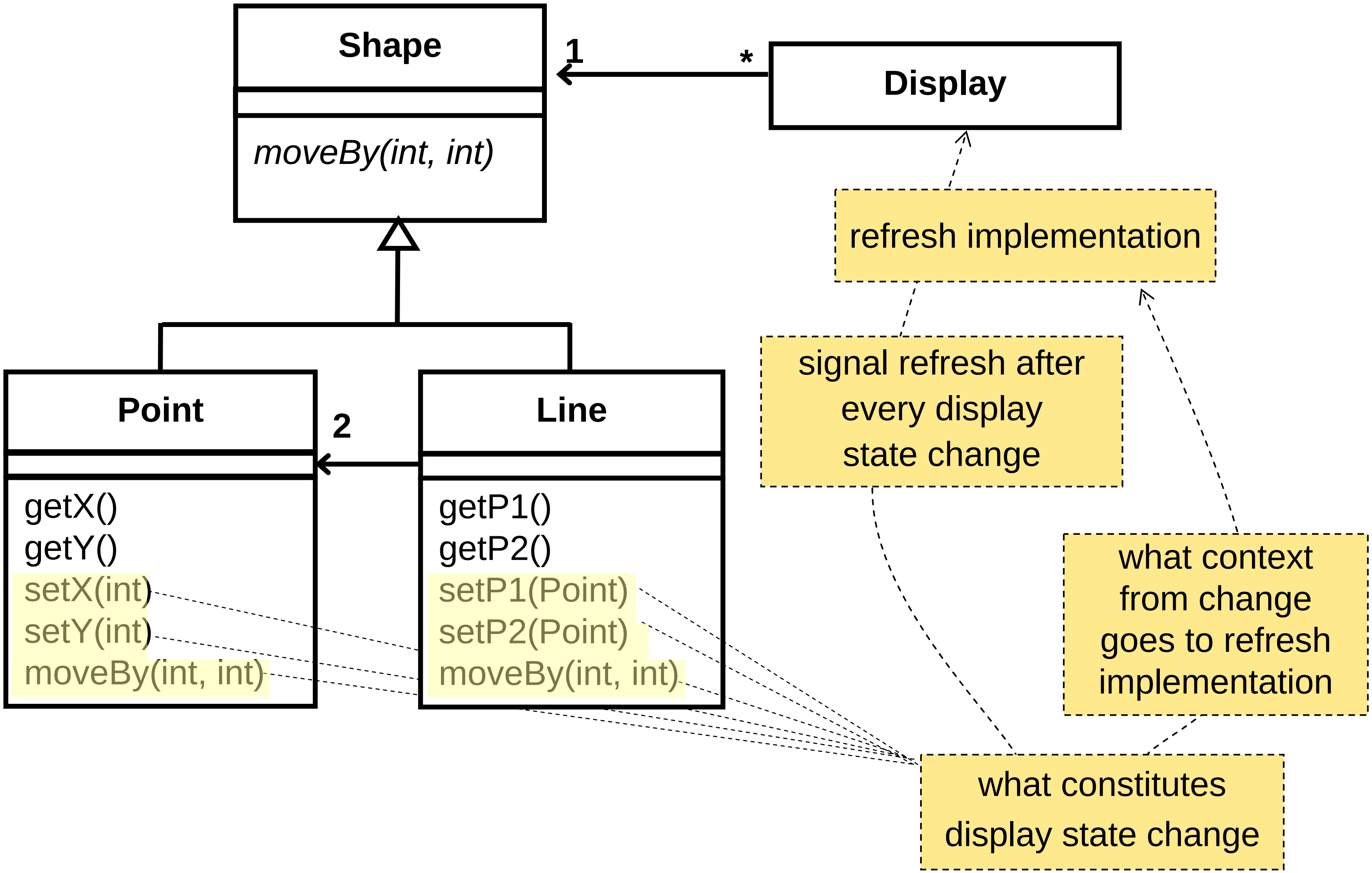


- testing pre- / post-conditions
- enforcing/checking adherence to architecture / design styles and rules
- co-ordination between objects, e.g., grouping semantics
- …

- exception handling
- performance monitoring and optimizations
- synchronization
- authentication, access control
- transaction & persistence management

Some Concerns "don't fit" with OOP

e.g., model view synchronization: whenever state changes that affects the display, refresh the latter

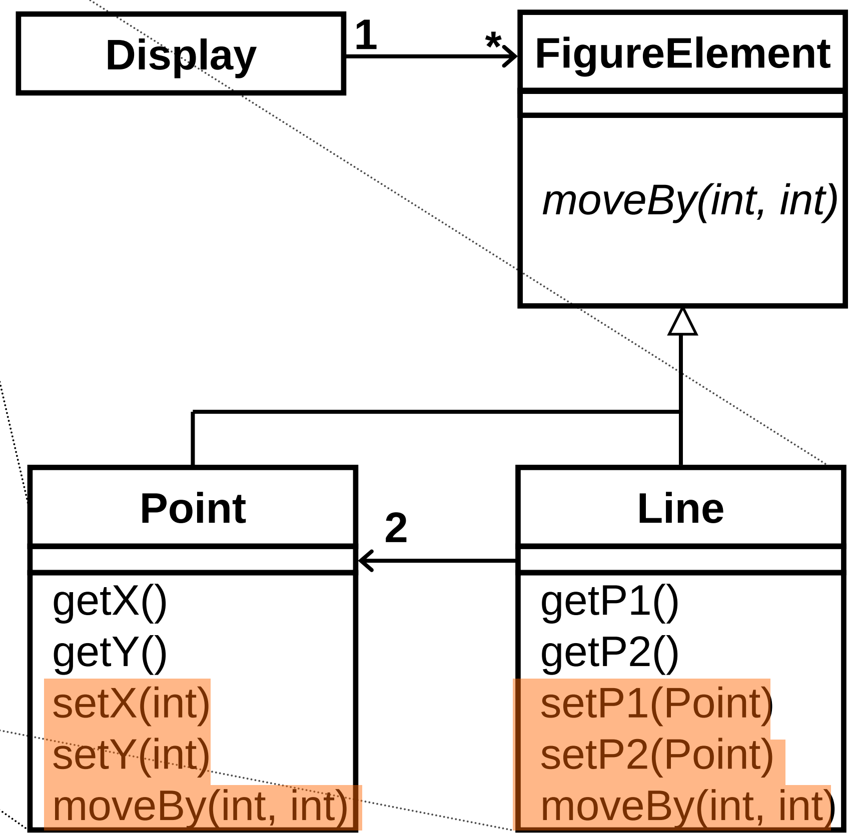


Model View Synchronization “doesn’t fit”


```
class Line extends FigureElement {
    private Point p1, p2;

   Point getP1() { return p1; }
   Point getP2() { return p2; }

   void setP1(Point p) {
     this.p1 = p;
     display.update(this);
   }
   void setP2(int p) {
     this.p2 = p;
     display.update(this);
   }
   void moveBy(int dx, int dy) {
     p1.x += dx; p1.y += dy;
     p2.x += dy; p2.y += dy;
     display.update(this);
   }
}
```



with AOP it fits better

```
class Point
  void setX(int nx) {
    x = nx;
  }
  void setY(int ny) {
    x = ny;
  }
  ...
```

```
class Line {
  void setP1(Point np1) {
    p1 = np1;
  }
  void setP2(Point np2) {
    p2 = np2;
```

```
aspect Updat
  pointcut c
          |
          |
          |
          |
      && t

  after(Shap
  { Display.
}
```

```
aspect UpdateSignaling {
  pointcut change(Shape s):
          (execution(void Shape.moveBy(..)
        || execution(void Shape+.set*(..)))
     && this(s);

  pointcut topChange(Shape s):
            stateChange(Shape s)
          && !cflowbelow(stateChange(Shape s))

  after(Shape s) returning: change(Shape s)
  { Display.update(s); }
}
```

- The aspect is
  - Localized and has a clear interface
- The classes are
  - better localized (no invasion of updating)
- Code modularity helps design modularity
- Forest versus trees
  - the global invariant is explicit, clear, modular
  - the local effects can be made clear by IDE

AJDT

		localized	interface	abstraction	enforced
non AOP	display updating	no	n/a	n/a	n/a
	Point, Line	medium	medium	medium	yes
AOP	UpdateSignaling	high	high	medium	yes
	Point, Line	high	high	high	yes

- “a way of thinking”
  - behavioral slices, partial models, crosscutting structure
- supporting mechanisms
  - join points, pointcuts, advice + virtual classes, open classes, intertype declarations…
- allows us to
  - make code look like the design
  - improve design and code modularity

many other benefits build on these

- many possible implementations
  - style, library, ad-hoc PL extension, integrated in PL

AOP is …

AOP and modular reasoning

AOP and information hiding

Speculations on the future

What’s a Developer to Believe?

- *Aspect-oriented programming improves software modularity by enabling modular implementation of crosscutting concerns.*

  - anonymous AOP researcher

- *AOP is anti-modular.*

  - anonymous non-AOP researcher

  – *I can’t understand the Point and Line in isolation*
  – *advice can violate client assumptions*
  – *AOP violates information hiding, since aspects may refer to implementation details of the components*

Questions Addressed in [KiczalesMezini05]

- **Does AOP improve or harm modularity?**
  - in presence of crosscutting concerns (CCC) improves modularity of aspects and non-aspects
  - does not harm modularity otherwise

- **If AOP is modular, what is modularity?**
  - nearly the same idea and mechanisms as before
  - except for how interfaces are determined
    - aspect-aware interfaces
    - interface depends on overall system configuration

```
Point implements Shape
  int getX();
  int getY();
  void setX(int);
  void setY(int);
  void moveBy(int, int);

Line
  <similar>
```

```
class Line {
  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;
  }
}
```

```
class Point {
   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;
  }
}
```

```
aspect UpdateSignaling {

  pointcut change(Shape shape):
    this(shape) &&
    (execution(void Shape.moveBy(int, int) ||
     execution(void Shape+.set*(*)));


  after(Shape s) returning: change(s) {
    Display.update(s);
  }
}
```

- Aspect cuts extended interface
  - through **`Point`** and **`Line`**
- Interface of **`Point`** and **`Line`**
  - depend on presence of aspects
  - and vice-versa

```
Point implements Shape
  int getX();
  int getY();
  void setX(int): UpdateSignaling – after returning change();
  void setY(int): UpdateSignaling – after returning change();
  void moveBy(int, int): UpdateSignaling – after returning change();

Line
  Point p1, p2;
  Point getP1();
  Point getP2();
  void moveBy(int, int): UpdateSignaling – after returning change();

UpdateSignaling
   after returning: change():
    Point.setX(int), Point.setY(int), Point.moveBy(int, int),
    Line.moveBy(int, int);
```

Interface Depends on Deployment

- **Main message to take away so far:**
  - Aspects contribute to the interface of the classes
  - Classes contribute to the interface of the aspects

- **Implication:**
  - To fully know interfaces of modules in a system,
    - a configuration is needed
    - a run through the modules to analyze crosscutting
  - this can be mostly done automatically
    - since the crosscutting structure is explicit,

- Code implementing a concern is *modular* if:
  - it is textually local
  - it is not tangled with other concerns
  - **there is** a well-defined interface
  - the interface is an abstraction of the implementation
  - the interface is enforced
  - the module can be automatically composed

vs. “it has a well-defined interface”

Intermediate Conclusions

- This might sound anti-modular
  - But: fundamentally, display update signaling is crosscutting.
- With AOP,
  - its interface cuts through the classes,
  - the structure of that interface is captured declaratively,
  - the actual implementation is modularized
- Without AOP,
  - the structure of the interface is implicit and the actual implementation is not modular.

- ***Modular reasoning***
  - make decisions about a module by studying only
    - its implementation
    - its interface
    - interfaces of other modules referenced in the module's implementation or interface
- ***Expanded modular reasoning***
  - also study implementations of referenced modules
- ***Global reasoning***
  - have to examine all the modules in the system

- In the example `x` and `y` fields of `Point` are public
- The programmer decides to make `x` and `y` private

```
class Line {
  ...
  void moveBy(int dx, int dy) {
    p1.x += dx; p1.y += dy;
    p2.x += dy; p2.y += dy;
  }
}
```

(s)he must ensure the system continues to work as before.

- We compare :
  - reasoning with traditional interfaces about the non-AOP code against
  - reasoning with AAIs about the AOP code.

- Both implementations start out the same
  - define accessors
  - global reasoning to find references to fields
    - change to use accessors
    - simple change to **`Line.moveBy`** method

```
void moveBy(int dx, int dy) {
  p1.x += dx;
  p1.y += dy;
  ...
}
```

```
void moveBy(int dx, int dy) {
  p1.setX(p1.getX() + dx);
  p1.setY(p1.getY() + dy);
  ...
}
```

What is the effect of this change?

What kind of reasoning do I need to reach a conclusion?

- Two pieces of information are needed:
  - a specification of the invariant:
    - "*update after any top-level change of a shape*"
  - structure of the update signaling to infer that the invariant would be violated

- Nothing in **`Line`** is likely to describe the invariant.

- Given the call **`Display.update()`**, the programmer might look at **`Display`**
  - assume, optimistically, that the documentation for the **`update()`** includes a description of the invariant.
  - expanded modular reasoning with one step leads the programmer to the invariant

- Discovering the structure of update signaling requires
  - at least further expanded modular reasoning
  - in general, global reasoning

Now that I discovered the problem, how do I recover?

- Add non-update-signaling methods to be called by **moveBy**?
  - … maintenance nightmare

```
class Line {
  void moveBy(int dx, int dy) {
    p1.nonSignalingSetX(...);
    p1.nonSignalingSetY(...);
    ...
  }
}
```

```
class Point {
   ...
  void setX(int nx) {
     x = nx;
     Display.update();
  }
  void nonSignalingSetX(int nx) {
     x = nx;
  }
}
```

- The best I can do is probably to let **x** and **y** public
  - … probably the reason why they were package public!

- The interface of **`UpdateSignaling`** includes the complete structure of what method executions will signal updates.
  - modular reasoning provides this information

- Once the programmer understands that the change is invalid,
  - the proper fix is to use **`cflowbelow`**:

```
after() returning:
        change() && !cflowbelow(change()) {
     Display.refresh(); }
```

- A proper formulation of the invariant would have been in terms of **`cflowbelow`** to start with
  - Such a formulation would absorb the change

Intermediate Conclusion

- The cost of AOP:
  - We must know the set of modules with which a given module will be deployed to know its interface

- But, for CCCs programmers inherently have to pay this cost:
  - They have to know about the total deployment configuration to do the global reasoning required for crosscutting concerns.

- By using AOP, they get modular reasoning benefits back, whereas not using AOP they do not.

The’re of Course Open Issues…

- form of the interface …
  - the extensional version of it could be not just the affected methods, but how they matched the pointcut?
  - or what part they matched?
  - or…
- Means of restricting aspects
  - suggests restrictions should be associated with configuration, not modules directly
- Means of expressing pointcuts
  - would like to express pointcuts without reference to names

- Two points to make in this regard:
  - AOP does not conflict with existing approaches for stating and enforcing behavioral sub-typing:
    - Approaches exist that extend JML to state and check pre- / post-conditions for advice
    - Work by Krishnamurthi et al. and Katz et al. on modular verification of advice
  - AOP comes with means to express global invariants to be imposed (also on aspects)

```
public aspect ShapeChange {
  public pointcut joinpoint(Shape s):
   target(s)
   &&   (call(void Shape+.set*(..))
      || call(void Shape+.moveBy(..))
      || call(Shape+.new(..)));

  public pointcut topLevelJoinpoint(Shape s):
      joinpoint(s)
   && !cflowbelow(joinpoint(Shape));

  protected pointcut staticscope(): within(Shape+);

  protected pointcut staticmethodscope():
      withincode (void Shape+.set*(..))
   || withincode(void Shape+.moveBy(..))
   || withincode(Shape+.new(..));
...
```

```
// PROVIDES: matches only calls to Shape mutators
declare error:
  (!staticmethodscope()
    && set(int FigureElement+.*)):
  “Contract violation: must set Shape”
  + “ field inside setter method!”;

 // REQUIRES: advisers must not change state

before():
   cflow(adviceexecution())
   && joinpoint(Shape) {
      ErrorHandling.signalFatal(
        “Contract violation:”
      + “ advisor of ShapeChange cannot”
      + “ change Shape instances”);
  }
}
```

Display Updating Aspect


```
public aspect DisplayUpdate {
  after():
    ShapeChange.topLevelJoinpoint(Shape s) {
          updateDisplay();
    }

   public void updateDisplay() {
    Display.update();
   }
}
```

- AOP is …
- AOP and modular reasoning
- AOP and information hiding
- Speculations on the future

```
pointcut change():
        call(Point.setX(int))
     || call(void Point.setY(int))
     || call(void Shape+.moveBy(int, int));
```

instead of specifying *WHAT* the crosscutting structure is,
this pointcut describes *HOW* it appears in the concrete syntax of the program

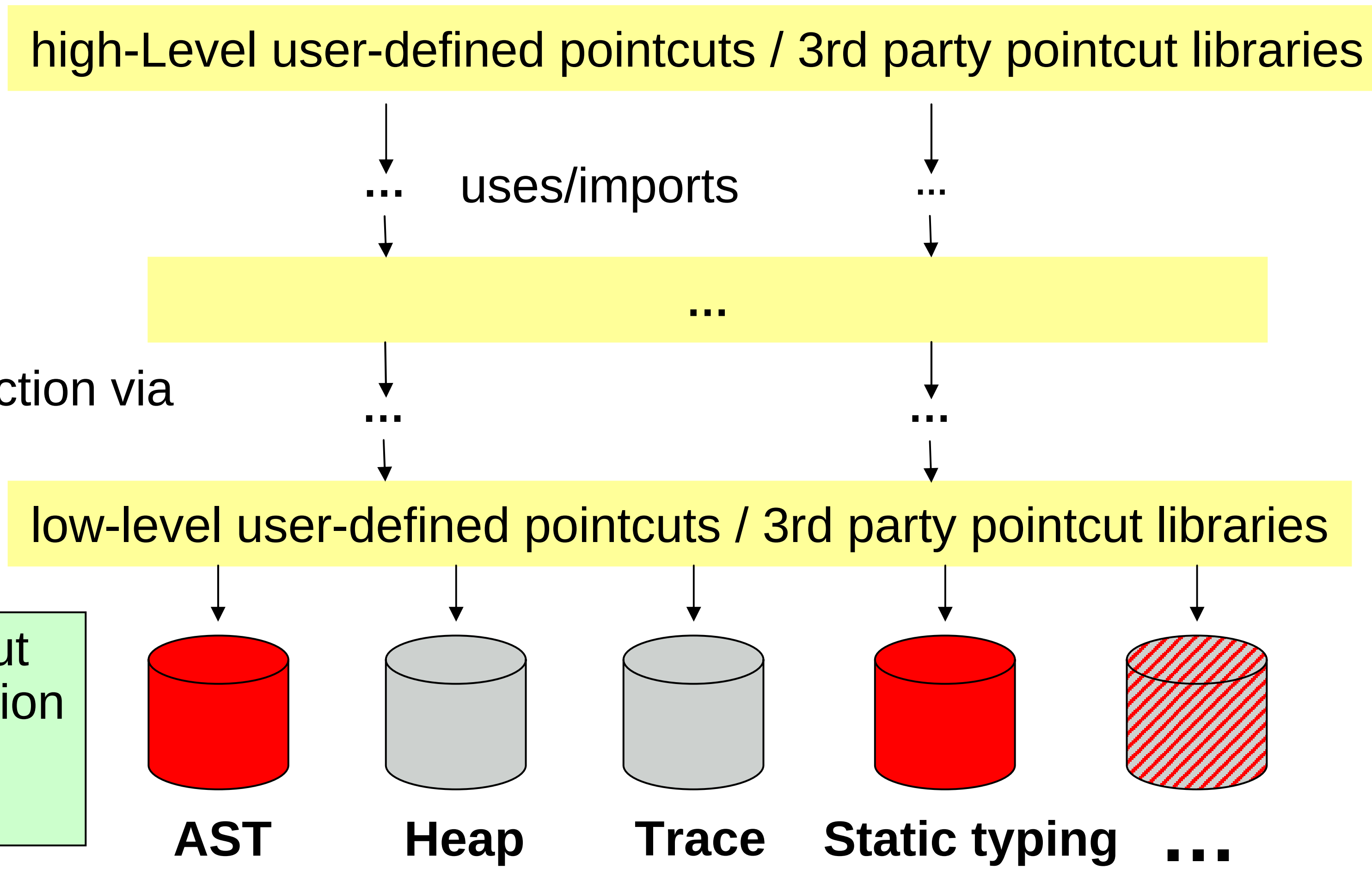
Wanted: ***"after data changes that was previously read during the most recent draw of a display, update that display"***

Robust.
Minimal knowledge about implementation details of figure elements.

Precise.
Avoids unnecessary updates,
e.g., after calls to **`setX`** modifying an **`x`** not read in control flow of **`draw`**

The Programming Language ALPHA

encode pointcuts as logic queries; pointcut “fires” if query has non-empty result



Store facts about program execution in an extensible list of logic DBs

Ostermann, Mezini, Bockisch, ECOOP 2005

Pointcuts in ALPHA

```
display d;

// cflowreach pointcut
before set (O, F, _),
       get (T1, _, O, F, _),
       calls (T2, _, @this.d, draw, _),
       cflow(T1, T2),
       reachable (O, d),
{ ... }
```

object specific pointcut

variables bound via unification

“after data changes that was read during the most recent redraw of a display, update that display”

Pointcuts in ALPHA

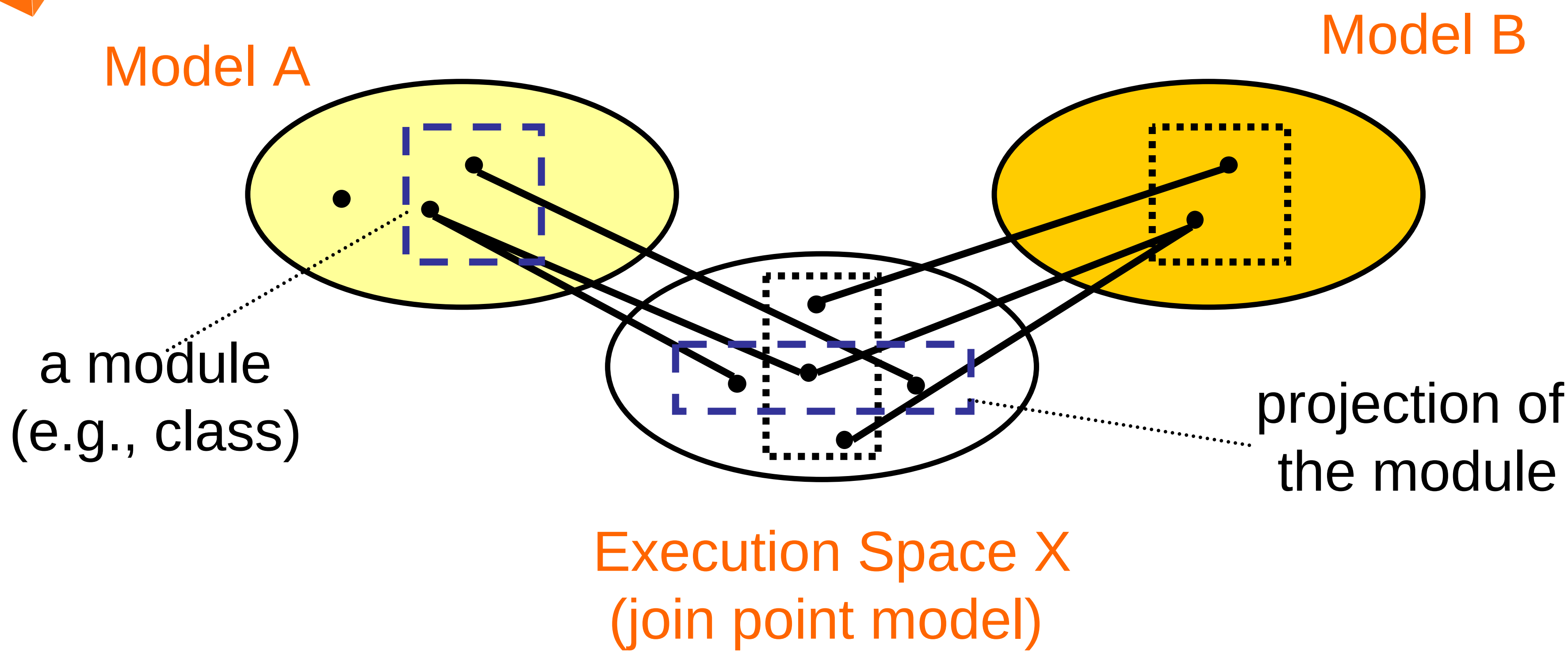
```
display d;

// cflowreach pointcut
before set (O, F, _),
       get (T1, _, O, F, _),
       calls (T2, _, @this.d, draw, _),
       cflow(T1, T2),
       reachable (O, d)
 { ... }
```

this module really “talks” about itself … about “its model” of the world

It doesn’t have an interface to shapes but rather to execution space …it pattern matches points in the execution

… or clusters them by some algorithm …



two models A&B crosscut when ***projections of their modules into X intersect & neither is a subset of the other***

Masuhara & Kiczales, ECOOP 2003

- Powerful extensible temporal quantification
  - precise, object-specific, history-aware, … use the data model that best suits
  - no need to build up complicated infrastructure
    - observer pattern infrastructure disappears in example
  - user-defined pointcuts, (domain-specific) pointcut libraries

- Extensible join point model
  - easy to expose new data, e.g., profiling information, resource usage, …

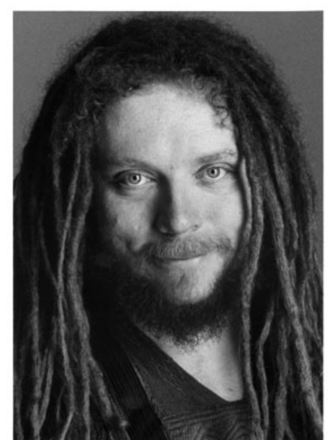
- Efficient implementation is challenging

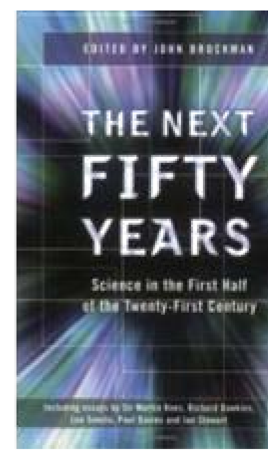
- AOP is …
- AOP and modular reasoning
- AOP and information hiding
- Speculations on the future

Lanier on Black-Box Abstraction…

- modules - expressions, functions, objects - little “black boxes”
  - relate to the rest through a well-defined IO interfaces (IO-wires)

- Intuition underlying communication between modules:
  - “sending pulses down a wire” - passing variables, messages
  - “single-point sampling of the world at the end of the wire” by algorithmic protocols





Lanier on Black-Box Abstraction…

- Programmers forced to stream intentions into sequential steps aligned with this pipeline view of the world
- Complex algorithmic protocols needed to give meaning to sequences of pulses
  - – accidental complexity!

Lanier: “world as a planet of the help desks in which human race will be largely engaged in maintaining very large software systems …”

Lanier's Surface Binding

- Components probe "measurable fundamental" properties of program execution and take decisions based on some evolving model of the world
  - components connected by "surfaces" sampled at several points in parallel
    - instead of "wires sampled at single points"
  - pattern classification and automatic maintenance of implicit confirmatory and predictive models
    - instead of sampling by algorithmic protocols

step towards more powerful binding?

this module really “talks” about itself … about “its model” of the world

```
display d;

// cflowreach pointcut
before set (O, F, _),
       get (T1, _, O, F, _),
       calls (T2, _, @this.d, draw, _),
       cflow(T1, T2),
       reachable (O, d)
 { ... }
```

It doesn’t have an interface to shapes but rather to execution space …it pattern matches points in the execution

… or clusters them by some algorithm …

referencing in natural communication

```
/**
* encodeStream converts stream of bytes into sounds.
* @param in stream of bytes to encode
* @param out stream of audio samples representing input
*/

encodeStream(InputStream input, OutputStream output) {

  while there is data in input: read N bytes from it,
       perform encodeDuration on those bytes, and write
       result into output

  if, however, after reading the input, the number of
       bytes read is less than N, then, before continuing
       with writing out, patch it with zeros.

}
```

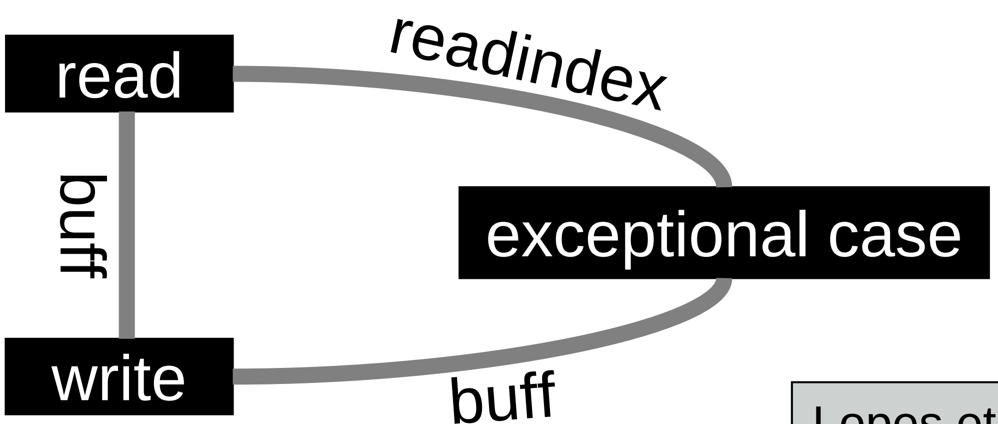
Crista Lopes et al. OOPSLA Onward 03

```
static void encodeStream(InputStream in, OutputStream out) {
  int readindex = 0;

  byte[] buff = new byte[N];

  while ( (readindex = in.read(buff)) == N) {
          out.write( Encoder.encodeDuration(buff) );
  }

  if (readindex > 0) {
     for (int i = readindex; i < N; i++) buff[i] = 0;
     out.write( Encoder.encodeDuration(buff) );
  }
}
```



Lopes et al. OOPSLA Onward 03

```
static void encodeStream(InputStream in, OutputStream out) {
  int readindex = 0;

  byte[] buff = new byte[N];

  while ( (readindex = in.read(buff)) == N) {
            out.write( Encoder.encodeDuration(buff) );
  }

  if (readindex > 0) {
     for (int i = readindex; i < N; i++) buff[i] = 0;
     out.write( Encoder.encodeDuration(buff) );
  }
}
```

The problem is much worse if one has to write things like

“after data changes that was read during the most recent draw of a display, update that display”

Lopes et al. OOPSLA Onward 03

Software Modularity Lab @ TUD

- Faculty:
  - Mira Mezini
  - Klaus Ostermann
- Research assistants
  - Ivica Aracic
  - Christoph Bockisch
  - Marcel Bruch
  - Anis Charfi
  - Tom Dinkelaker
  - Michael Eichberg
  - Vaidas Gasiunas
  - Slim Kallel
  - Sven Kloppenburg
  - Karl Klose
  - Thorsten Schäfer
  - Tobias Schuh
  - NN

Software Modularity Lab @ TUD

Current Research Focus

Divide and Conquer as a Construction Principle:
AO module concepts and expressive pointcut languages
Efficient compilation of AO languages and AO virtual machines
Aspect-oriented web service composition and middleware
Virtual types and advanced type systems for better supporting variations
Software product line engineering
Dynamically adaptable software

Two complementary ways to master software complexity.

Intelligent Software Development Environments
Open static analysis and development environments
Data mining for supporting framework-based development
Tailorable software information spaces

END