Concepts of Aspect-Oriented Languages

AOSD-Europe summer school

Lodewijk Bergmans, Mira Mezini, Johan Brichau, Jacques Noyé, Mario Südholt

Overview of AOP Languages Course

- Concepts of Aspect Languages *(Lodewijk Bergmans)*
  - Part I: Crosscutting Concerns
  - Part II: General Approach: Aspect Orientation
  - Part III: A concrete AOPL: AspectJ
  - Part IV: An overview of AOP Languages
  - Part V: AOSD Obstacles and Issues
- AOP and Reflection *(Jacques Noyé)*
- Hands-on
  - EAOP *(Mario Südholt)*
  - CaesarJ *(Mira Mezini)*

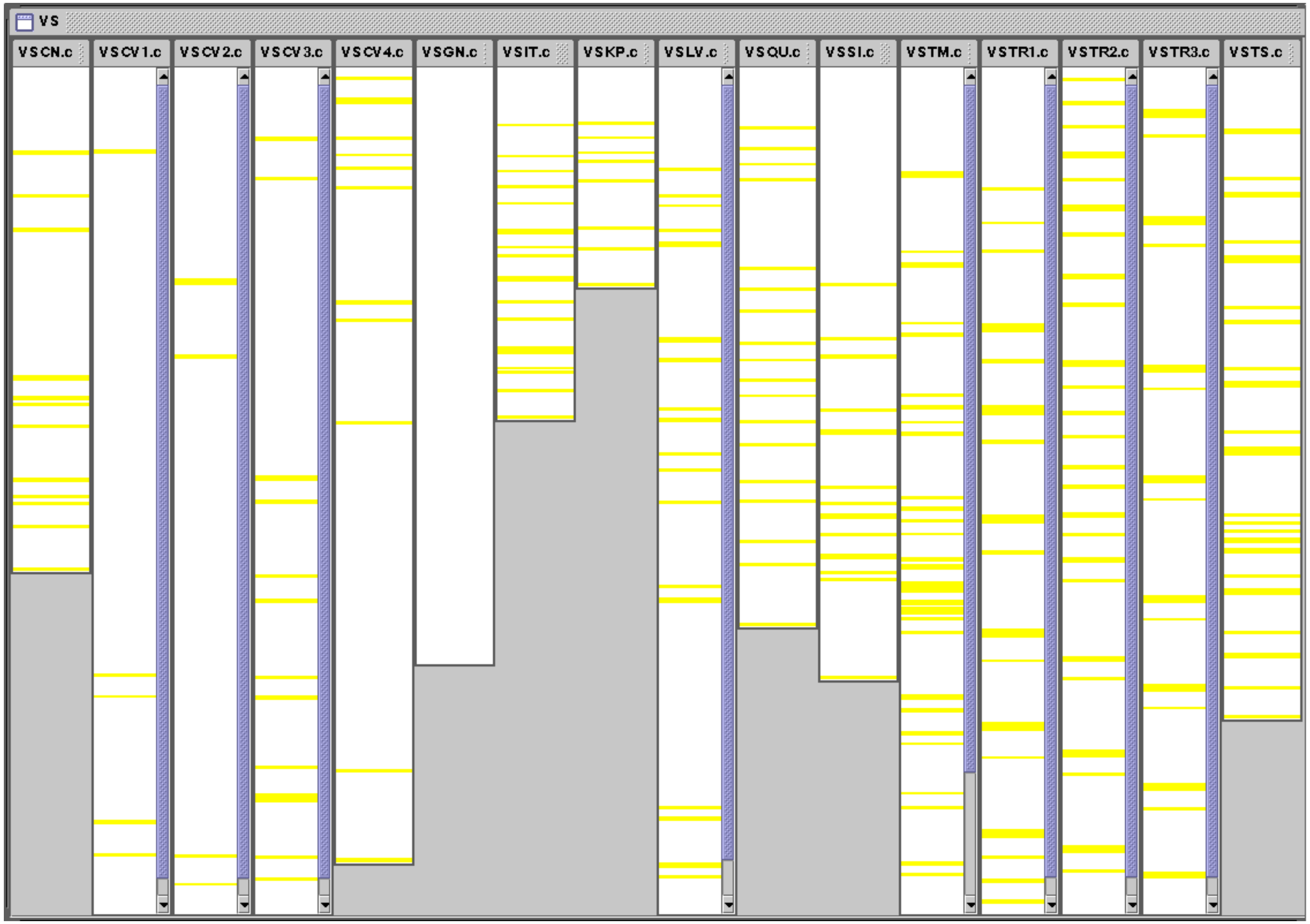
Part I: Crosscutting Concerns

Software Complexity

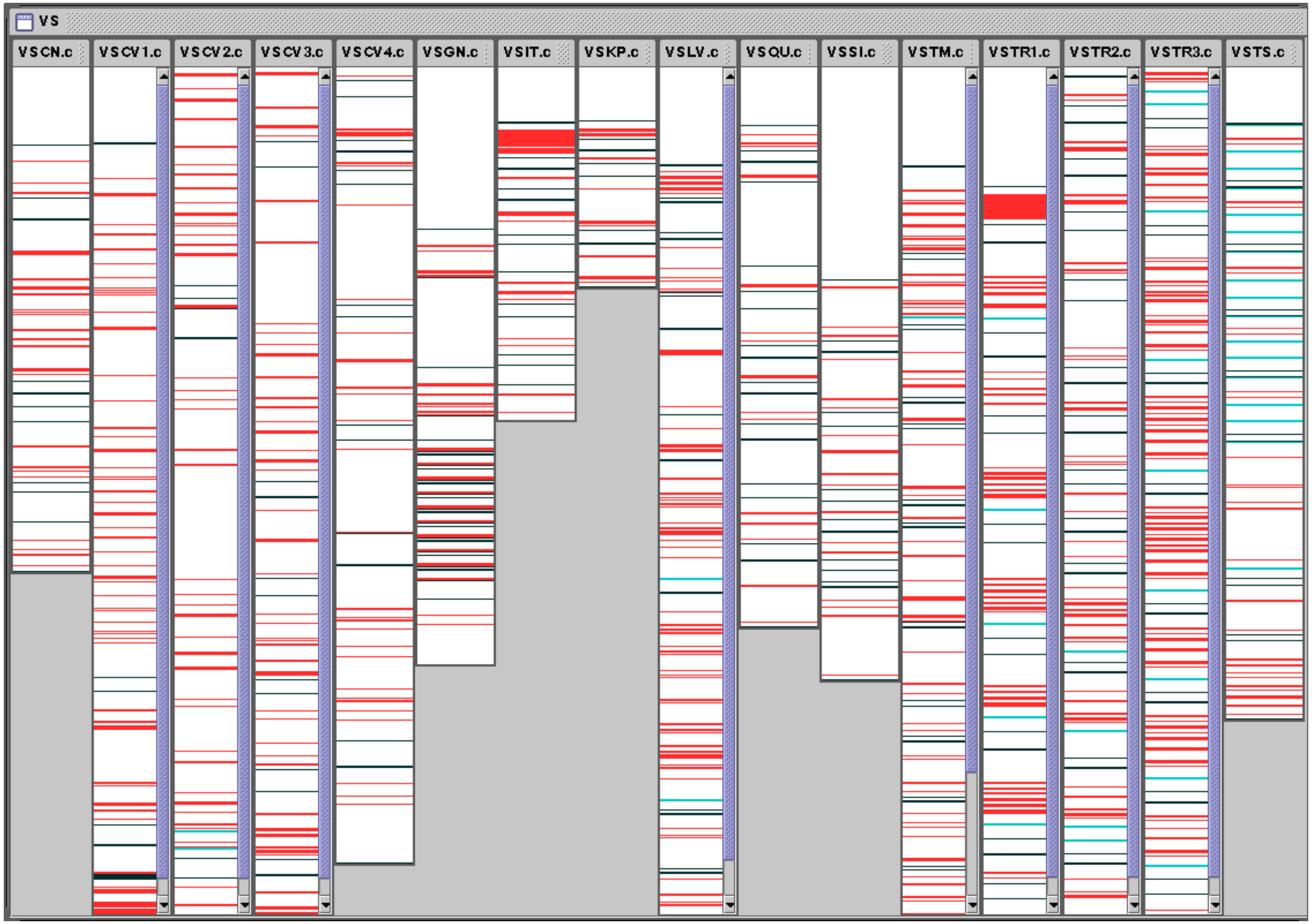
in an Industrial Case:

- **ASML lithography systems: waferscanners**
  - 400 sensors, 300 actuators, 50 processors
- **Software: 12 MLoC (mostly C)**
  - highly optimized for nm precision and high throughput
  - software structured into 6 layers, ~ 200 components
- **4 releases each year**
  - Continuous stream of change- and problem requests
- **Several 'aspects' are reappearing throughout the software**
  - → illustrated by (manual) analysis of 1 module:

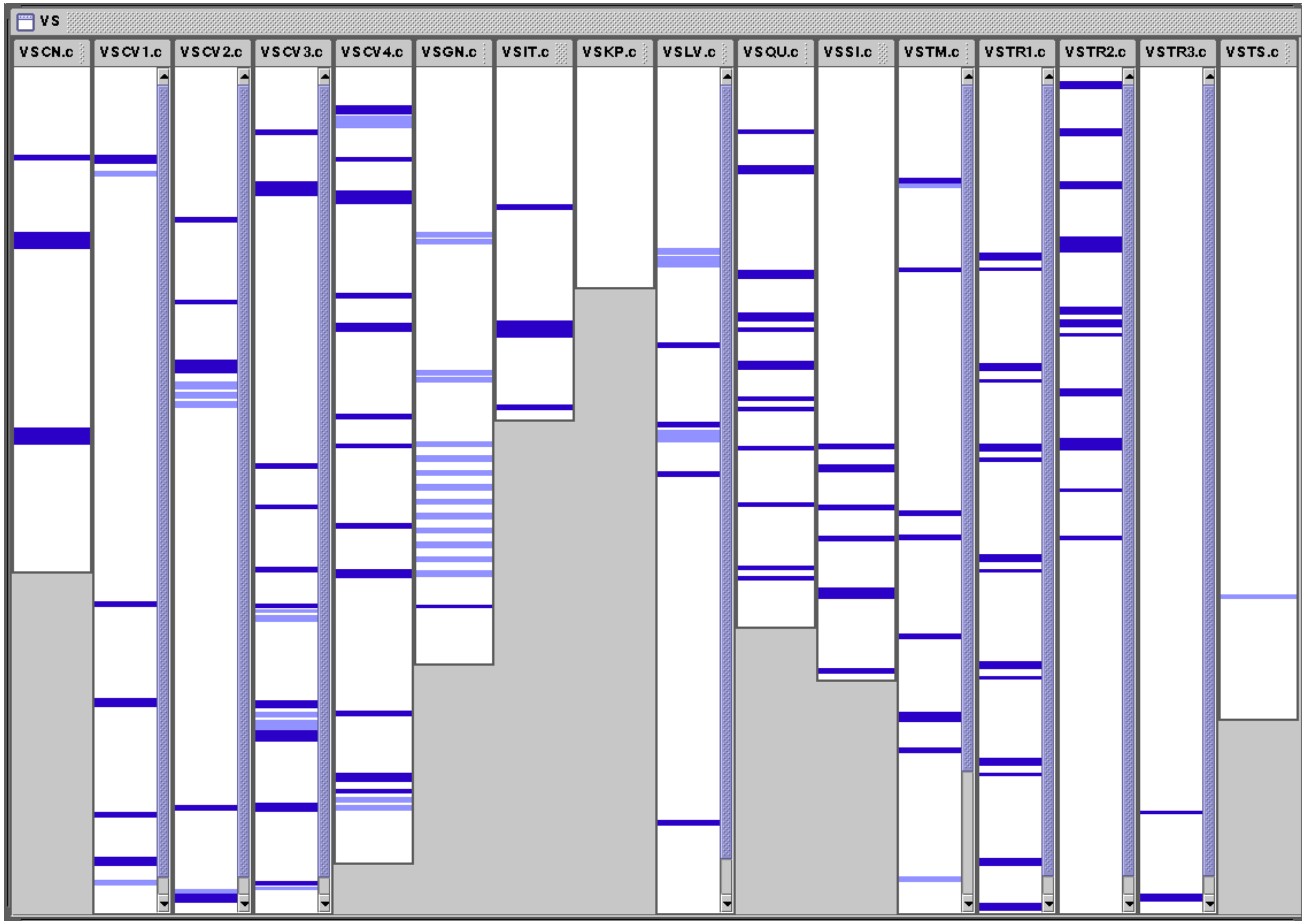
Function value tracing (8% LOC)



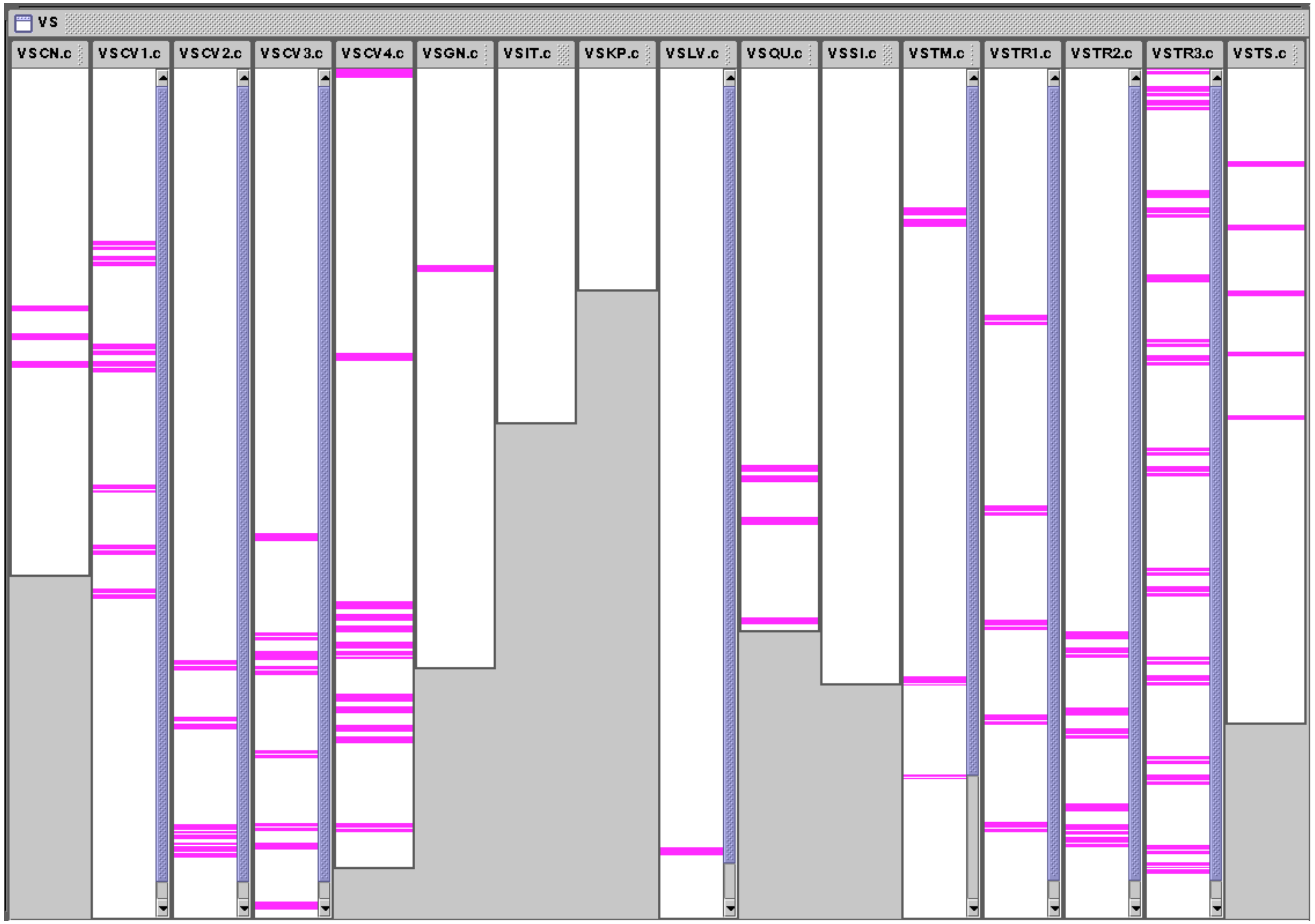
Error Handling (9% LOC)



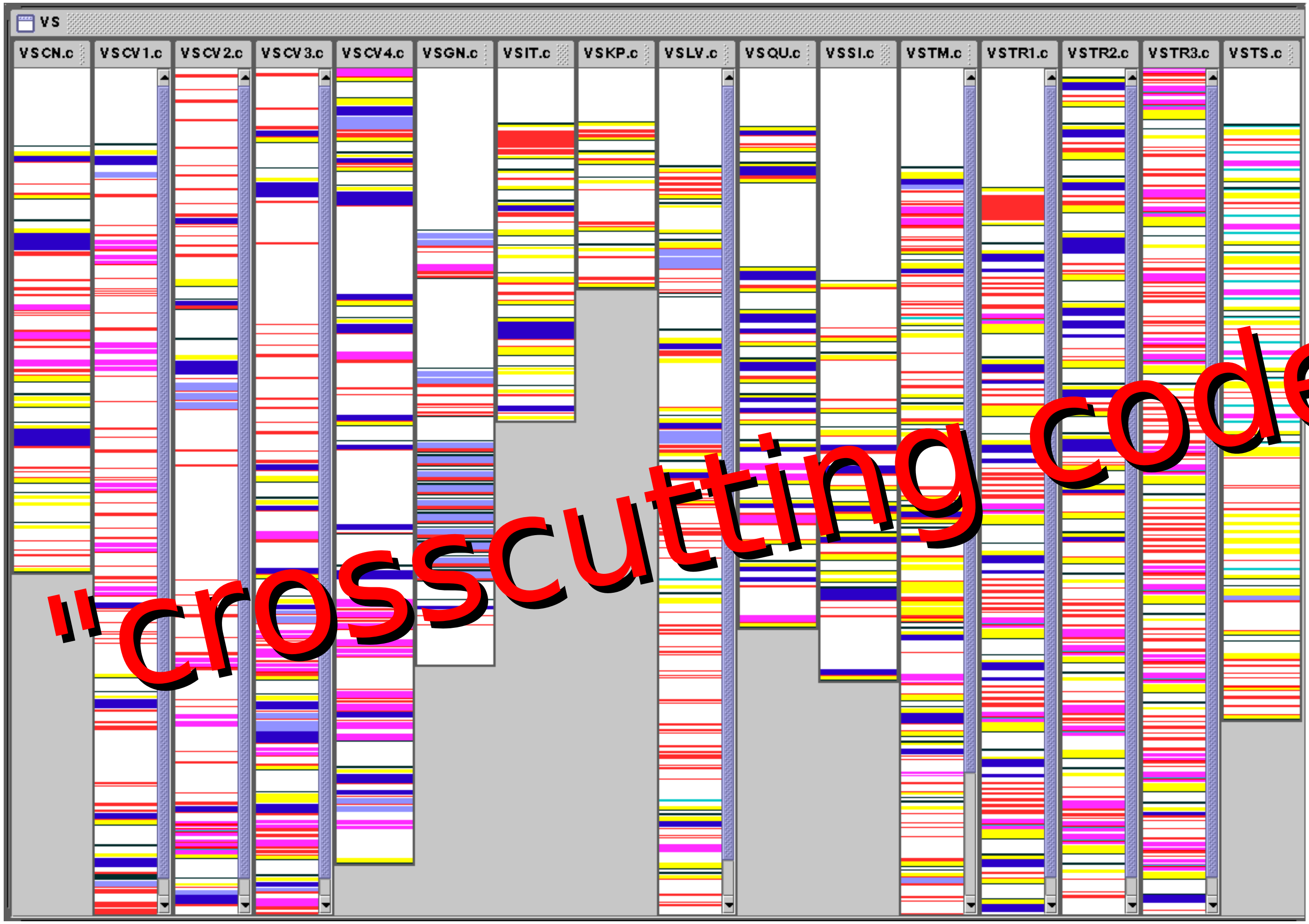
Function parameter checking (7% LOC)



Memory allocation (error) handling (5% LOC)



Total impact of 4 aspects: 29% LOC

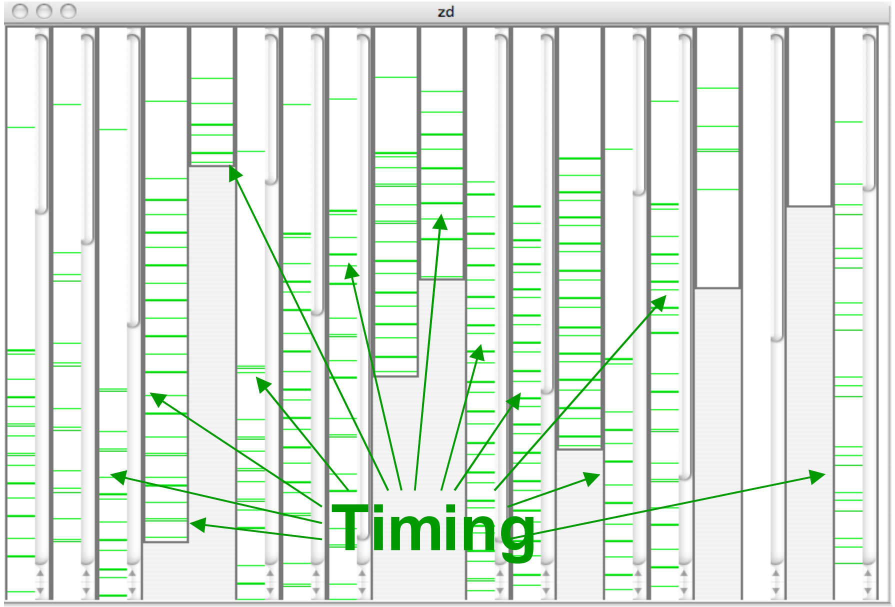


"crosscutting code"

Problems of 'Crosscutting Code'-1

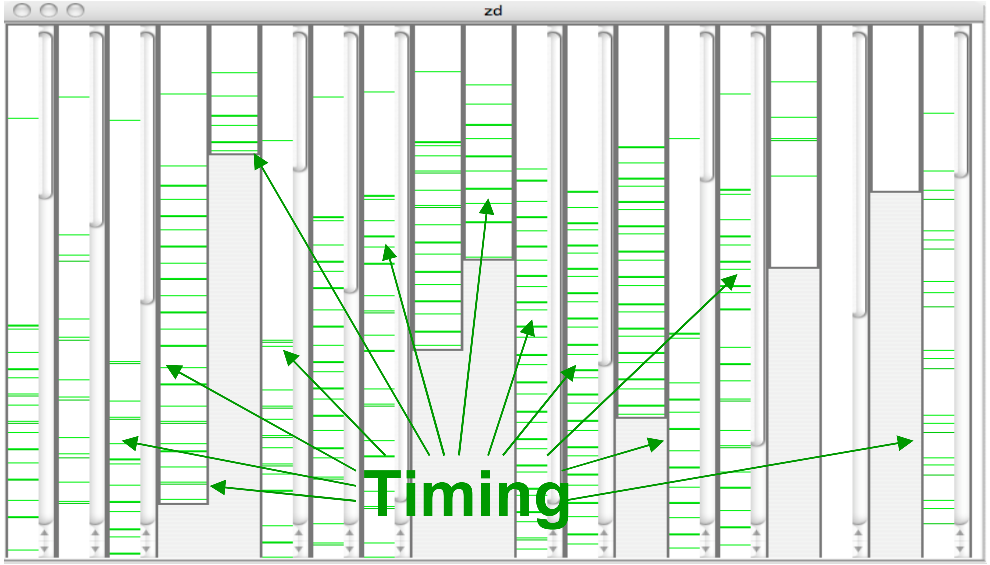
Scattering

- One logical functionality ('concern') is distributed over multiple locations
  - → **hard to maintain** (what if the program is extended, or the cross-cutting concerns evolve)
  - → **hard to keep overview** (where is a concern implemented)



Scattering cnt'd: Replicated Functionality

- ❑ Due to scattered implementation of one function
  - ❑ Often by 'copy-past-edit' process



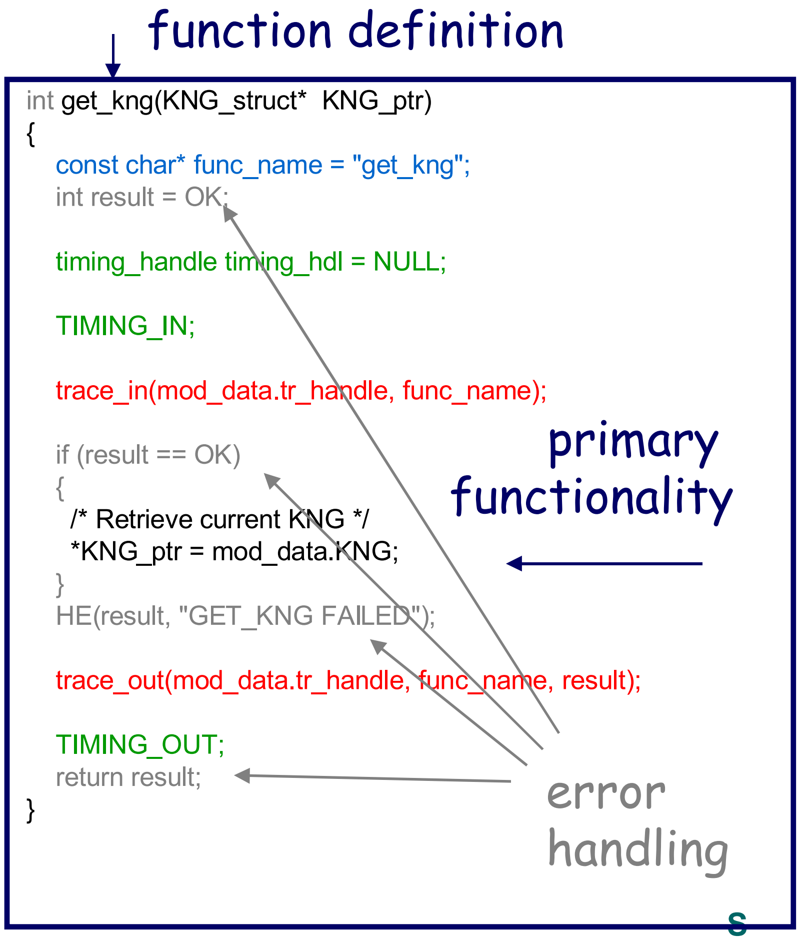
Problems:

- ❑ a lot of effort involved
- ❑ still a lot of errors occur in 'boilerplate code'!
- ❑ change of functionality requires many updates
- ❑ often the same functionality is designed and implemented inconsistently in distinct subsystems!

Problems of 'Crosscutting Code'-2

Tangling

- If a program unit contains mixture/interleaving of concerns
- Example function:
  - get data element from struct
  - colored per concern
  - deals with many concerns
- Problems:
  - comprehensibility
  - maintainability: updating one concern may break surrounding code



```
int get_kng(KNG_struct*  KNG_ptr)
{
   const char* func_name = "get_kng";
   int result = OK;

   timing_handle timing_hdl = NULL;

   TIMING_IN;

   trace_in(mod_data.tr_handle, func_name);

   if (result == OK)
   {
     /* Retrieve current KNG */
     *KNG_ptr = mod_data.KNG;
   }
   HE(result, "GET_KNG FAILED");

   trace_out(mod_data.tr_handle, func_name, result);

   TIMING_OUT;
   return result;
}
```

Separation of Concerns

Separation of Concerns

- ***Concept* [Merriam-Webster] :**

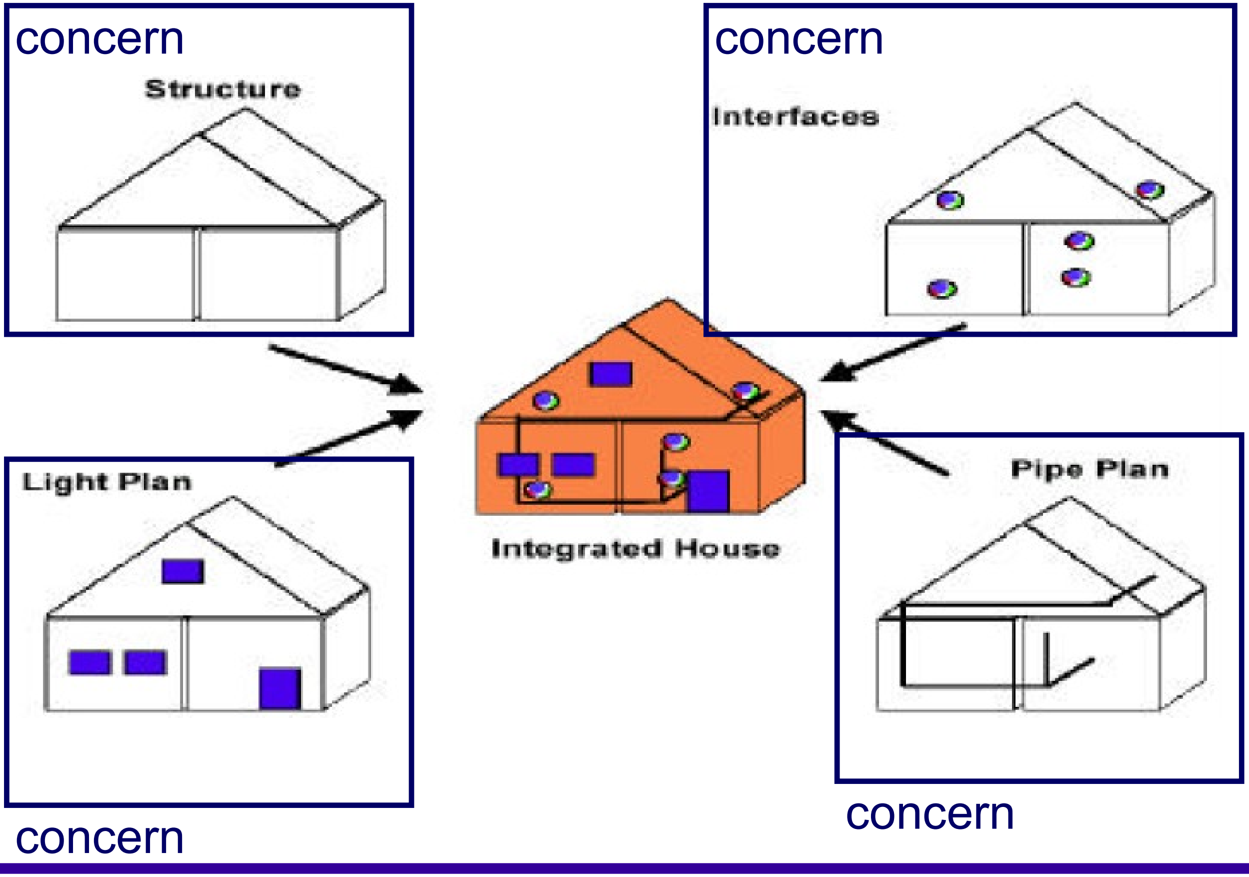
  1: something conceived in the mind: thought, notion.

  2: an abstract or generic idea generalized from particular instances

- ***Concern***: "*concept* that is relevant in the development of the system under consideration"
  - E.g. functionality, requirement, domain concept, ..
  - Can vary from top-level conceptual entity to bottom-(implementation-) level entity
    - **depends on how far you decompose**

Separation of Concerns

- **Separation of concerns**: handle (model) each concern separately
  - At design/programming level: represent in separate module



concern

concern

concern

concern

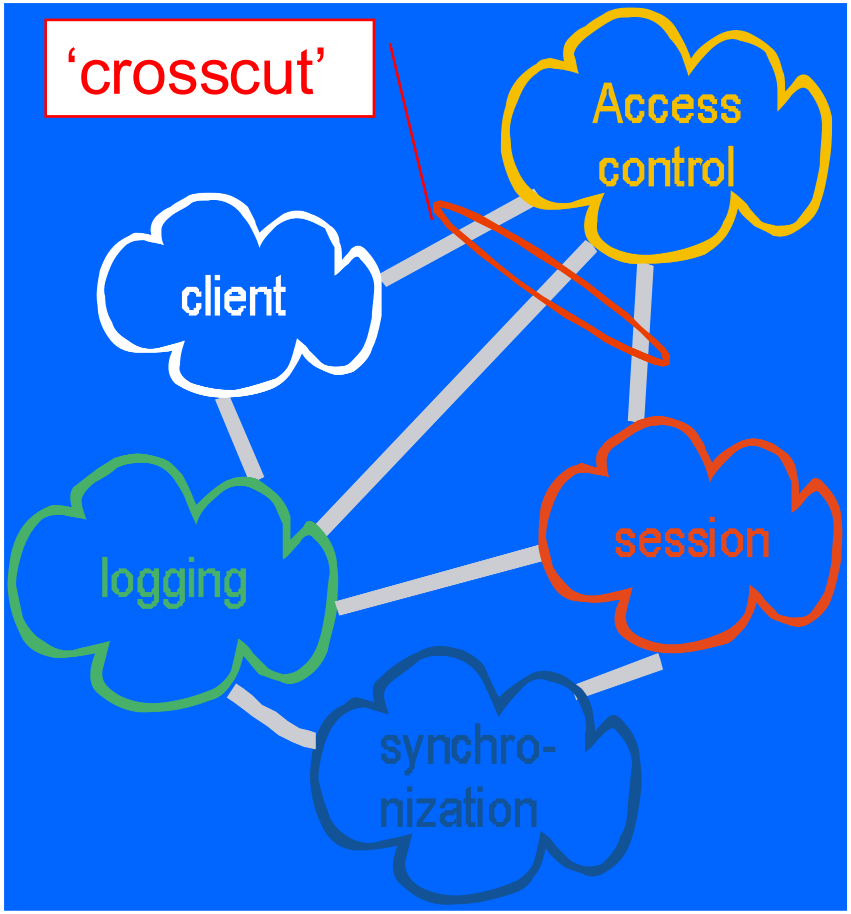
Separation of Concerns

- These concepts & terminology are 'ancient':
  - **"Separation of Concerns"**
    - Structured Programming
    - E.W. Dijkstra, "On the Role of Scientific Thought", 1974
  - **"Modular Design"**
    - Modular Programming
    - D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", 1972

Problem at conceptual/design level:

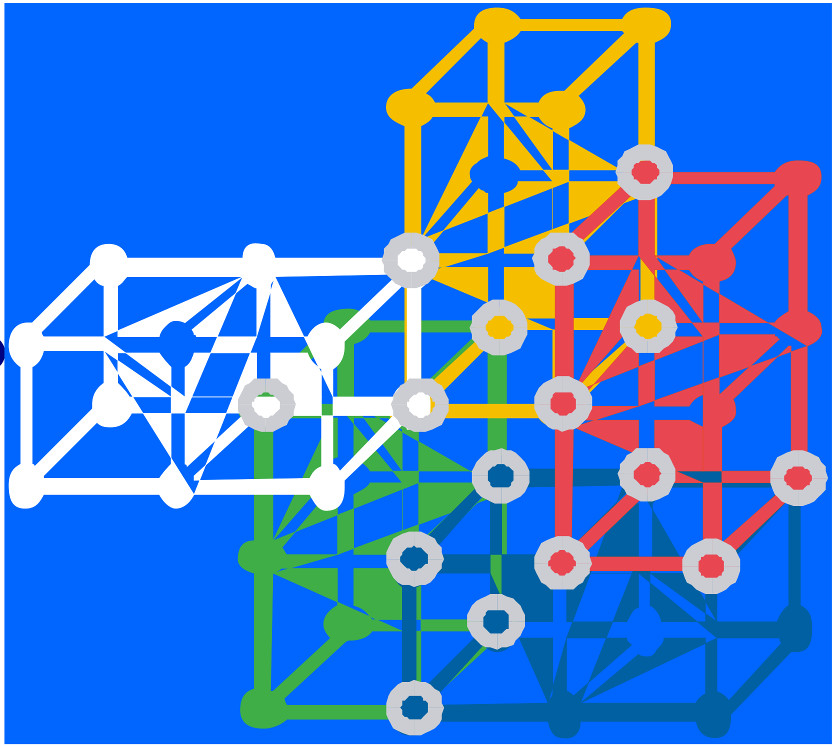
separation of concerns-1

- (domain) analysis:
  - conceptual concerns are identified & separated



Separation of concerns-2

- detailed design & implementation:
  - some concerns are so tied together they need to be implemented –at least partly– in the same module, e.g. same class



Goal of AOP:
implement concerns in separate modules anyhow

Part II: General Approach:

Aspect Orientation

Crosscutting Concerns

- **A concern that 'interacts' with multiple other concerns:**
  - **In many systems, some issues (concern/‘aspects’) crosscut parts of the system (other concerns);**
  - **When realizing such a concern (without AOP), this yields:**
    - 'scattering': one concern is implemented by multiple modules
    - 'tangling': multiple concerns *inseparably* mixed in one module
  - **'crosscutting' is considered as a relative concept**
    - if its’ refinement at the implementation level is tangled & scattered with refinements of other concerns.
  - **An 'aspect' is a modular implementation of a crosscutting concern**

Examples of Crosscutting Concerns

❑ From application domain

- ❖work flow in administrative systems,
- ❖control loop in control/embedded systems.
- ❖e.g. in an ERP system: tax, insurance, laws, ..

❑ From realization domain

- logging & tracing code
- security code
- synchronization
- recovery code
- memory management
- distribution
- persistence & serialization
- ...

Two categories of crosscutting

1. **crosscutting replicated behavior**
   - e.g. logging, synchronization, etc.
   - one piece of behavior re-appearing in many places
2. **single behavior spread over multiple modules**
   - e.g. page prefetching [Coady03], use cases, initialization, layered systems
   - single logical module crosscuts several modules

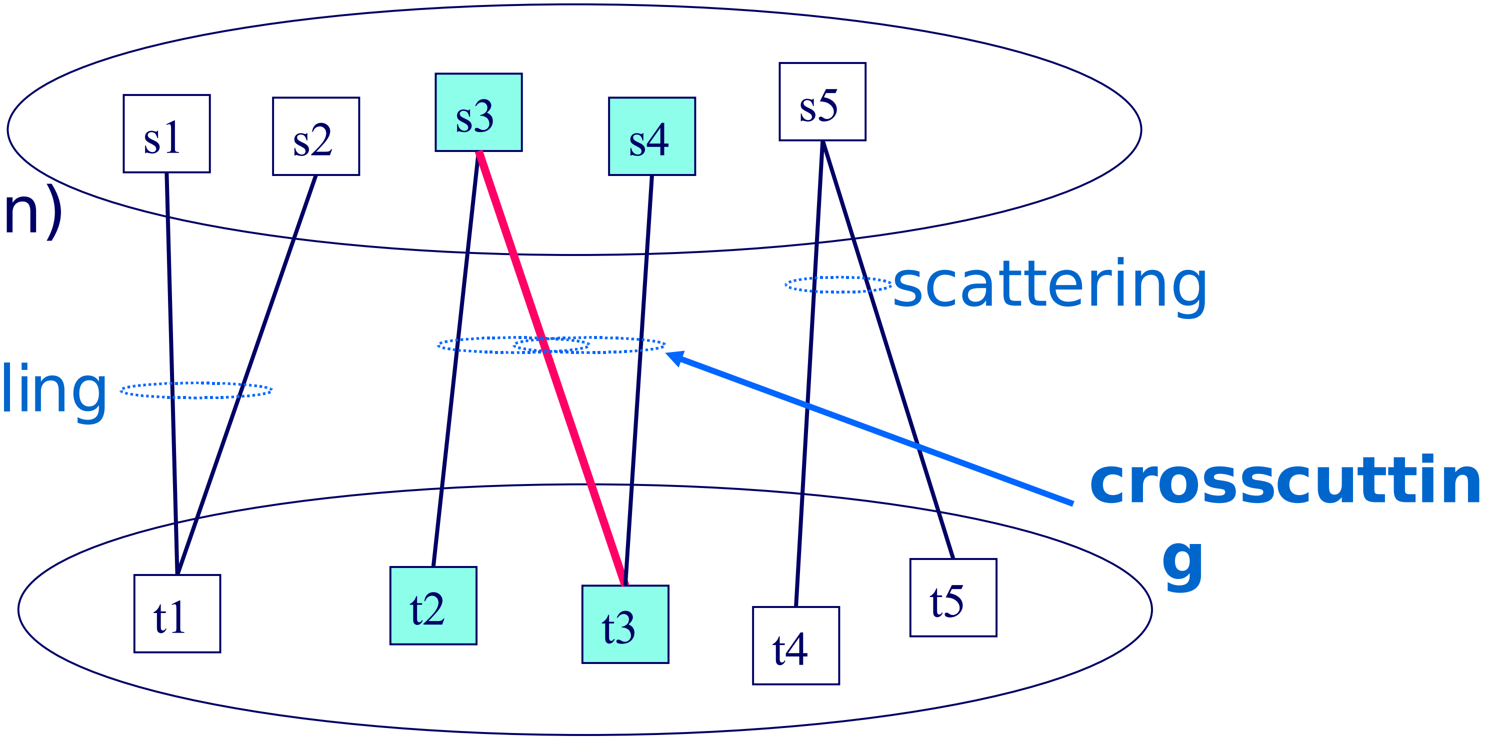
The above cases may be combined

- e.g. persistence

Crosscutting

Elements involved both in scattering and tangling

s3 crosscuts s4



Source

(e.g. design)

s1

s2

s3

s4

s5

scattering

tangling

crosscuttin g

t1

t2

t3

t4

t5

Target

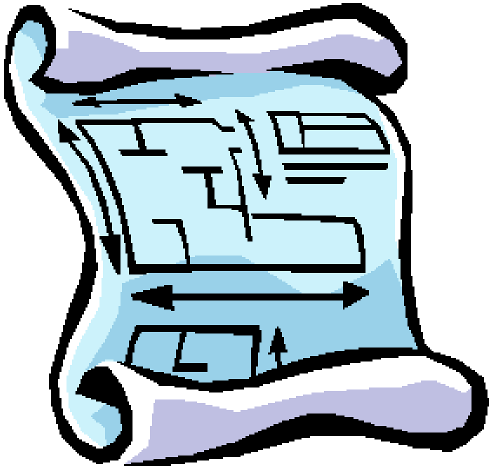
(e.g. implementation)

Crosscutting

- **Crosscutting**
  - a source element is scattered over target elements and where in at least one of these target elements, multiple source elements are tangled
- **s1 *crosscuts* s2**
  - for a given mapping between source and target (i.e. for a given implementation)
  - if s1 is scattered over target elements and in at least one of these target elements, s1 is tangled with source element s2
  - s1 *crosscuts* s2 with respect to a mapping to and decomposition of the target

Addressing Software Engineering Complexity with (Language) Abstractions

Software Engineering Complexity



Functional Requirements

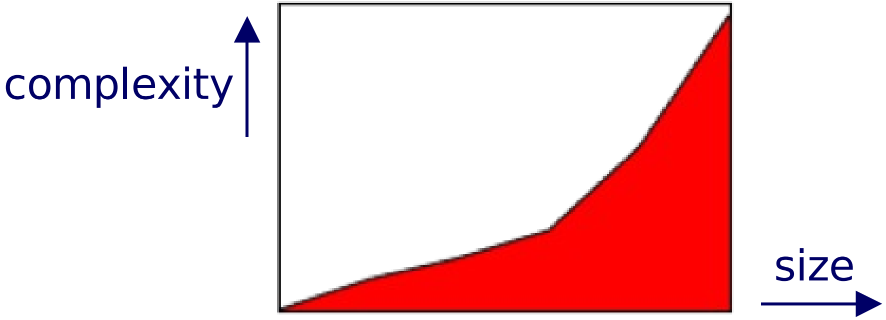
+

Quality (non-functional) Requirements

+

Software Development Requirements

=> Complexity:

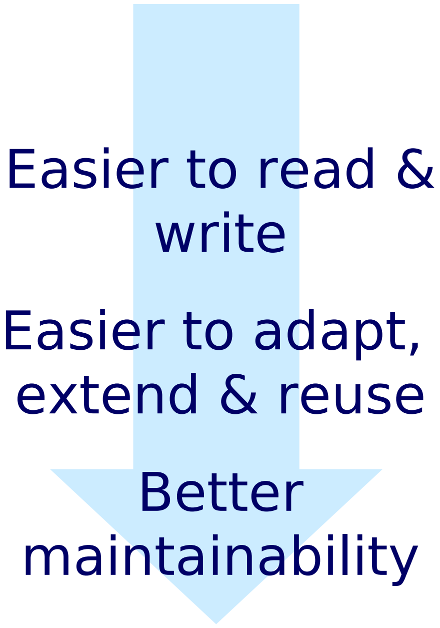


Need for adequate software engineering techniques

Programming Language Abstractions

Evolve:

- Machine/assembly programming
- Structured programming
- Procedural programming
- Modular programming
- Object-oriented programming



Easier to read & write

Easier to adapt, extend & reuse

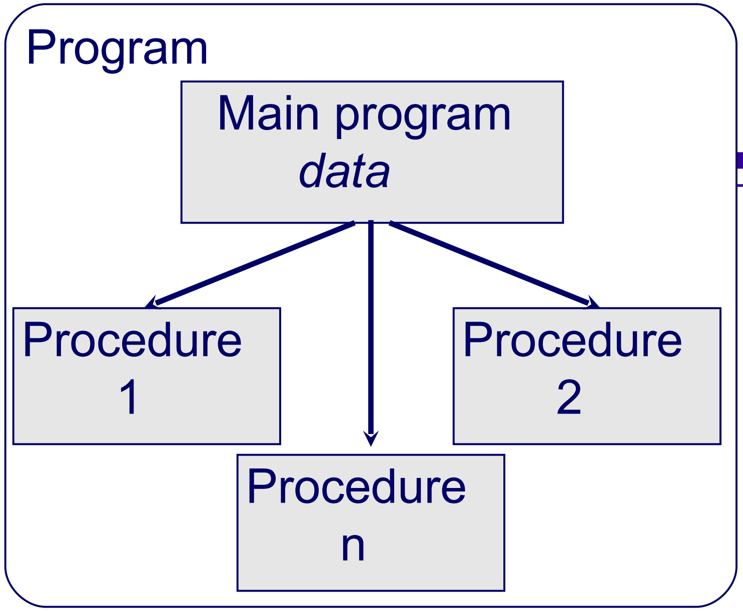
Better maintainability

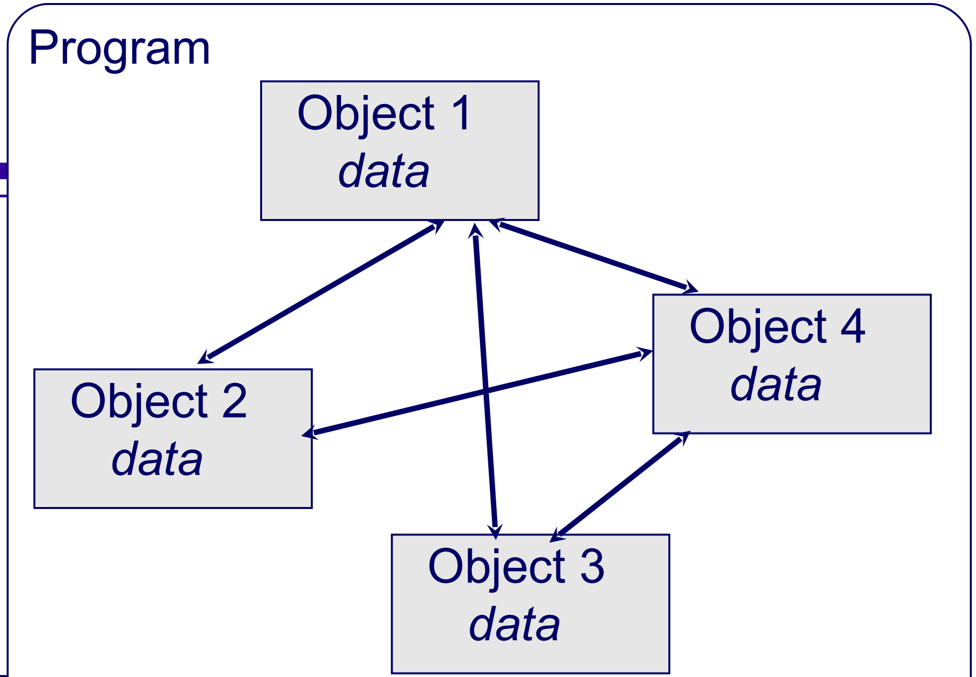
❑ Why?

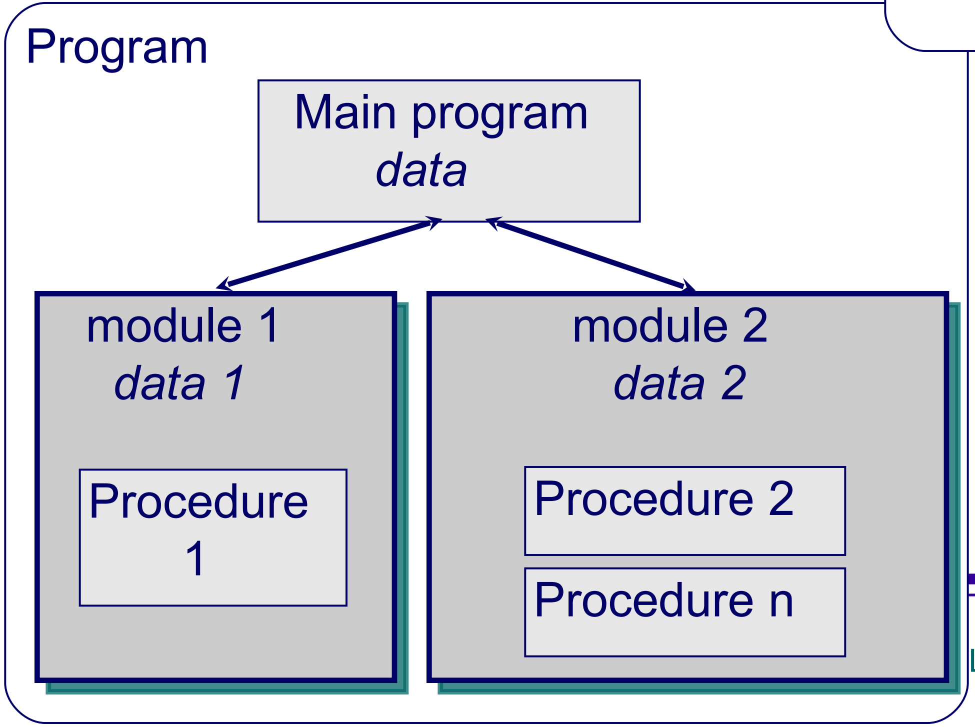
- Improved separation & modularization of concerns
- → Aspect Abstractions to deal with crosscutting concerns

Programming with Aspect Abstractions

- **Two fundamental issues must be addressed:**
  - **how to spread/distribute concerns (code)**
    - *Join Point Model* (JPM):
      - **what are the join points**
      - **means of identifying join points**
      - **means of specifying semantics at join points**
  - **how to compose ('superimpose') concerns (code):**
    - implicit invocation instead of explicit invocation model
    - Requires an additional composition mechanism
    - new (different?) abstractions needed

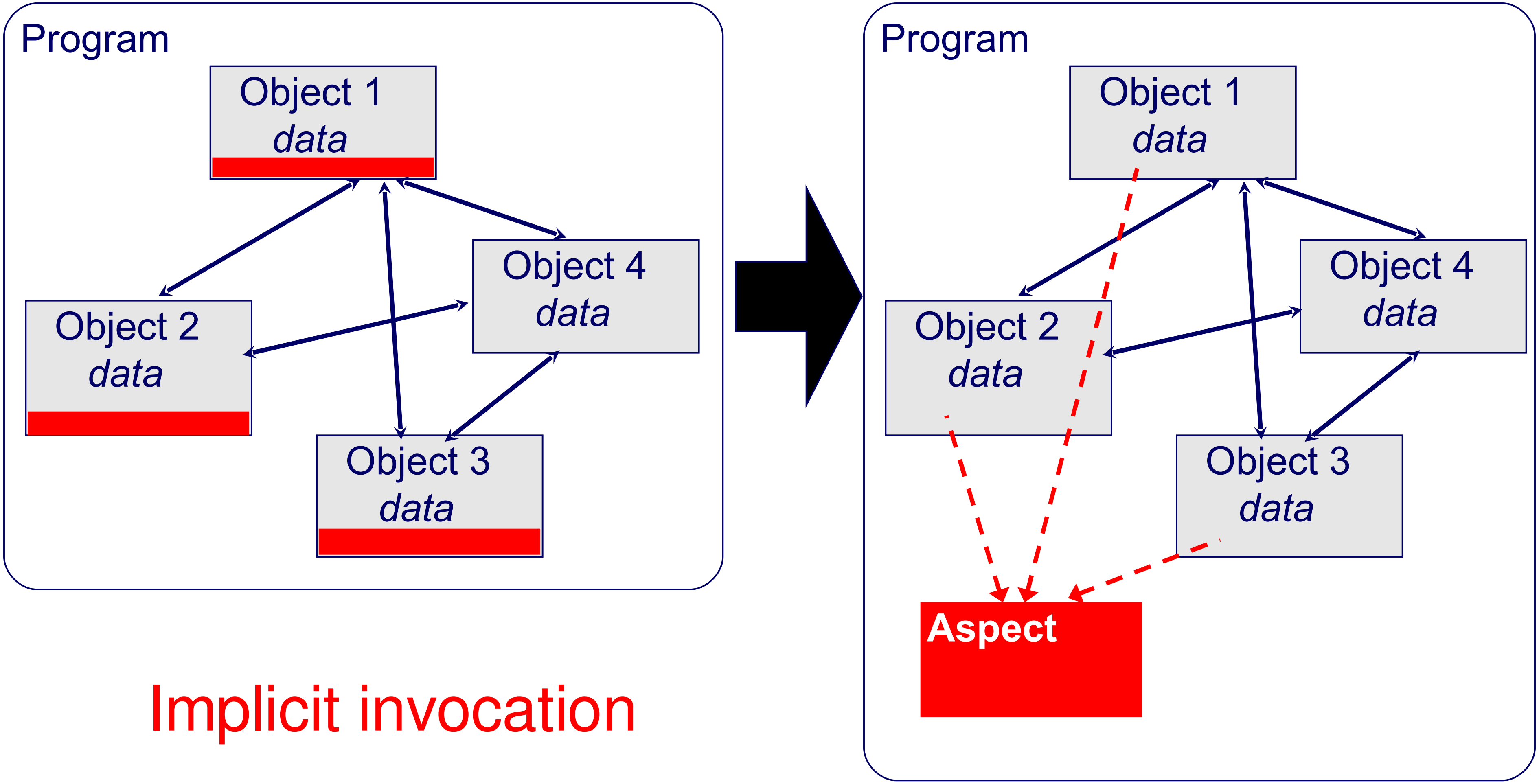






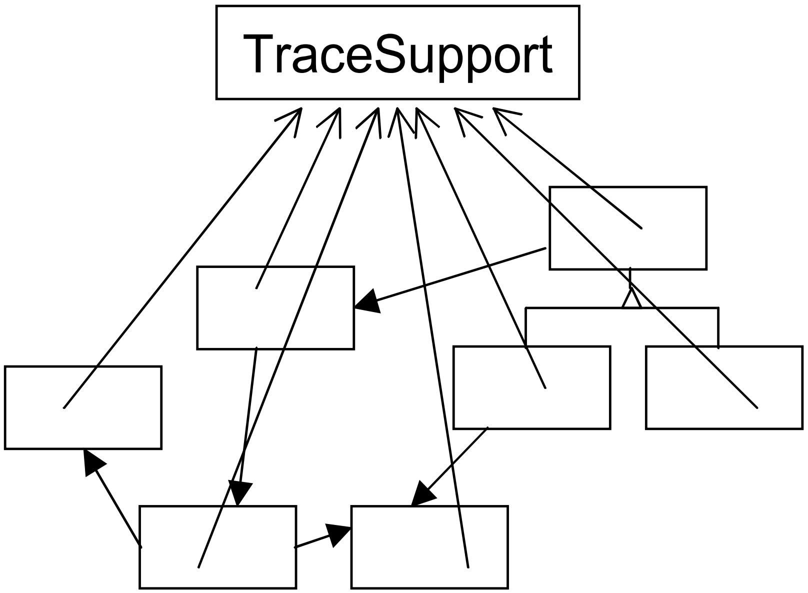
Explicit invocation

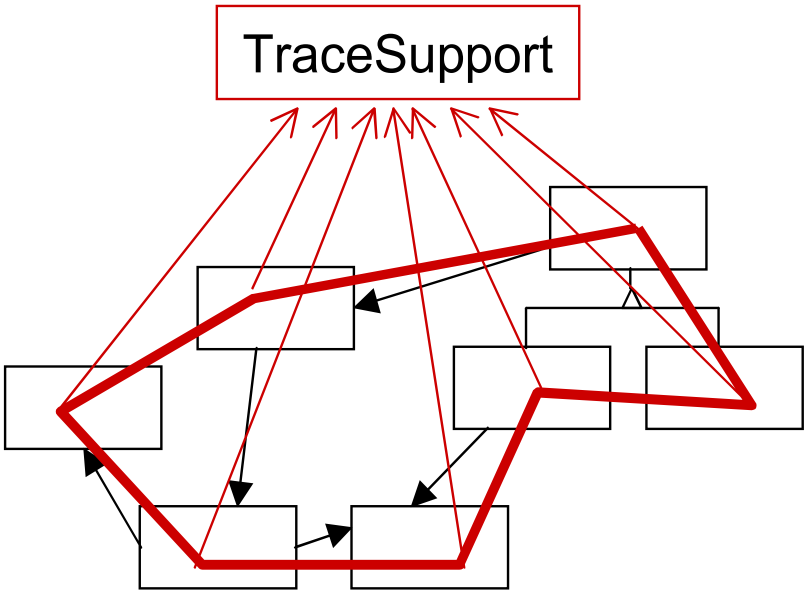
Aspects



Implicit invocation

Implicit Invocation

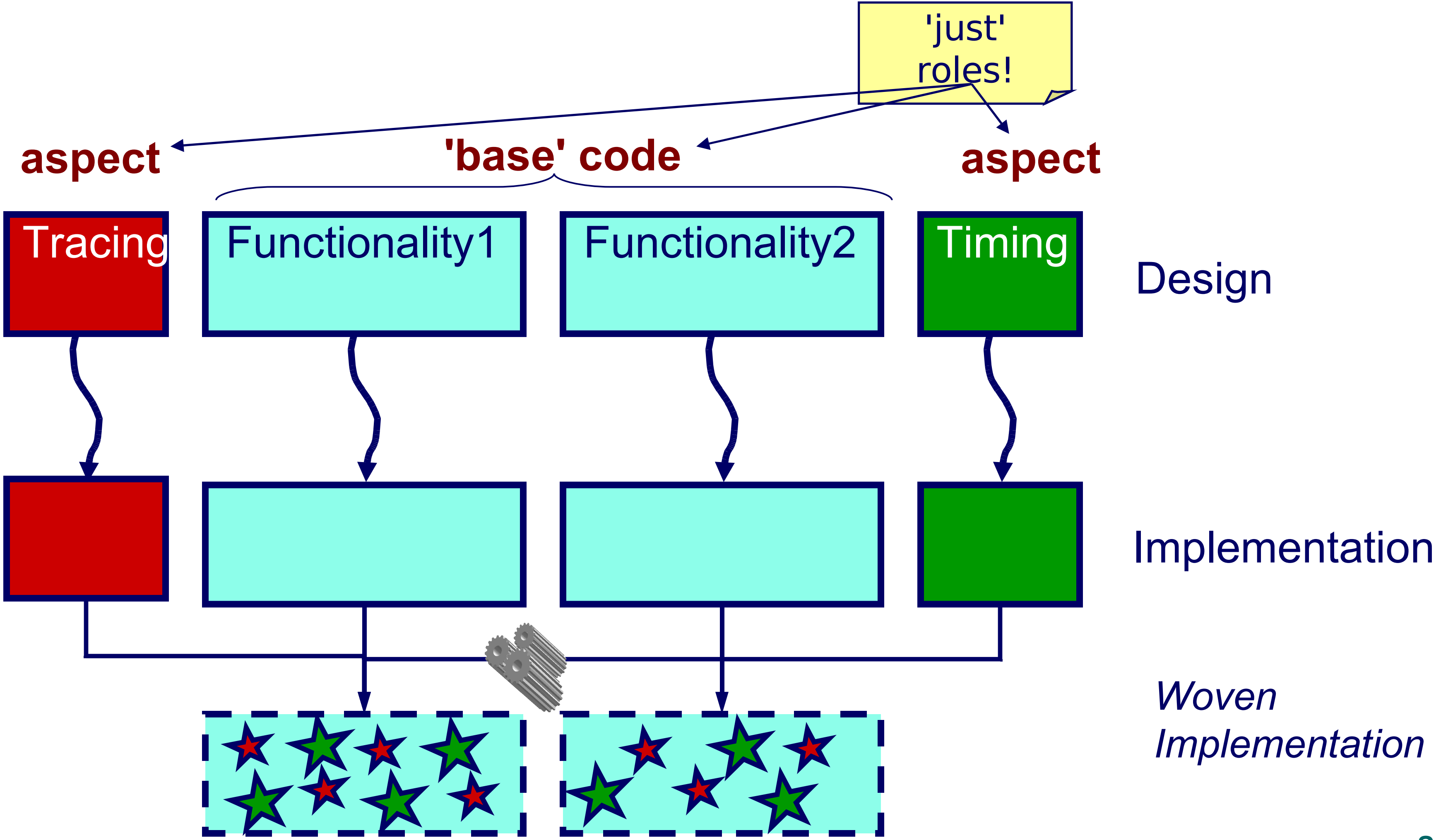




Objects are invoked by other objects through message sends

Aspect captures its own invocations that crosscut other modules

Modular Development with Aspects



'just' roles!

aspect

'base' code

aspect

Tracing

Functionality1

Functionality2

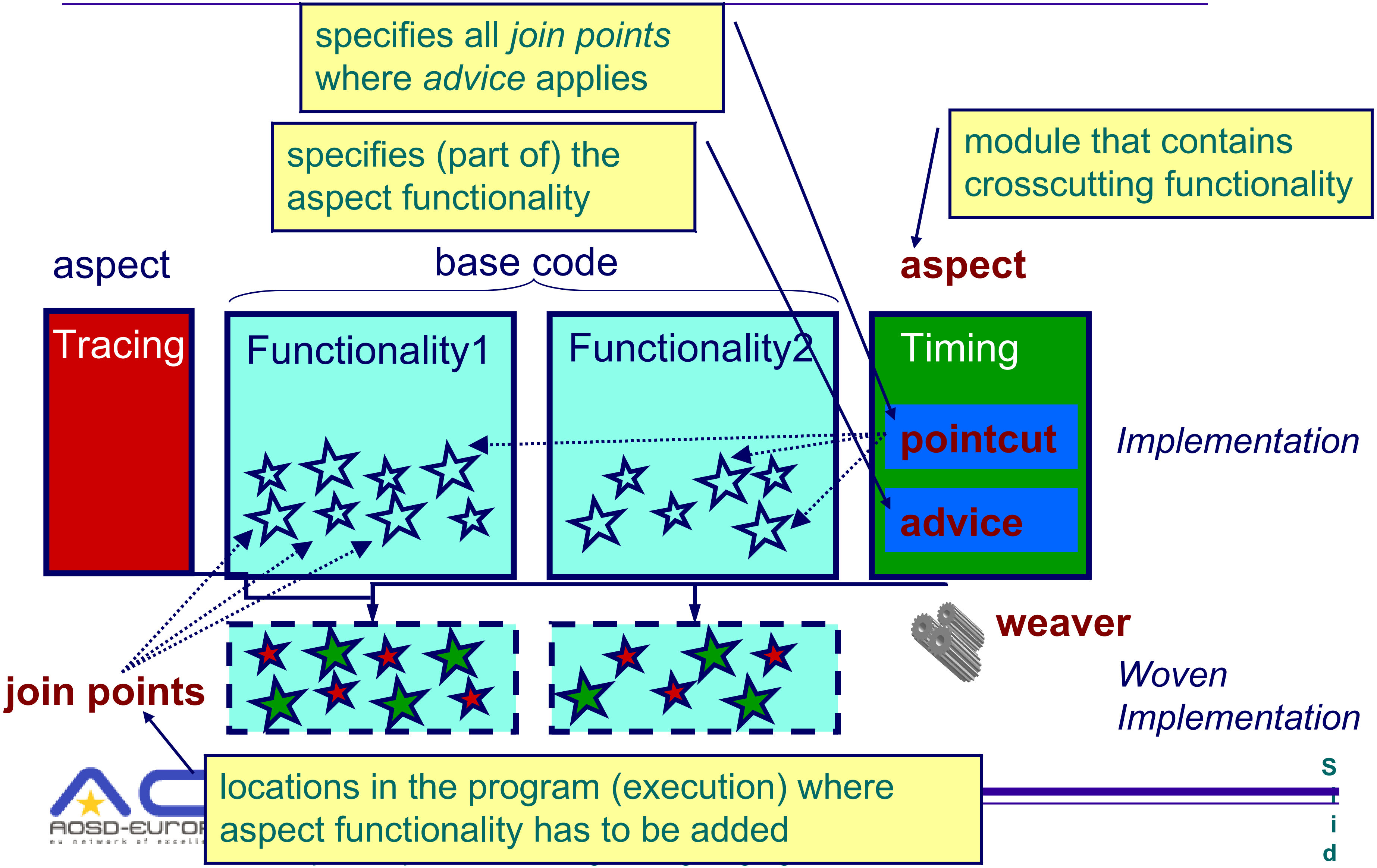
Timing

Design

Implementation

Woven Implementation

Aspect-Oriented Programming



specifies all *join points* where *advice* applies

specifies (part of) the aspect functionality

module that contains crosscutting functionality

locations in the program (execution) where aspect functionality has to be added

Key Concepts of AOP

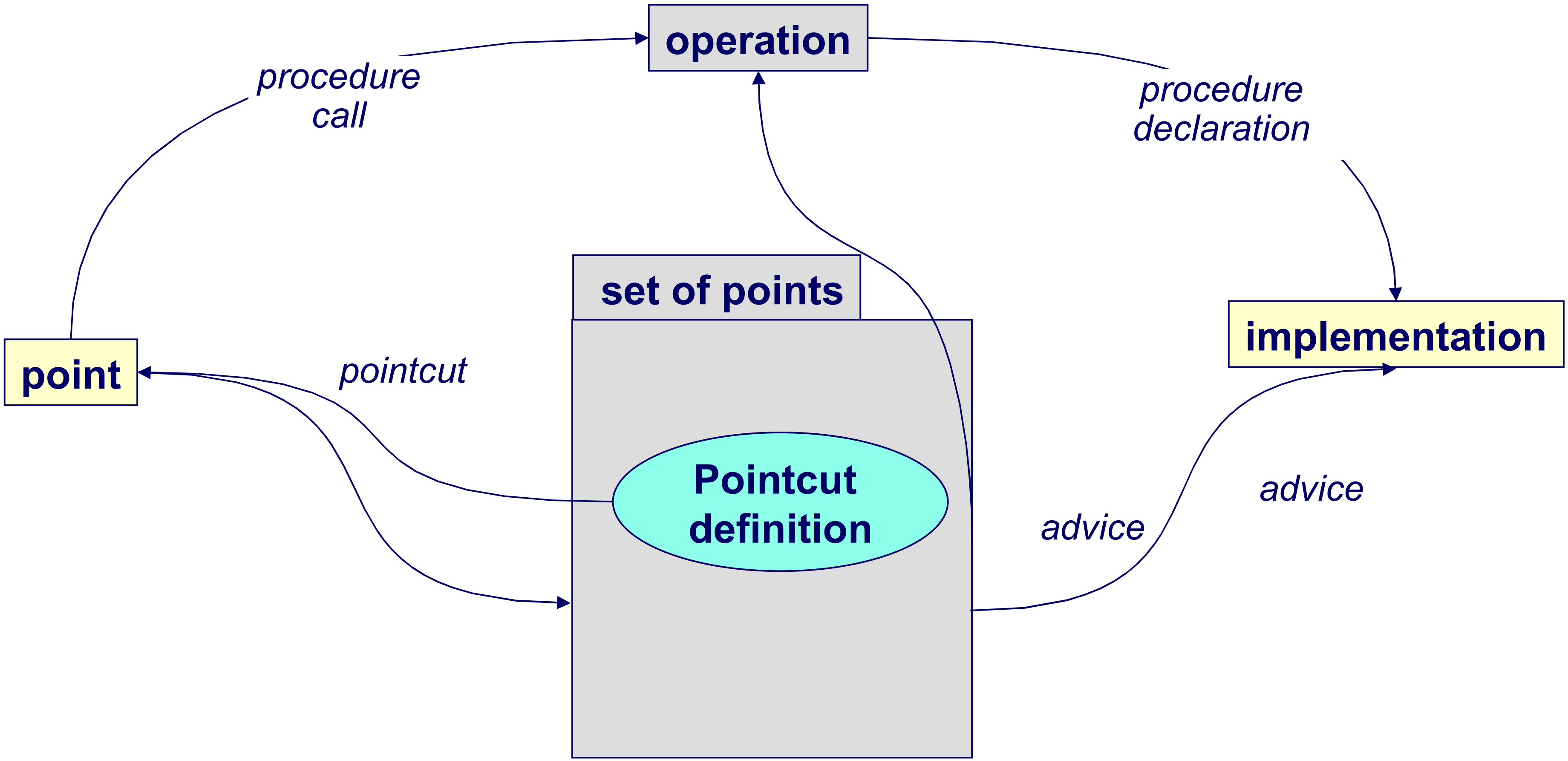
- **Join Point Model**
  - **Points where behaviour can be invoked implicitly**
  - **E.g. points in the control flow of a program**
- **Pointcut Language**
  - **Means of identifying join points**
  - **E.g. logic predicate language**
- **Aspect Behavior**
  - **Means of effecting semantics at identified join points**
    - Behavioral
    - Structural
  - **E.g. before/after/around advices**

Uniform Characterization

- **Binding mechanisms**
  - **(Static) function call**
    - E.g. C functions
  - **(Dynamic) method lookup**
    - E.g. polymorphic method in Java, Smalltalk, etc..

The JPMs + Pointcuts mechanism can be seen as an extension and generalization of bindings

Uniform Characterization



Part III: A concrete AOPL:

AspectJ

(includes –modified- material

from the AspectJ tutorial)

AspectJ Extends Java with Aspects

Some design criteria/goals:

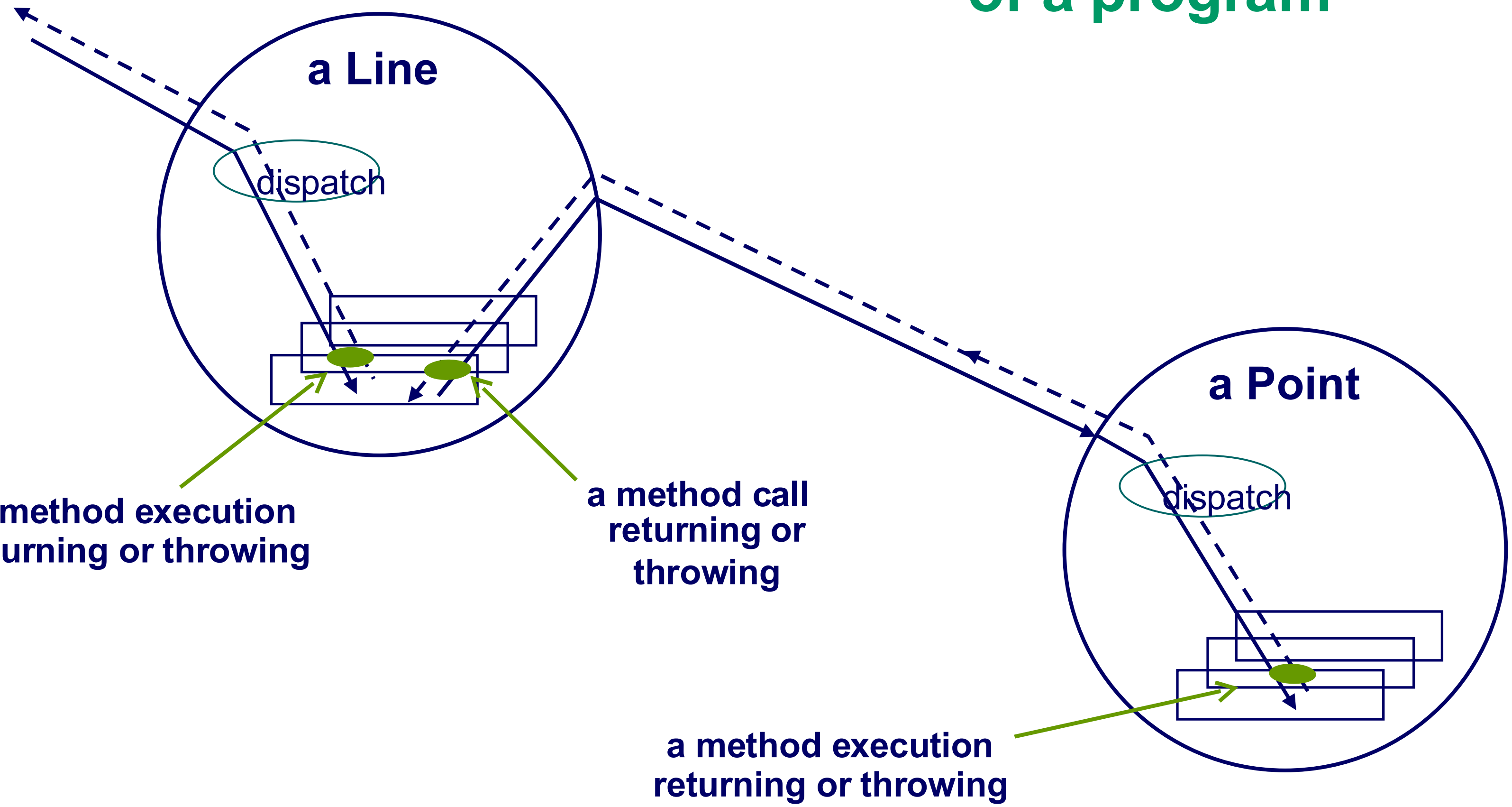
- **compatible extension to Java:**
  - *upward* compatibility (Java program => AspectJ)
  - *platform* compatibility (use regular JVM)
  - attempt to make a small addition to Java (...)
- **general-purpose rather than domain-specific**
- **balance of declarative & imperative constructs**
- **statically typed, uses Java’s static type system.**

Main Concepts

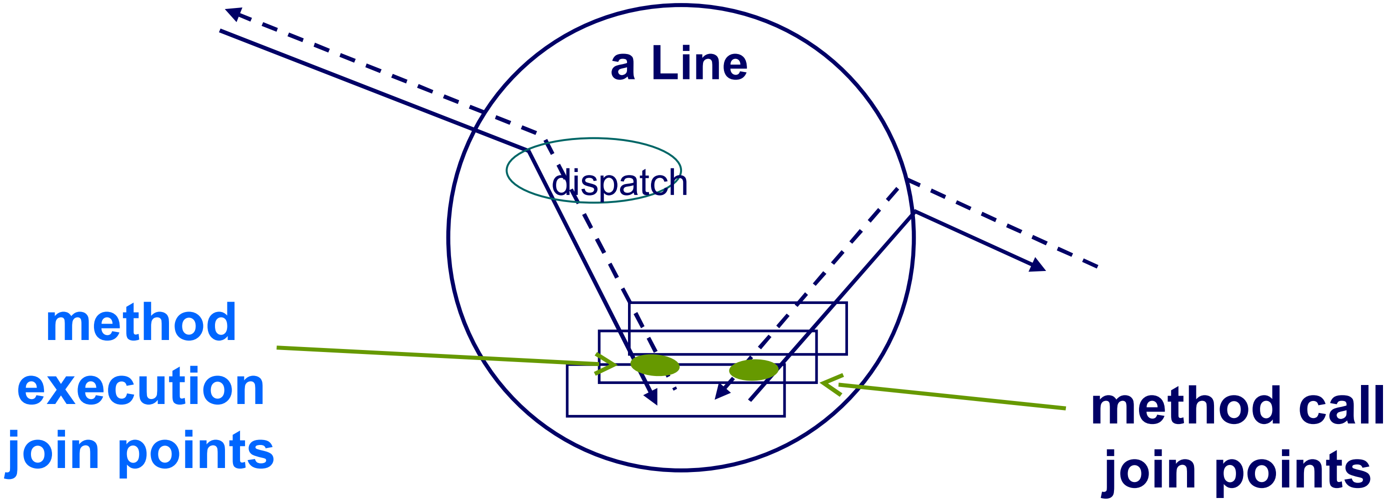
- **Join Points**
  - "points in the execution of a Java program"
- **Pointcuts**
  - Selection of set of join points and values at those points
- **Advice**
  - code that can be 'added' to join points
- **Aspects**
  - modules of crosscutting implementation
    - □ **= pointcuts + advice + Java member declarations**

Join points

points in the structure, or the execution, of a program


a Line
dispatch
a Point
dispatch
a method execution returning or throwing
a method call returning or throwing
a method execution returning or throwing

Join point terminology-1



method execution join points

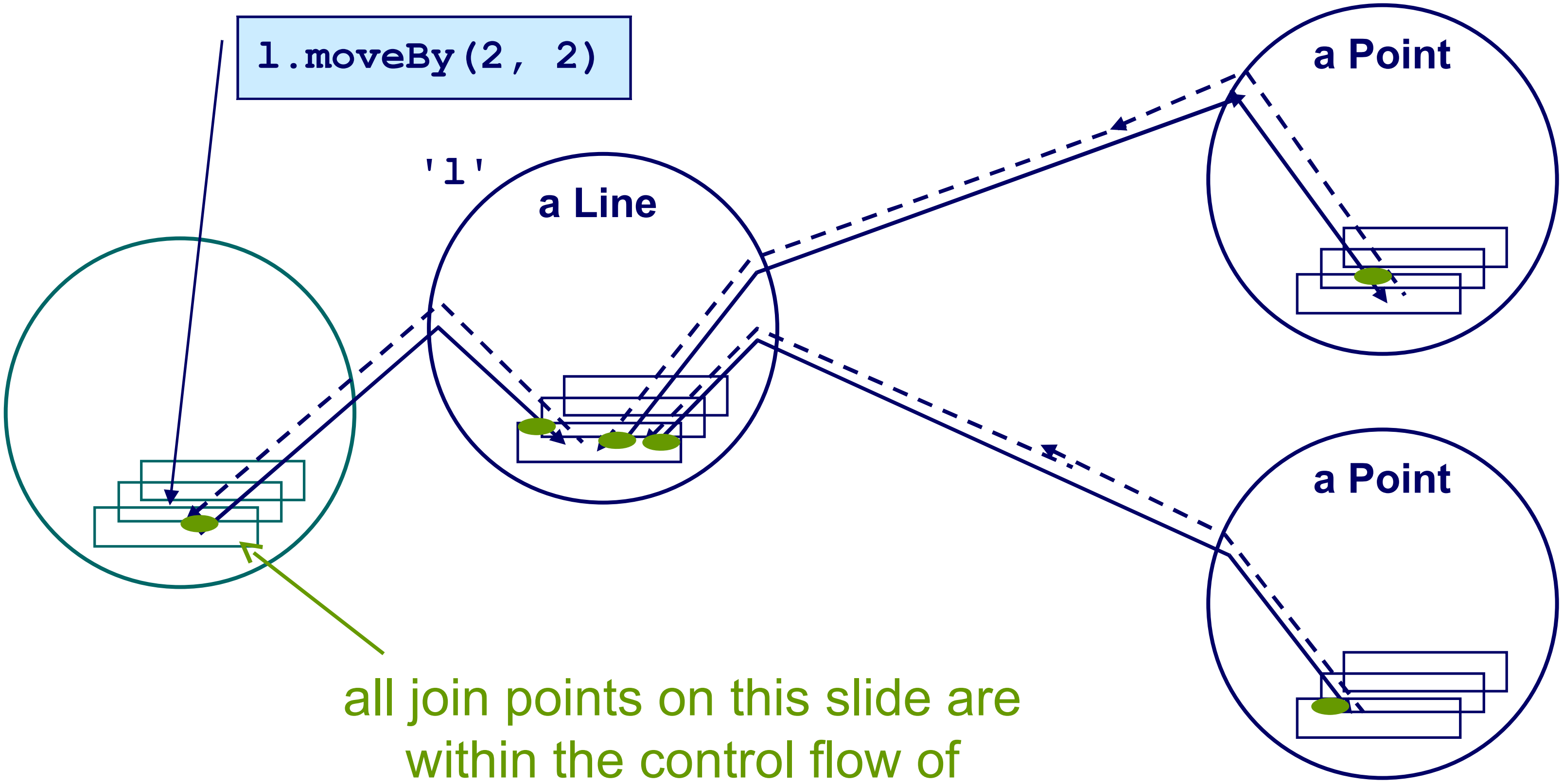
method call join points

❑ **AspectJ has**

- method *execution* join points
- method *call* join points
- constructor call & constructor execution join points
- field access join points
- exception handler execution join points

Join point terminology-2

key points in dynamic call graph



all join points on this slide are within the control flow of this join point

The pointcut construct

- **Names certain (set of) join points**
  - it is an abstraction of the 'crosscut', which can be referred to and reused.
  - specified by 'pointcut designators'
    - □ **declarative matching specification**
    - □ **predicates that can be composed with AND & OR, select a subset of the joinpoints.**
    - □ **A number of primitive pointcuts have been defined, for which there are predefined designators..**

Some predefined (primitive) pointcuts

- □ **when a particular method body executes**

```
execution(void Point.setX(int))
```

- □ **when a method is called**

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

- □ **when an exception handler executes**

```
handler(ArrayOutOfBoundsException)
```

- □ **when the object currently executing (this) is of type *SomeType***

```
this(SomeType)
```

- □ **when the target object is of type *SomeType***

```
target(SomeType)
```

- □ **when the executing code belongs to class *MyClass***

```
within(MyClass)
```

- □ **all join points in the dynamic control flow of any joinpoint specified by *Pointcut***

```
cflow(Pointcut)
```

User-defined pointcuts

example (user-defined PC Designator)

name and parameters

a “void <a Line>.setP1(<a Point>)” call

or

```
pointcut moves():
  call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point));
```

a “void <a Line>.setP2(<a Point>)” call

each time a Line receives "void setP1(<a Point>)” or “void setP2(<a Point>)” method calls

'Advice'

- Action to be taken at a certain join point
  - specified as a Java method body (arbitrarily complex)
- Can be attached to join points:

before	before proceeding at join point
after returning	a value to join point
after throwing	a throwable to join point
after	returning to join point either way
around	on arrival at join point: gets explicit control over when & if program proceeds

multi-class aspect

```
aspect MoveTracking {
  private static boolean _flag = false;
  public static boolean testAndClear() {
    boolean result = _flag;
    _flag = false;
    return result;
  }
  pointcut moves():
    execution(void Line.setP1(Point)) ||
    execution (void Line.setP2(Point)) ||
    execution (void Point.setX(int))   ||
    execution (void Point.setY(int));
  static after(): moves() {
    _flag = true;
  }
}
```

aspect defines a special class that can crosscut other classes

multi-class aspect with context

```
aspect MoveTracking {
  private static Set _movees = new HashSet();
  public static Set getMovees() {
    Set result = _movees;
    _movees = new HashSet();
    return result;   }

  pointcut moves ():
     execution(void Line.setP1(Point)) ||
     execution (void Line.setP2(Point)) ||
     execution (void Point.setX(int))   ||
     execution (void Point.setY(int));

  pointcut movesWhat(FigureElement figElt):
     this(figElt) && moves();

  after(FigureElement fe): movesWhat(fe) {
     _movees.add(fe); }
}
```

Advice Composition

Ordering

- **Advice that applies to the same join point is executed sequentially**
- **Precedence rules:**
  - if advice A has precedence over advice B, A can be executed before B.
  - explicit declaration:
    - □"`declare precedence: AspectA, AspectB;`"

Advice Composition 2

- **If advice comes from different aspects:**
  - follow declare precedence rules
  - *subaspects* have precedence over *superaspects*
  - otherwise undefined
- **If advice comes from the same aspect:**
  - the one that appears earlier in the aspect specification has precedence over others
  - further:
    - □ **around1 → before → body →after returning → after throwing → after → around2**

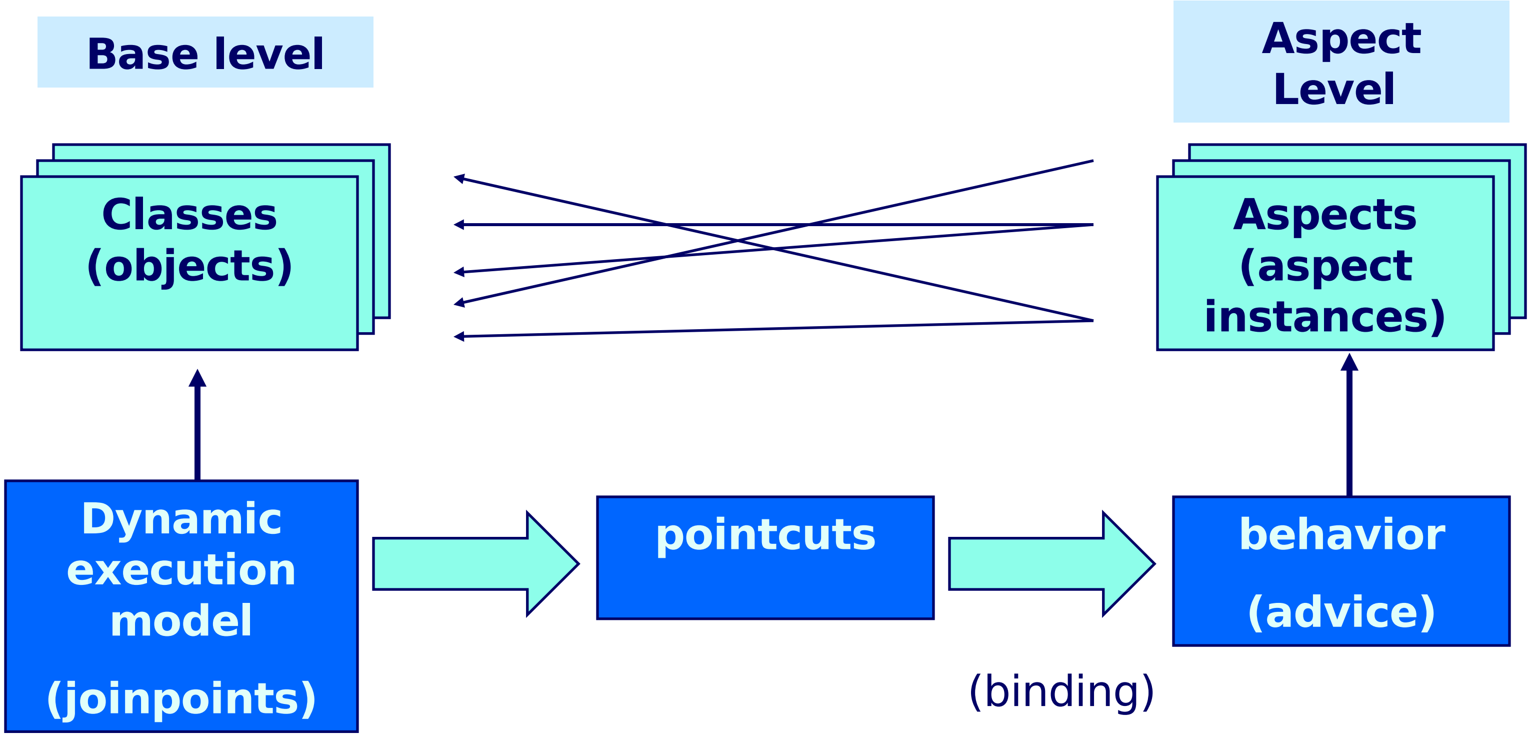
Inter-type declarations

- **Previously called *static join point model***
- **Aspects make declarations that hold for other types (classes):**
  - **inter-type member declarations**
    - **methods & fields**
    - **no full pointcuts can be used**
  - **declare parents**: change inheritance/interface hierarchy
  - **declare error/warning**: when a certain join point is encountered.

Aspect Instantiation

- **advice always executes in the context of an aspect instance**
  - by default, each aspect is a singleton; the same instance used throughout the program
  - aspects may be declared as:
    - □ **perthis(Pointcut): for each executing object**
    - □ **pertarget(Pointcut): for each target object**
    - □ **percflow(Pointcut): for each entrance to a *cflow***
    - □ ***...***

Summary of AspectJ


Base level

Aspect Level

Classes (objects)

Aspects (aspect instances)

Dynamic execution model

(joinpoints)

pointcuts

behavior

(advice)

(binding)

The AspectJ Concepts

- **what are the join points**
  - points in runtime call graph
  - class members
- **means of identifying join points**
  - pointcuts
  - member signatures (plus …)
- **means of specifying semantics at join points**
  - advice
  - define members

dynamic JPM

static JPM

Evaluation of AspectJ

Strong points

- + well-integrated in Java (including typing)
- + single language (not multiple *ADLs*)
- + many powerful constructs
- + addresses crosscutting concerns
  - □ ‘semantic-based’ queries
  - □ wildcards
- + offers imposition of behavior
- + pointcut composition is supported
- + advice composition is possible, but ...

Evaluation of AspectJ

Limitations

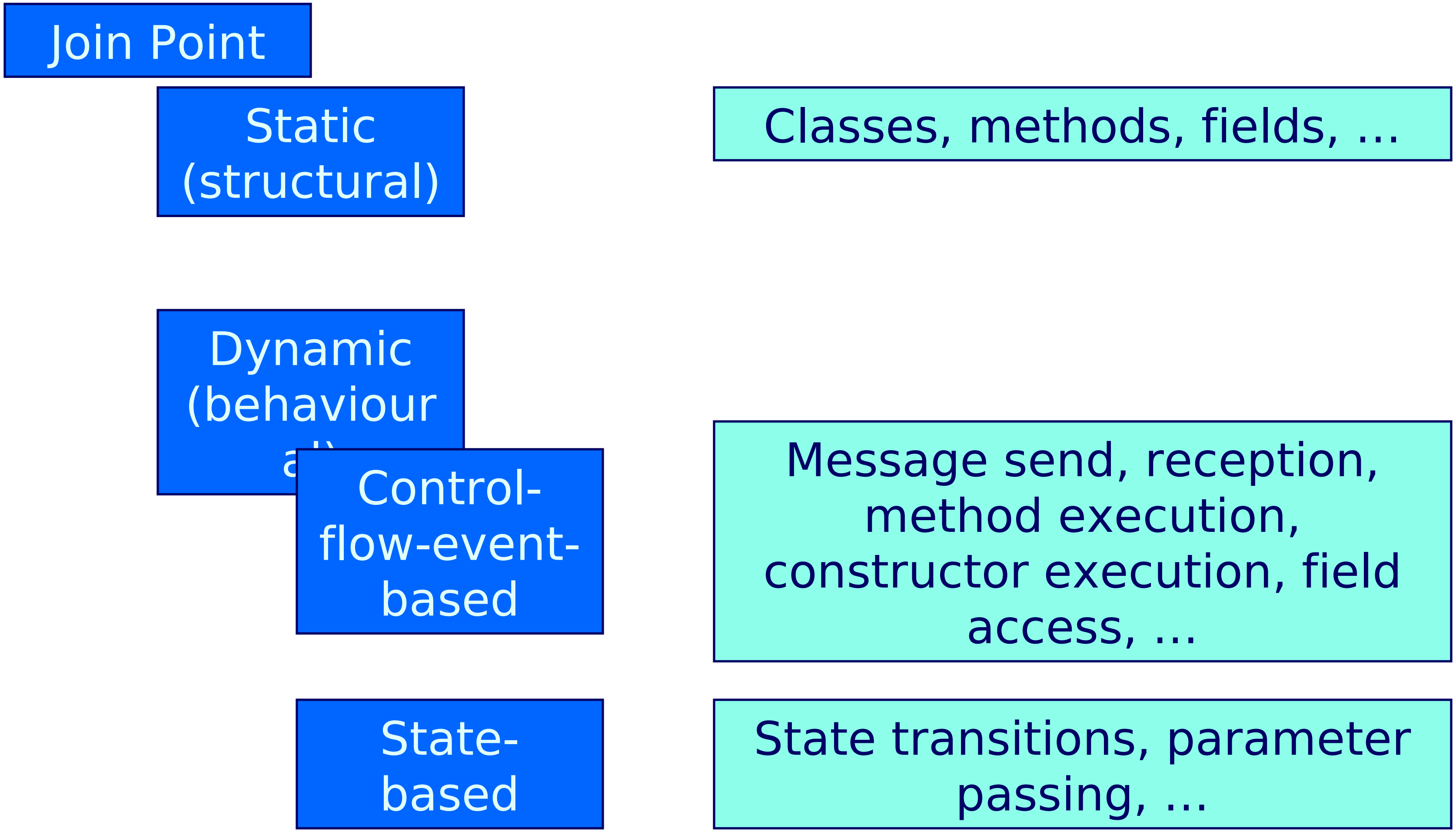
- Advice constructs and ordering are complicated but limited
- Reuse of aspects is primitive/restricted
- Aspects are statically applied (also positive)
- No (few) joinpoints in advice
  - □ **I.e. a 'non-symmetrical model'**
- Composability of aspects is sole responsibility of the programmer(s) !!
  - □ **due to the full expression power within advice**

Part IV: An overview of AOP Languages

Concepts of AOP

- **Join Point Model**
  - Points where behaviour can be invoked implicitly
  - E.g. points in the control flow of a program
- **Pointcut Language**
  - Means of identifying join points
  - E.g. logic predicate language
- **Aspect Behaviour**
  - Means of effecting semantics at identified join points
  - E.g. before/after/around advices

Join Point Models



Join Point

Static (structural)

Classes, methods, fields, …

Dynamic (behavioural)

Control-flow-event-based

Message send, reception, method execution, constructor execution, field access, …

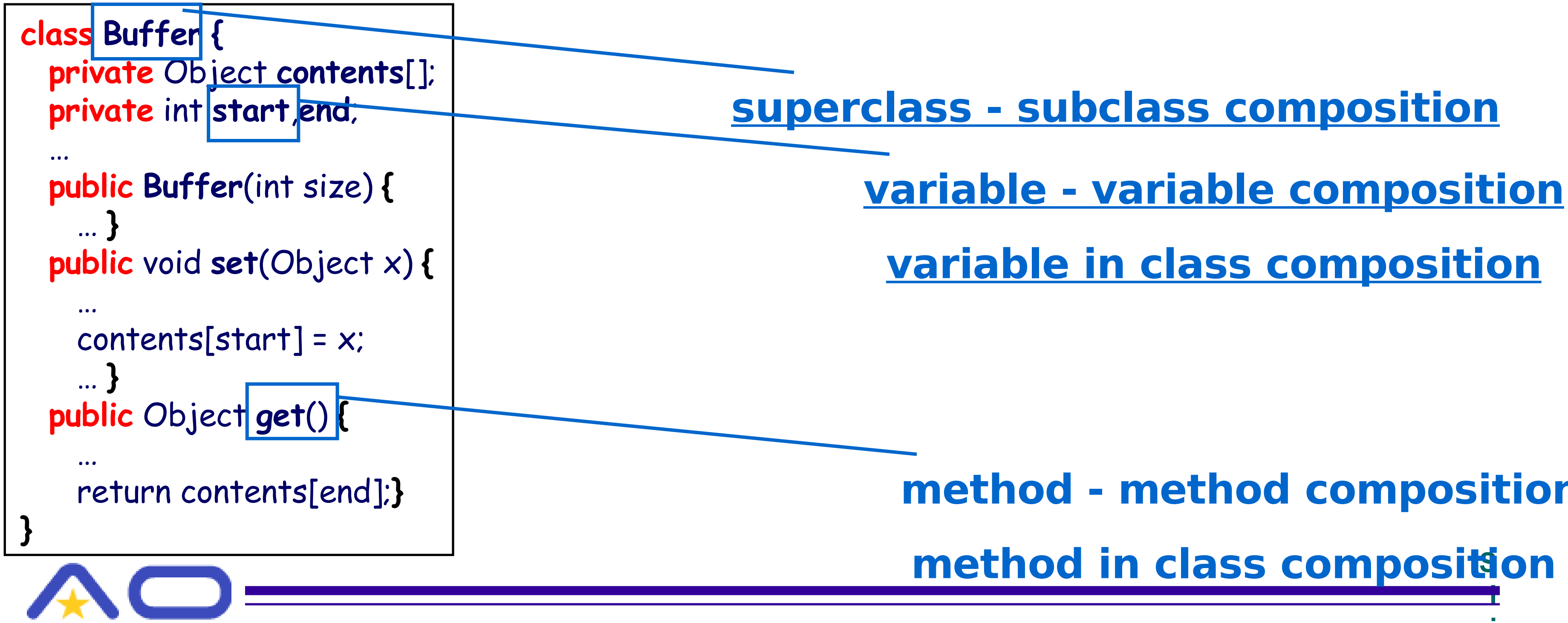
State-based

State transitions, parameter passing, …

Static/Structural JPM

Syntactic/structural program elements

- Methods, variables, classes, etc…



```
class Buffer {
  private Object contents[];
  private int start,end;
  …
  public Buffer(int size) {
    … }
  public void set(Object x) {
    …
    contents[start] = x;
    … }
  public Object get() {
    …
    return contents[end];}
}
```

superclass - subclass composition

variable - variable composition

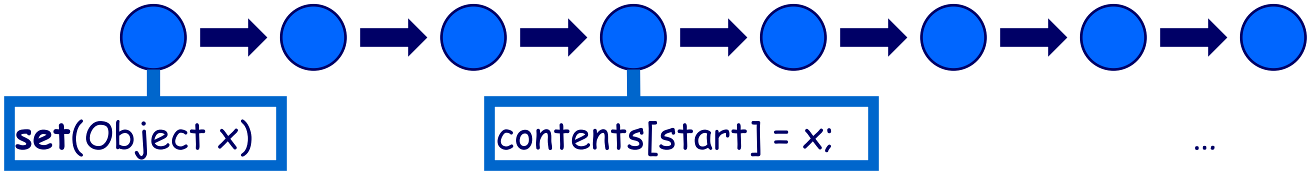
variable in class composition

method - method composition

method in class composition

Dynamic/Behavioural JPM

- **Control flow: events in program execution**
  - **Method invocation, exception handling,…**

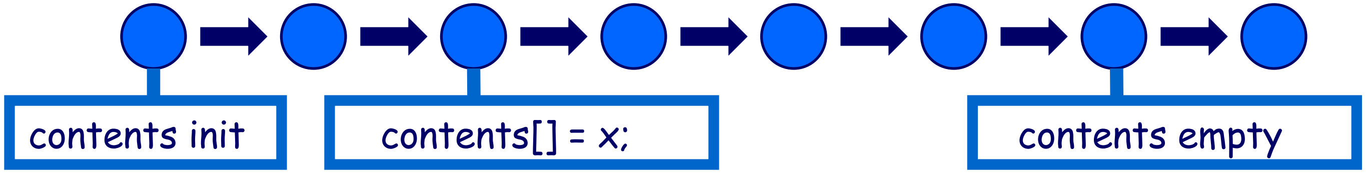


set(Object x)

contents[start] = x;

…

- **Data flow: events in program state**
  - **Changes to values, operations with values, …**



contents init

contents[] = x;

contents empty

Can correspond to various execution events!

Pointcut Languages

- Pointcuts
  - Language — AspectJ pointcuts, Xpath, Logic language, functional language, meta-object protocol, …
  - Properties (predicates)
    - JP properties — Message send, reception, method execution, constructor execution, field access, …
    - Scoping properties — Control flow, data flow, …
    - Program structure

Pointcut Languages

- AspectJ (pattern) matching language
- Logic-based languages
  - Carma, Alpha, Compose*, …
- Meta(-object) Protocols
  - AspectS, …
- Functional languages
  - EAOP, …

Enumeration Pointcut

AspectJ

```
pointcut access():
     execution (void Buffer.set(Object)) ||
     execution (Object Buffer.get());
```

Carma

```
access(?jp) if
   execution(?jp,?method,?args),
  or(methodWithNameInClass(?method, set:,Buffer),
     methodWithNameInClass(?method,get,Buffer))
```

AspectS

```
OrderedCollection
  with: (AsJoinPointDescriptor targetClass: Buffer
                               targetSelector: set:)
  with: (AsJoinPointDescriptor targetClass: Buffer
                               targetSelector: get)
```

Name Pattern-based Pointcut

AspectJ

```
pointcut access():
    execution (void *.set*(Object)) ||
    execution (Object *.get*());
```

Carma

```
access(?jp) if
   execution(?jp,?method,?args),
  or(methodWithNameInClass(?method, {set?},?class),
     methodWithNameInClass(?method,{get?},?class))
```

AspectS

```
Object withAllSubclasses
  collect:[:each | (each selectors)
           select:[:sel|('set*' match: sel)|
                        ('set*' match: sel) ]
           thenCollect: [:sel|AsJoinPointDescriptor
                              targetClass: each
                              targetSelector: sel)]]
```

Property-based Pointcut

AspectJ

```
pointcut access():
     execution(*),
     if(thisJoinPointStaticPart. .... )
```

(Or test in advice)

Carma

```
access(?jp) if
   execution(?jp,?method,?args),
  or(assigns(?method,?iv),returns(?method,?iv)),
  instvarInClass(?iv,?class)
```

AspectS

```
Object withAllSubclasses
  collect:[:each | (each allMethods)
          select:[:meth| methd readsField | …
                                     … … … … ]
          thenCollect:[:meth|AsJoinPointDescriptor
                                    targetClass: each
                                  targetSelector:meth selector)]]
```

Aspect Behaviour

- Advices
  - Most aspect languages
- Domain-specific Aspect Languages
  - COOL + RIDL
  - Business rules (OReA), transactions (Kala), traversals (DemeterJ), etc…
- Join Point Reflection
  - Access to representation of current join point

Advices

Cool

```
coordinator Rectangle {
   selfex set_width, set_height, fill; // area is not selfex
   mutex {set_width, area}; …
}
```

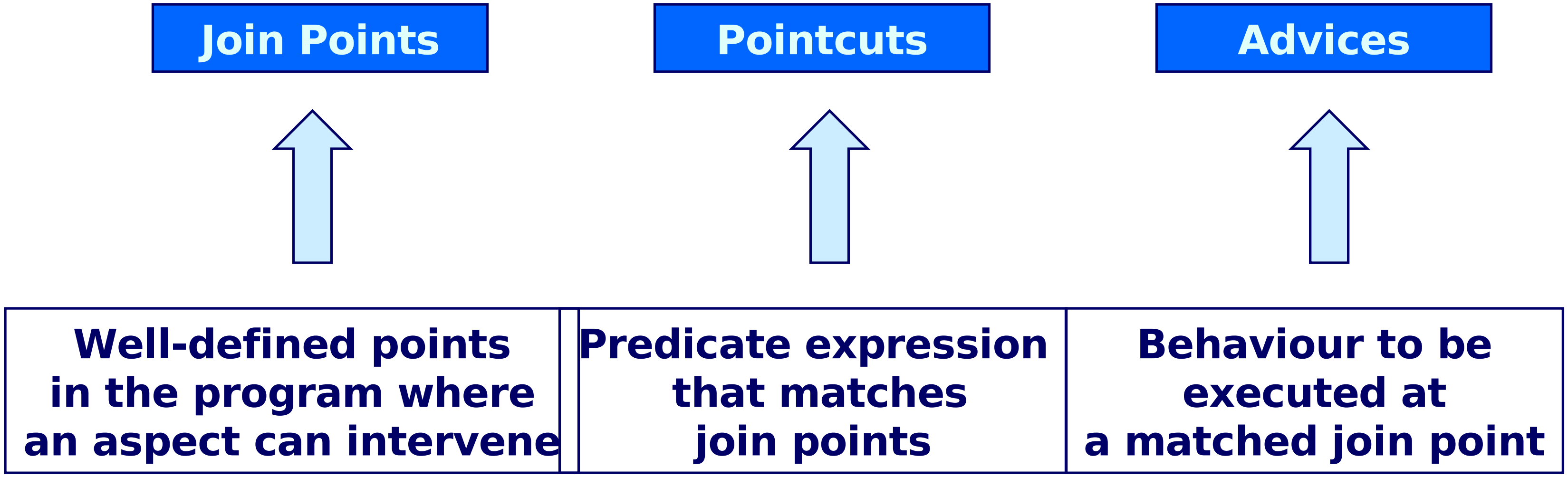
Compose*

```
inputfilters
   selfex:Wait ={ NotSelfActive~>{set_width, set_height, fill} };
   mutex:Wait = { NoMutalActive ~>{set_width, area} };
}
```

AspectJ

```
before (): synchronizationPoint() {
    this.guardedEntry(thisJoinPointStaticPart.getSignature().getName());
 }
after (): synchronizationPoint() {
   this.guardedExit(thisJoinPointStaticPart.getSignature().getName());
 }
```

Conceptual model of AOP



Join Points

Well-defined points in the program where an aspect can intervene

Pointcuts

Predicate expression that matches join points

Advices

Behaviour to be executed at a matched join point

Part V: AOSD Obstacles and Issues

Problems with the application of aspects

&

Issues in the design of AOP techniques

Common Issues in AOSD

- **"fragile pointcut problem"**
  - pointcut declarations result in a high coupling between aspect and base system.
  - These pointcuts are fragile, as non-local changes easily may break pointcut semantics.
- **Breaking Encapsulation (→)**
  - hence aspect abstractions may depend on implementation details of base abstractions
- **"AOSD evolution paradox“ [Tourwe, Brichau & Gybels 2003]**
  - Application evolution through e.g. refactoring tools does not apply to the aspects
  - Fragile pointcuts & breaking encapsulation:
    - **the aspect abstractions may depend too strong on the base abstractions**

More Common AOSD Issues

- **Incremental compilation**
  - Aspect compilers are 'incremental' (AspectJ, Compose*)
  - But due to crosscutting, there is always a 'closed world assumption'
  - and if aspects can break encapsulation, this is even worse.
- **OA Design support yet immature (→)**
  - bad design → bad implementation..
  - e.g. Theme/UML (Clarke et.al.)
  - but als AO requirements analysis, AO architecture design
- **Lack of tool support (debugging, code analysis, etc.)**
  - existing base languages adopted, but sometimes 'tool interference'
- **Aspect reuse & aspect libraries**
  - aspect reuse mechanisms are simplistic and largely unexplored
- **Robustness & Aspect interactions (→)**

AOP vs. Robustness

an Introduction

Discussing the trade-offs in the application of AOP

AOP vs. Robustness

- An important property of AOP is that it reverses the dependencies between modules:
  - **previously, additional behavior was always** explicitly imported **from other modules**
  - **with aspects, behavior is** exported from **(superimposed upon) other modules**
- → this scares many software engineers..
- But we know: with aspects, we can create better modularization
  - → less tangling, less coupling, better maintainability…
- But: *does AOSD improve the ability to reason about the correctness of a system?*

warning! controversial topic ahead

YES...

for 'managers' & software architects

I can now understand how the system works at the top level, and delve into the details of individual pieces, uncluttered by other concerns

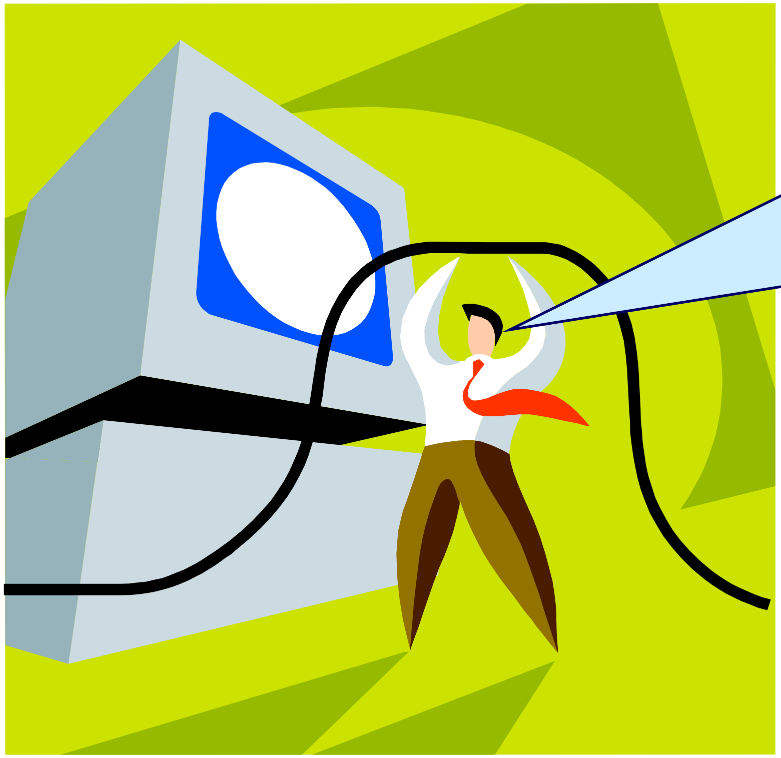
So that means I understand whether the system is correct, more or less..

Christine

YES…

for Aspect module Developers

*John Doe**

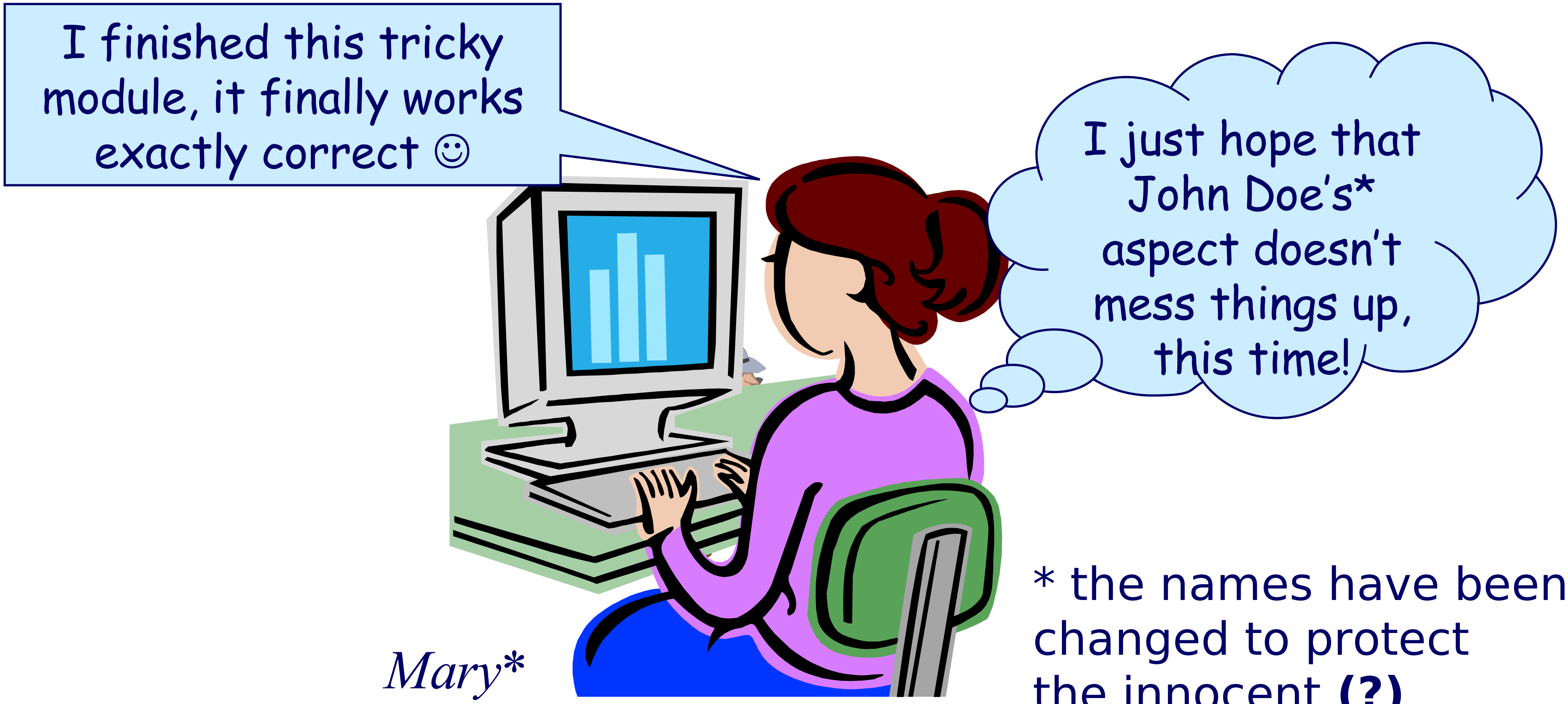


I can now consider the crosscutting concerns by looking at a single module, instead of spread over the system like before!

* the names have been changed to protect the innocent **(?)**

NO...

for base code developers


I finished this tricky module, it finally works exactly correct ☺
I just hope that John Doe's* aspect doesn't mess things up, this time!
Mary*

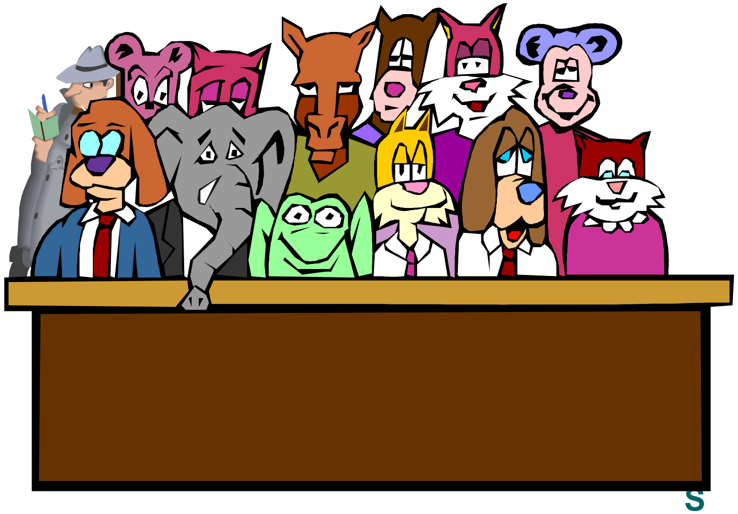
* the names have been changed to protect the innocent **(?)**

Case: John Doe vs. Mary

Is John Doe guilty of writing erroneous aspects?

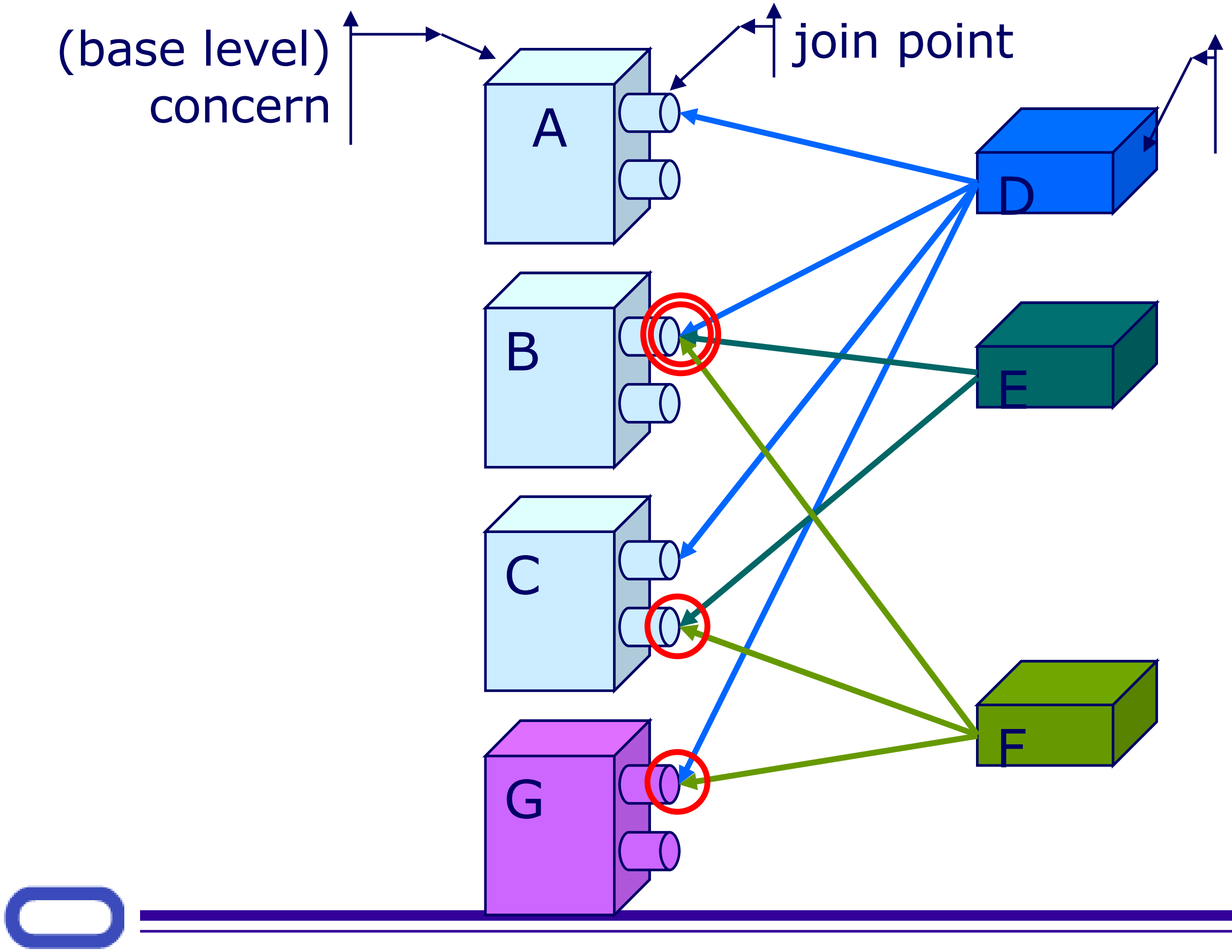
- Yes, he has the responsibility; otherwise Mary's modules would still work correctly
- No, perhaps John could not know that his aspect would interfere with Mary's module without looking at its details

❑ **Verdict of the jury?**



Aspect Composition

at Shared Joinpoints



(base level) concern

join point

(crosscutting) concern

Shared Join Point Appearance

- Develop a class
  - some aspects may interfere with the class.. (tools needed)
- Develop an aspect
  - where will the aspect really be applied (dynamically?)
  - how does it compose with base classes and other aspects?
  - → tools & communication between developers needed
- Develop another class
  - suddenly two existing aspects may apply and interfere..
  - tools may (not?) be able to address this

Case John Doe vs. Mary Reopened

The Observer

Detective Discovers Shocking Truth:

"Not J. Doe, but H. Acker wrote
 Interfering Aspect”

p.5: J. Doe released from prison

**p.6: Detective Arnold S. Pect
declared hero, will run for president**

p.6: Christine Fired, Mary Promoted

**p.12: Tools are needed to prevent
similar tragedies.**

Disclaimer: this does not mean the benefits of aspects cannot outweigh the possible risks (nor does it mean the reverse..)

Aspect Interactions

Composition conflicts,
interference,
composition anomalies,
..

Composition Conflicts (code interference)

- Syntactical
  - e.g. in the case of source code weaving the woven code can no longer compile
- Structural
  - E.g. when introducing existing method or creating cyclic inheritance
- Semantical
  - directly derivable from code
    - e.g. variable access, calling dependencies
  - design intentions
    - "is it a bug, or is it a feature?"
    - e.g. ordering of actions/events

Major Classification

1. **aspect-base composition**
2. **aspect-aspect composition**
   - especially at shared join points
     - **(but not exclusively)**
3. **Due to weaving process/specification**

1. Aspect-base interference

- **Obliviousness is bliss?**
  - **Reversal of import dependency to export dependency brings major software engineering issues**
  - **but: obliviousness ≠ (programmer) ignorance**
    - it is about reducing (cyclic) coupling
    - hence tool support can make a large practical difference
      - **scalability is an issue, though**
  - **Goal: developer responsible for creating a dependency must be accountable for the consequences**
- **kind of interferences:**
  - **behavioral conflicts (changing state or control flow)**
  - **through structural changes**

Encapsulation in AOP

- **Encapsulation is a tool to avoid interference**
  - Encapsulation differs per aspect language
- **e.g. in AspectJ:**
  - allows pointcuts to be dependent on implementation details
  - always allows *modification* of the interface (by enhancement)
  - allows access to *private members* by a superimposed method/advice only for *friend aspects*
- **e.g. in Compose*: strong encapsulation in advice**
- **Aspects are a client of the base classes**
  - c.f. callers are a client of classes -> strong encapsulation
  - c.f. subclasses are a client of superclasses -> declared strong/weak encapsulation (in java)
  - → similar encapsulation rules should hold for aspects

Role of design information in pointcuts

addressing the fragile pointcut problem

- **Design intentions**
    - or: semantic information (semantic properties)
    - **are implicitly present (encoded, hard-wired) in the sources programs**
    - **Semantic property – abstract notion, e.g. behavior of a program element, its intended meaning; realization**
- **Programmers would like to use these as concrete ‘hooks’ for composition (weaving)**
    - **hence *not* rely on the 'accidental' structure and naming conventions of the program**

'Semantic Pointcuts'

expressing design intentions in pointcuts

- Be able to refer to annotations in pointcuts
- Example (in AspectJ 5):
  - **Dedicated Pointcuts**

```
pointcut foo(Customer c):
        target(c) &&
        hasAttribute(c, @PersistentRoot) && …
```

  - **Extending Type Patterns**

```
pointcut updateMethods():
   within(@PersistentRoot *) && execution(@Update * *(..));
```

2. Aspect-Aspect interference

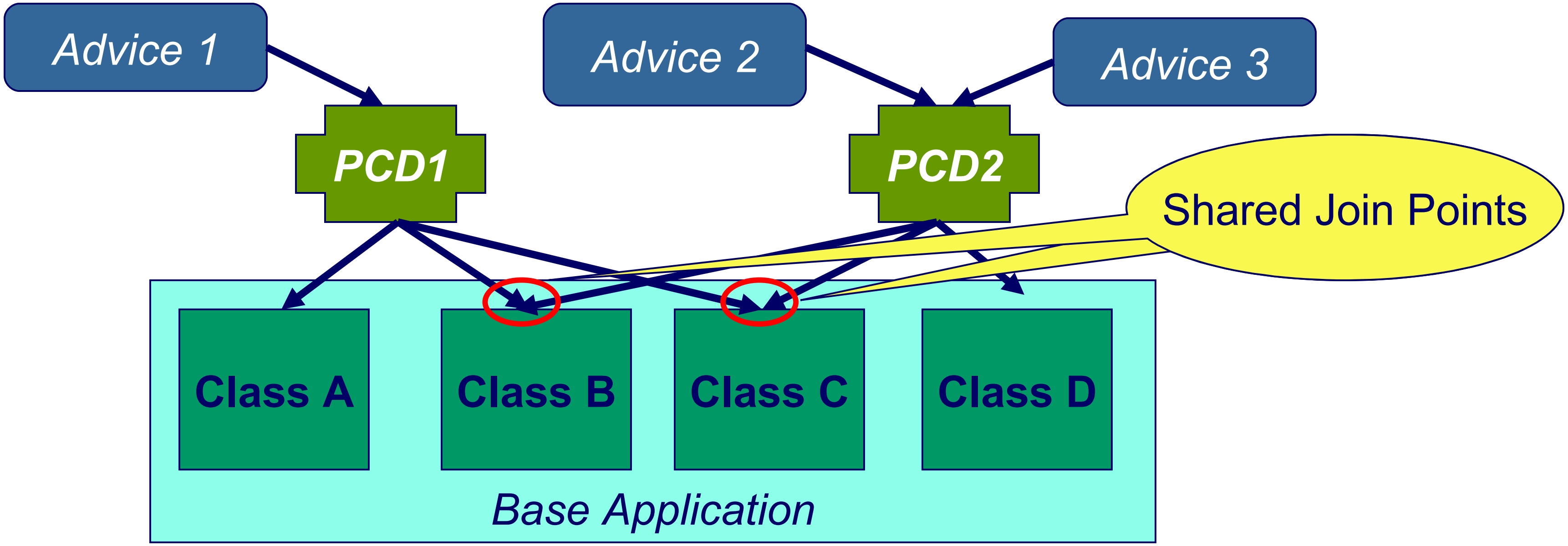
- **Indirectly through the base program**
  - **affecting the base program causes other aspects to break**
    - because its assumptions are violated
  - **can be due to:**
    - state change
    - control flow changes
    - structural changes

Aspect-Aspect interference Cont'd

- **Interference at shared join points**
  - **depending on the correct composition (superimposition) operators**
    - usually just one way of superimposing aspects
  - **dependencies among aspects**
    - e.g. conditional execution
  - **semantic incompatibility**
    - among aspects/advices—by definition
    - only in specific base context
      - **i.e.aspect1/aspect2/baseX conflict**
  - **affected by different orderings**

Aspect-Aspect Interference

- Advices are superimposed on join points



- Due to this superimposition, unintended behavior might emerge

Aspect Interference

- **How can this happen?**
  - **Separation of concerns:**
    - Pointcuts are developed separately
    - Thus, we may not be aware of shared join points
  - **Aspects are usually written in Turing complete advices → AspectJ, AspectC++, AspectWerkz, JBossAOP …**
    - This makes it hard to reason about the sanity of the composition
    - E.g. "What is the intended behavior of <u>not</u> calling the proceed in one specific around advice?"

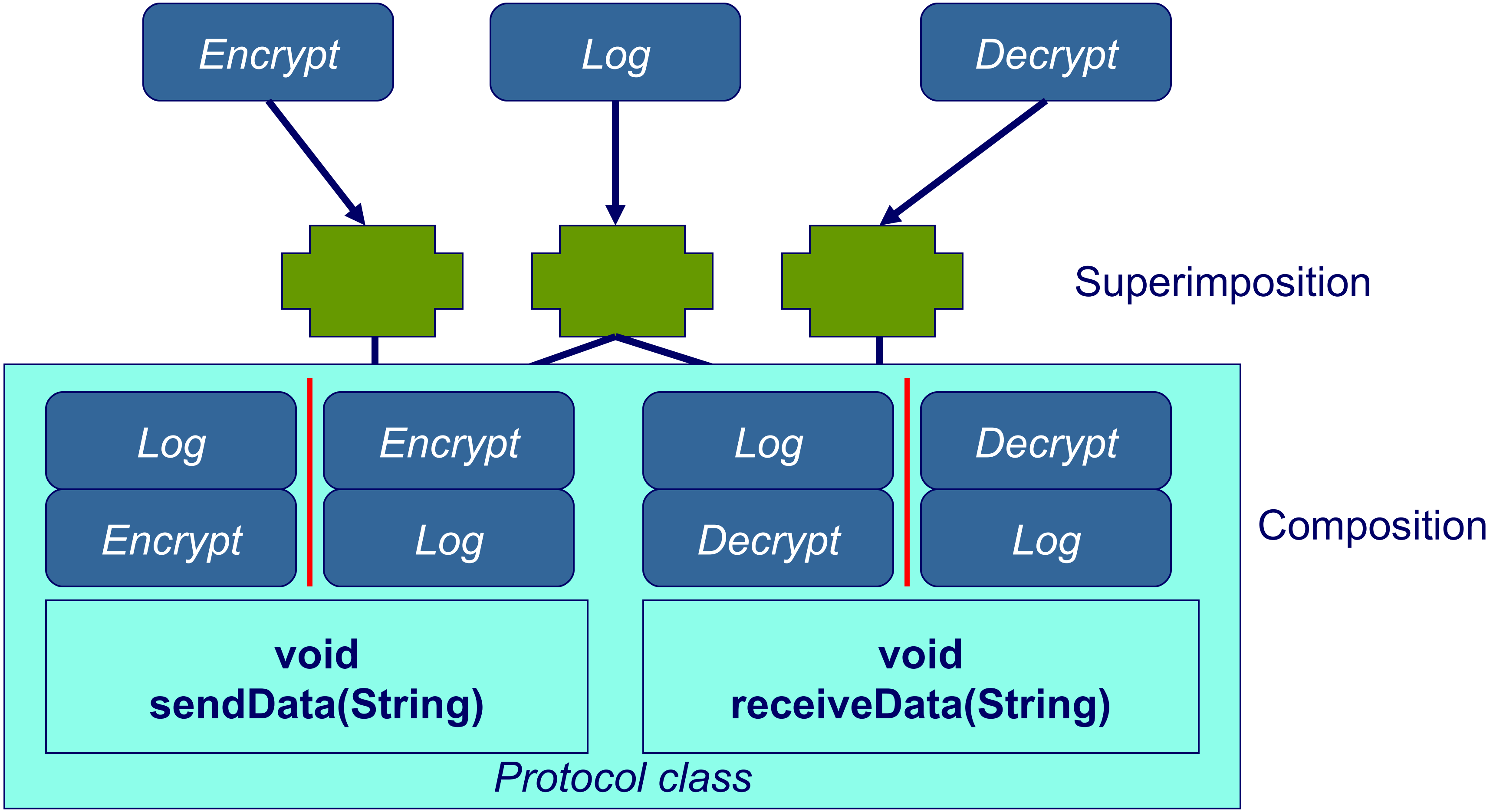
Aspect Interference

- **Semantic conflicts**
  - **Is the intended behavior preserved when composing two aspects?**
  - **Hard to detect as you have to know the semantics of advice**
  - ***"A semantic conflict, is a situation where the composition of multiple advices at a shared join point, influences the behavior of the advices or of the base system, causing the system requirements to be violated."***

Example:

- **Encryption:**
  - **Encrypt all outbound traffic on some protocol**
    - *call(* *.sendData(String)) && args(data) → Encrypt advice*
  - **Decrypt all inbound traffic on some protocol**
    - *call(* *.receiveData(String)) && args(data) → Decrypt advice*
- **Logging:**
  - **Log all sent and received data on the protocol:**
    - *call(* *.receiveData(String) || * *.sendData(String)) && args(data) → Log advice*

Example:


Encrypt
Log
Decrypt
Superimposition
Log
Encrypt
Encrypt
Log
Log
Decrypt
Decrypt
Log
Composition
void
sendData(String)
void
receiveData(String)
Protocol class

Example:

- Both orderings are correct from a compiler point of view!
- However, depending on the requirements one order might be intended.
  - In a hostile domain we want to ensure that no unencrypted data is read.
  - In a protocol debugging domain we need to read the unencrypted data.
- Assume we want the latter option:
  - Log before Encrypt and Decrypt before Log

Other examples

- “If the authorization aspect denies access for a state-changing operation, a second aspect may not be executed"
  - e.g. the authorization around advice does not do a proceed()
- "combining a real-time constraint aspect and a (possibly blocking) synchronization constraint"
- "two advices that both modify e.g. the arguments or return value of a call"
  - unless these are associative modifications

3. Aspect ordering

- **Ordering of advice at shared join points**
  - **execution 'must' be sequential**
    - always an order is chosen
  - **ordering *may* affect behavior**
    - desired order is determined by requirements!
  - **→ explicit, finegrained ordering specification is required**
    - AspectJ: 'declare precedence'
    - EAOP: advice composition (→)
    - Compose*: declarative, fine-grained composition specification
      - **ordering**
      - **conditional execution**
      - **static constraints**

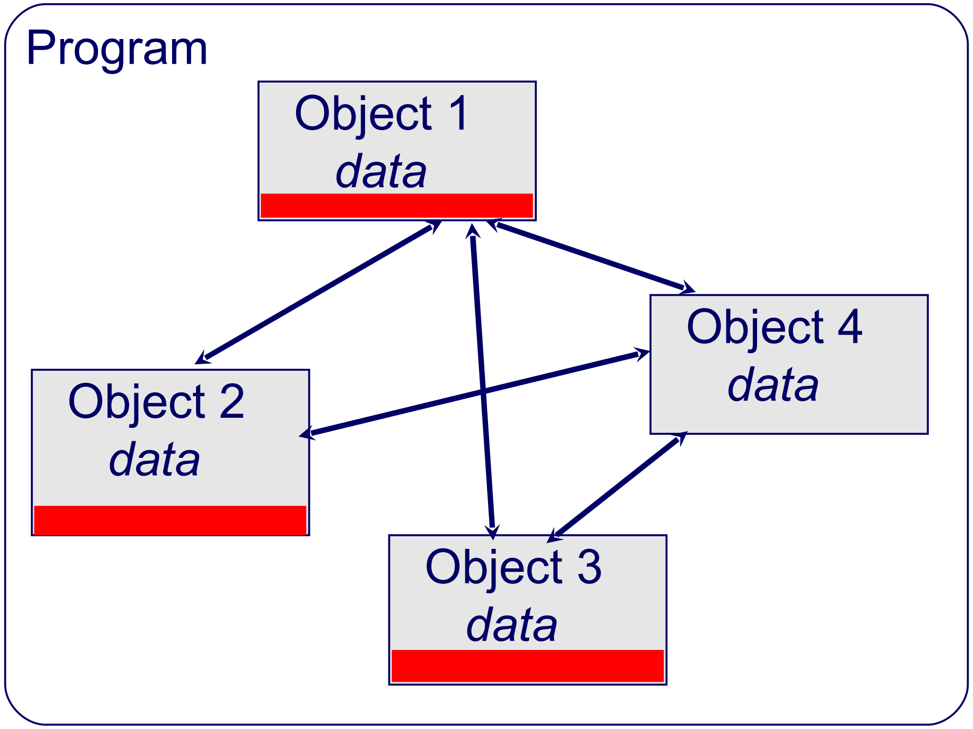
Conclusion

- AOP is no 'silver bullet' (..)
  - brings substantial new modularization possibilities
  - there are many different language design alternatives (not all explored yet)
  - don't use aspects for everything!
    - not a patching mechanism!
    - AODesign is important
  - still a lot of improvements/research needed

- The End

Lost Puppies

Crosscutting Concerns



Concern	Implementation
A	Object 1
B	Object 2
C	Object 3
D	Object 4
E	Object 1,2,3

Typical examples: synchronisation, error handling, timing constraints, user-interface, login security, business rules, ...

Proposal for aspect interference classification-1

Direct Interactions between aspect & base level:

- Augmentation:
  - After superimposition, the entire body of the method always executes.
  - e.g. logging
- Narrowing:
  - After superimposition, either the entire body of the method executes or none of the body executes— in effect, the advice conditionally executes the method.
  - e.g. security or consistency checks
- Replacement:
  - After superimposition, the method does not execute at all — the advice replaces the behavior of the method with completely new behavior.
  - e.g. advice that uses a static pointcut specification to check a safety condition
- Combination:
  - After superimposition, the method and aspect combine in some other way to produce potentially new behavior.

Proposal for aspect interference classification-2

Indirect Interactions between aspect & base level:

- Orthogonal scopes:
  - The advice and method access disjoint fields.
- Independent scopes:
  - Neither the advice nor the method may write a field that the other may read or write.
- Observation of method scope:
  - The advice may read one or more fields that the method may write but they are otherwise independent.
- Actuation of method scope:
  - The advice may write one or more fields that the method may read but they are otherwise independent.
- Interference of scopes:
  - The advice and method may write the same field.

3. Ordering of Weaving Operations

- **Ordering of weaving operations**
  - **if two weaving operations are dependent**
    - e.g. counting variable access & logging
  - **the order in which weaving takes place makes a difference!**

4. need for aspect-based design

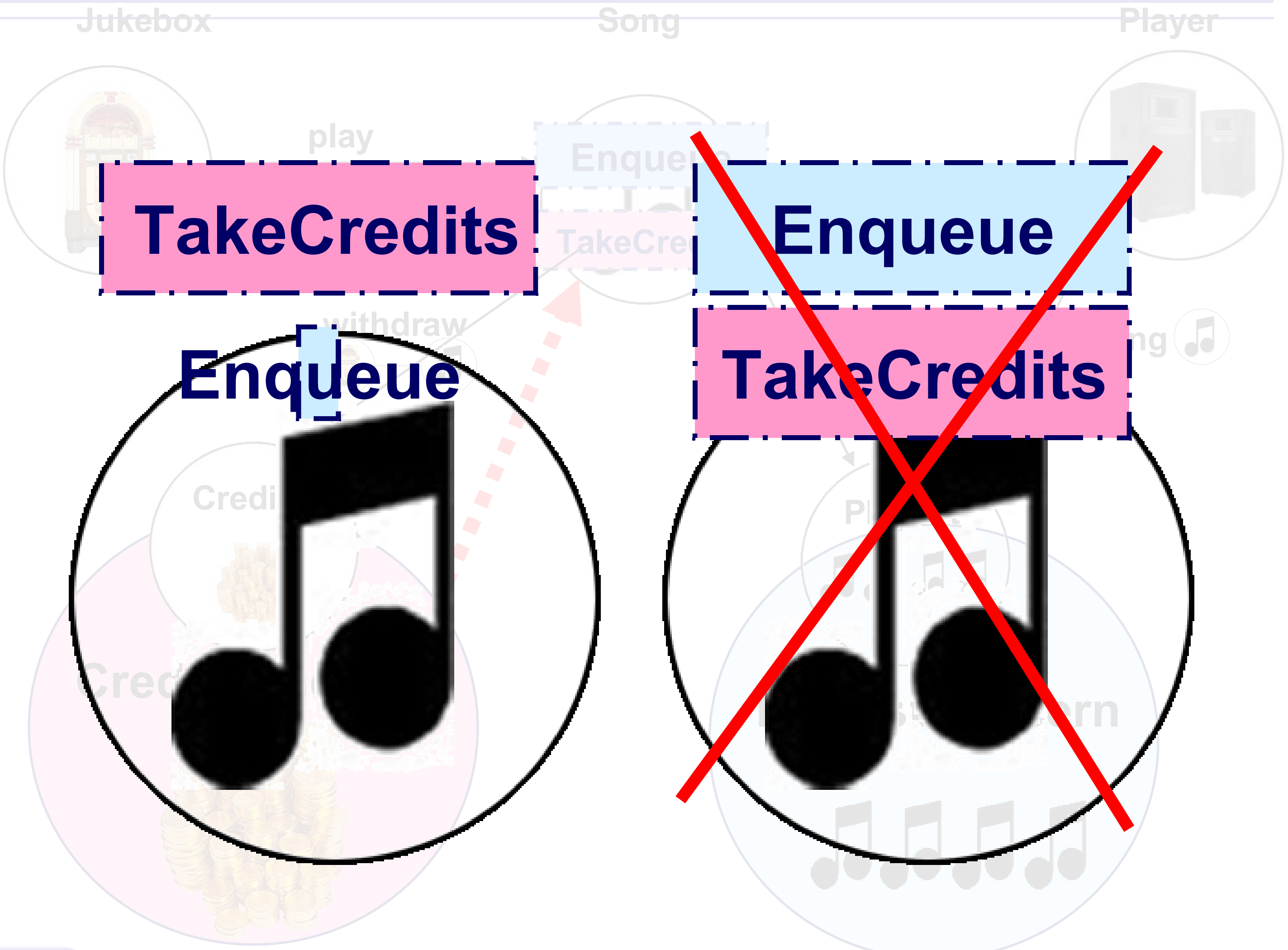
Composable designs

- Software with aspects must be designed as such from the start!
- Software development methods still largely missing:
  - requirements specificaiton
  - architecture design
  - notations (UML)
  - tool support
  - etc..
- How can we create composable modules?
  - powerful compostion operators
  - by design

model composability & composable language

- **Before composing software modules, they must be designed composable:functional composability**
  - sound & logical to compose at the conceptual / semantic level
  - procedural composability
    - □ **the dependencies and interactions between composed components must ‘match’ (be compatible)**
- **The composition capabilities of the language are the 2nd critical factor**
  - e.g. compare OOPL with AOPL..
  - or lack of advice composition (ordering) specification in AOPL

Example Jukebox



TakeCredits

Enqueue

Enqueue

TakeCredits