UNIVERSITÄT
BERN

Software Design and Evolution

11. Dynamic Analysis

Jorge Ressia

Roadmap

- Motivation
- Sources of Runtime Information
- Dynamic Analysis Techniques
- Advanced Dynamic Analysis Techniques
- Dynamic analysis in a Reverse Engineering Context
- What can we achieve with all this?
- Conclusion

Roadmap

- **Motivation**
- Sources of Runtime Information
- Dynamic Analysis Techniques
- Advanced Dynamic Analysis Techniques
- Dynamic analysis in a Reverse Engineering Context
- What can we achieve with all this?
- Conclusion

What does this program do?

```
#include <stdio.h>main(t,_,a)char *a;{return!0<t?t<3?
main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):1,t<_?main(t
+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?main(2,_+1,"%s %d %d
¥n"):9:16:t<0?t<-72?main(_,t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}
+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#¥;#q#n+,/+k#;*
+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l ¥q#'+d'K#!/+k#;q#'r}eKK#}
w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# ¥){nl]!/
n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw'
¥iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c ¥;;
{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}
{rl#'{n' ')# ¥}'+}##(!!/"):t<-50?_==*a?
putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)  :0<t?
main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,"!ek;dc i@bK'(q)-
[w]*%n+r3#l,{}:¥nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Thomas Ball, The Concept of Dynamic Analysis, FSE'99

Source code can be hard to read and understand.
This is a legal C program, an extreme case of course.

This is intentionally obfuscated code. Obfuscation is a set of transformations that preserve the behavior of the program but make the internals hard to reverse-engineer.
- Programming contests
- Prevent reverse engineering

Finding Features

Software Feature:
A distinguishing characteristic of a software item.

IEEE 829

Bug reports often expressed in terms of Features.

The software engineer needs to maintain a mental map between features and the parts of the code that implement them.

Features are not implemented in one class. Their implementation spreads out over lots of classes. The behavior consist of objects that collaborate at runtime.

"Change requests and bug reports are usually expressed in a language that reflects the features of a system"

[Mehta and Heinemann 2002]

Bug reports often expressed in terms of Features.

The **"add contacts"** feature

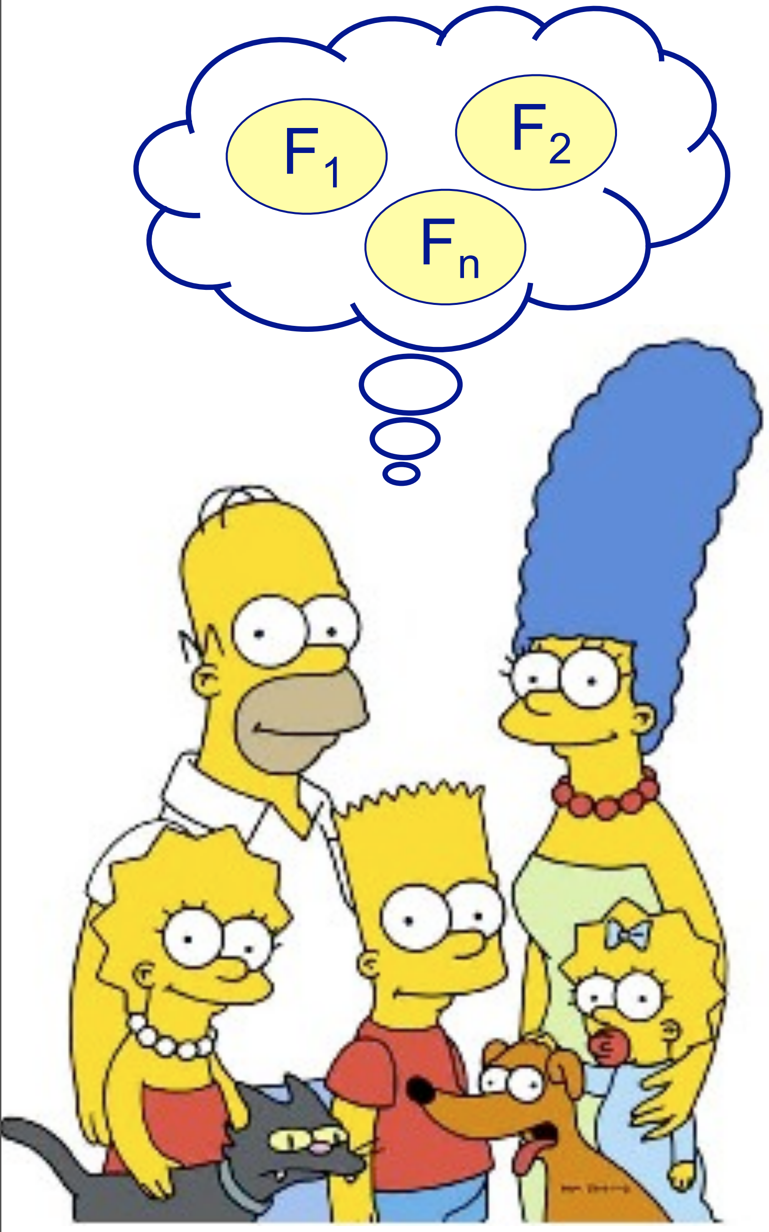
The software engineer needs to maintain a mental map between features and the parts of the code that implement them.

Features are not implemented in one class. Their implementation spreads out over lots of classes. The behavior consist of objects that collaborate at runtime.

"Change requests and bug reports are usually expressed in a language that reflects the features of a system"

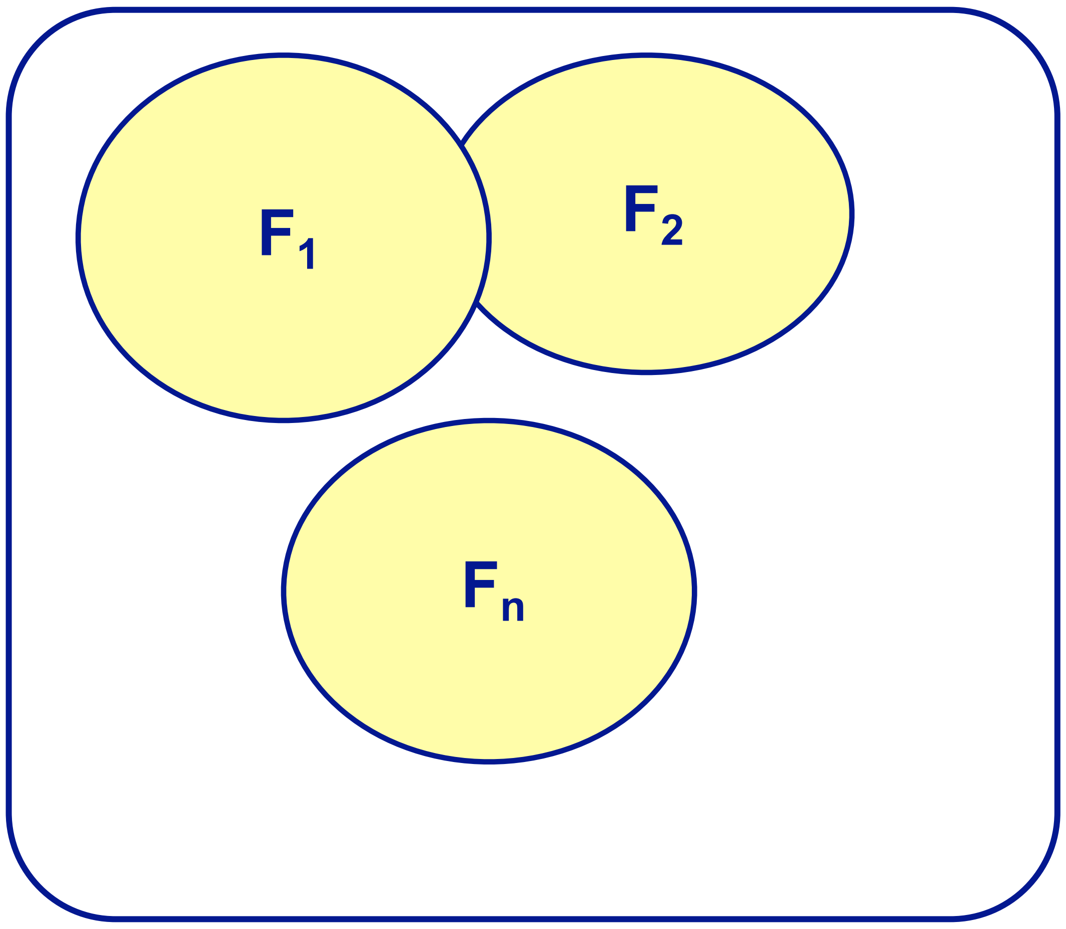
[Mehta and Heinemann 2002]

Feature-Centric Reverse Engineering



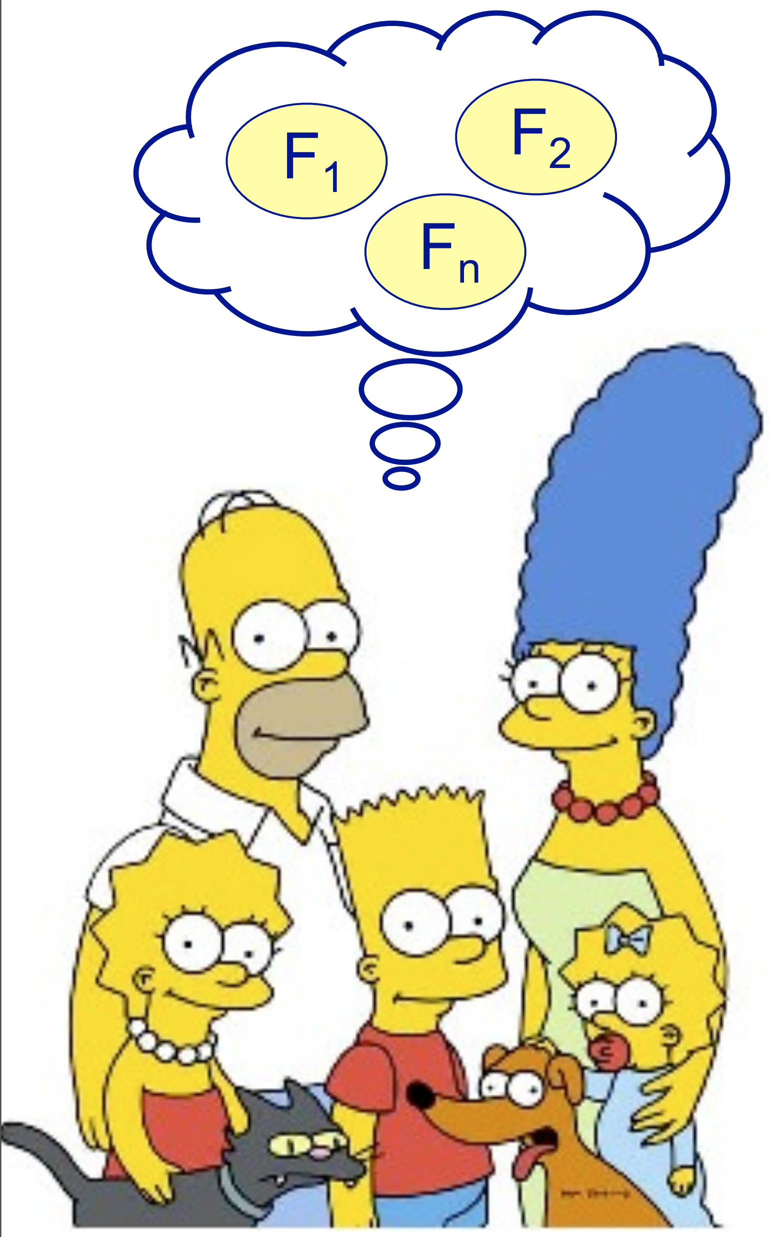
Users

Software System



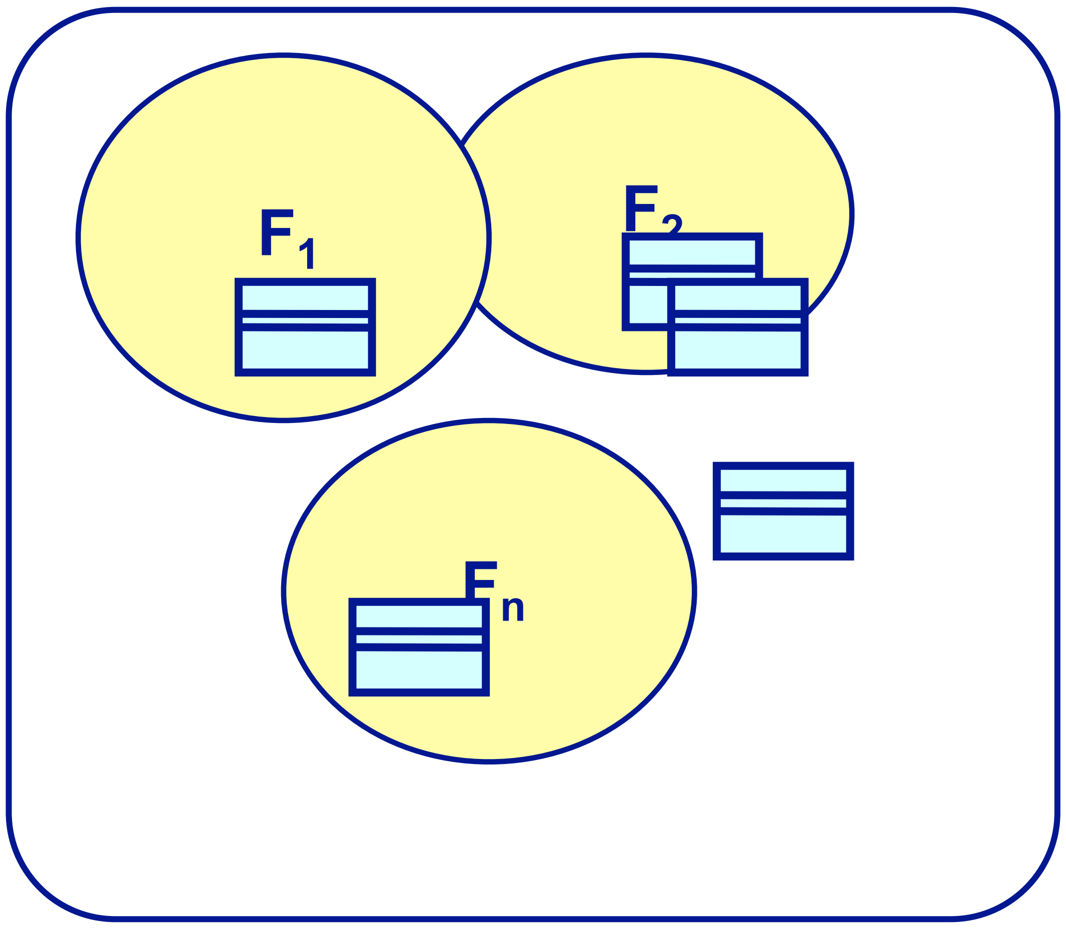
Software developer

Feature-Centric Reverse Engineering



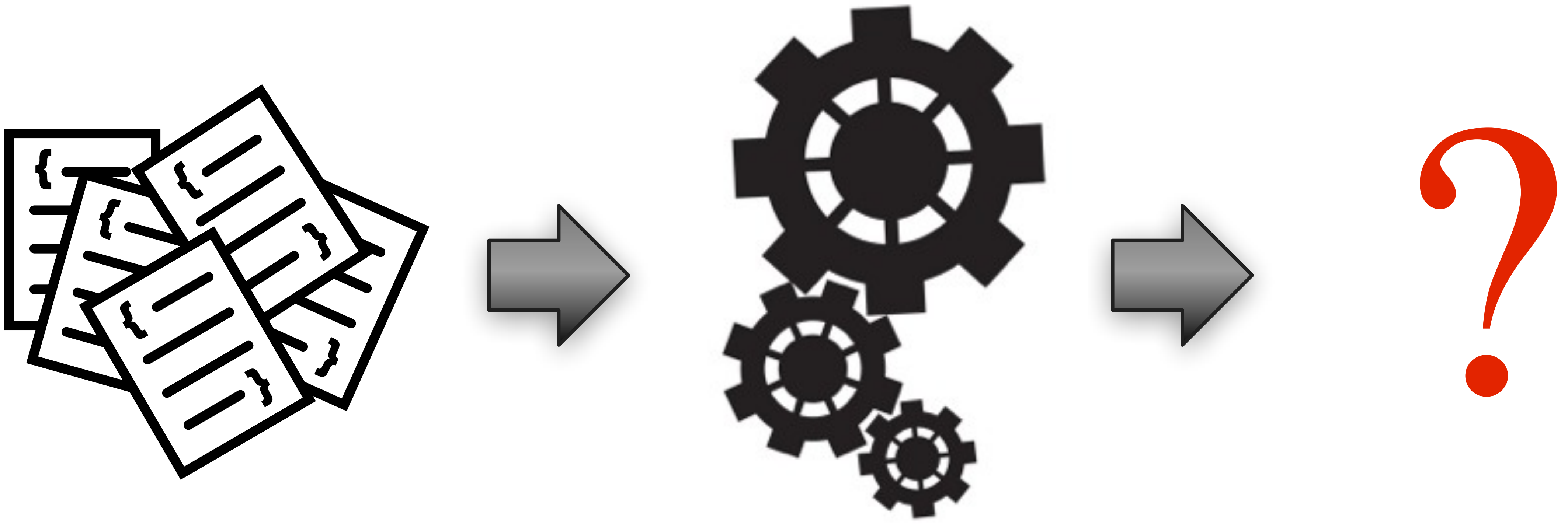
Users

Software System



Software developer

Finding Features



I have a system and I need to find the features. Which part of the system belongs to which features?

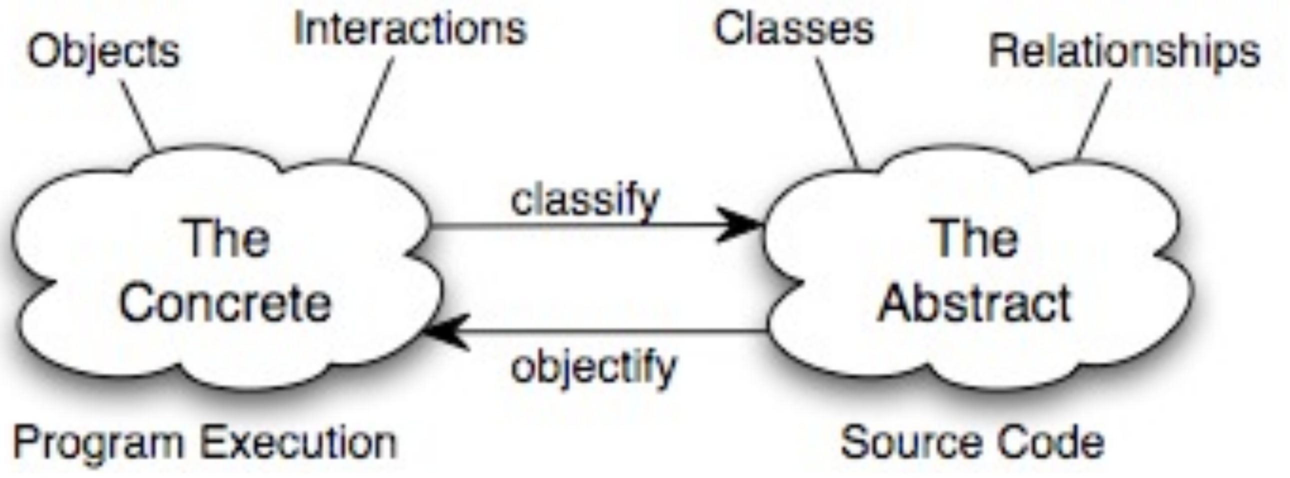
What is Dynamic Analysis?

The investigation of the properties of a *running* software system over one or more executions

(Static analysis examines the program code alone)

Why Dynamic Analysis?

Gap between run-time structure and code structure in OO programs



> *Trying to understand one [structure] from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice-versa.*
>
> -- Erich Gamma et al., Design Patterns

Programs can be hard to understand from their source code alone.
The static code does not explicitly reflect the dynamic behavior.
The code structure consists of classes in fixed inheritance relationships. Difficulties:
OO source code exposes a class hierarchy, not the run-time object collaborations

- Collaborations are spread throughout the code
- Polymorphism may hide which classes are instantiated

A program's run-time structure consists of rapidly changing networks of communicating objects.

Roadmap

- Motivation
- **Sources of Runtime Information**
- Dynamic Analysis Techniques
- Advanced Dynamic Analysis Techniques
- Dynamic analysis in a Reverse Engineering Context
- What can we achieve with all this?
- Conclusion

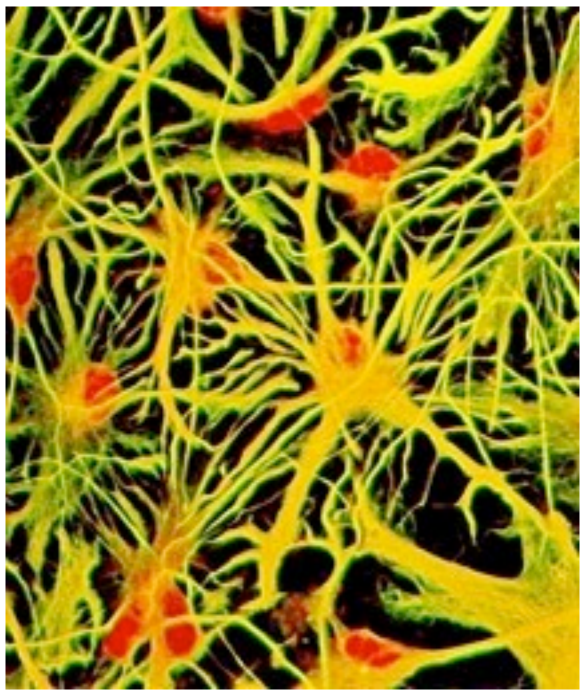
Runtime Information Sources

Many possibilities: hardware monitoring, tracing method execution, values of variables, memory usage etc...

External view

execute program and watch it from outside

Internal view



instrument program and watch it from inside

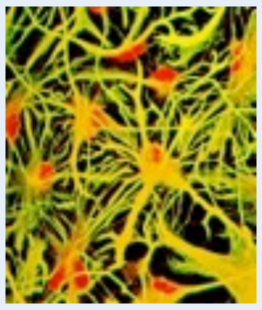
External View

Program output, UI (examine behavior, performance, …)

Analyze used resources

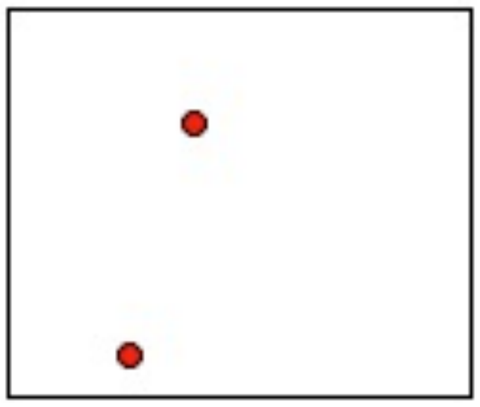
- CPU and memory usage (top)
- Network usage (netstat, tcpdump)
- Open files, pipes, sockets (lsof)

Examine logs (syslog, web logs, stdout/stderr, …)

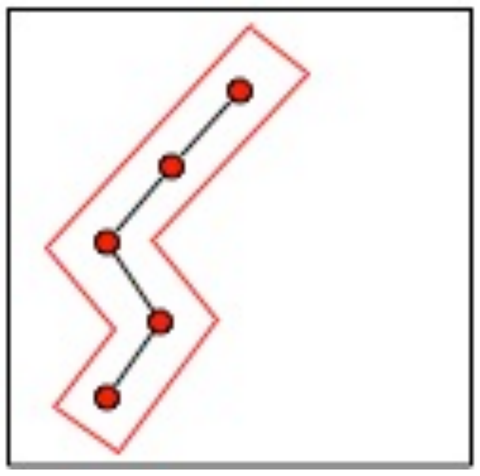


Internal View

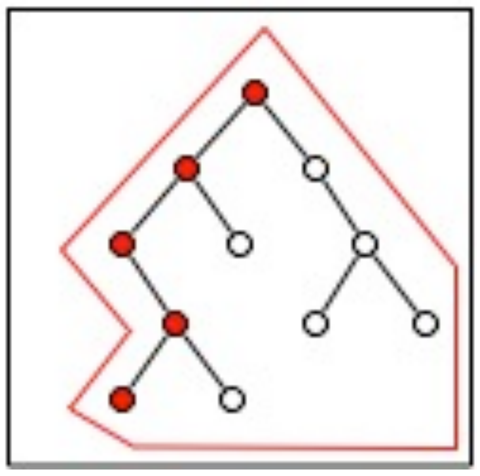
Log statements in code



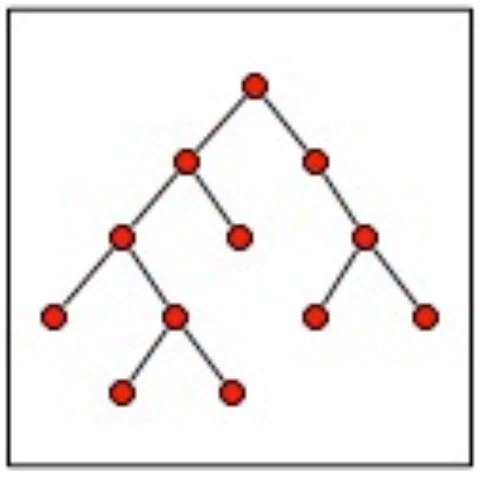
Stack trace



Debugger



Execution trace



Many different tools are based on tracing: execution profilers, test coverage analyzers, tools for **reverse engineering…**

- Logs: Single points in the execution.
- Stack trace: snapshot of the current stack
- Debugger: interactive, allows one to step into future method executions. Not persistent.
- Tracing: full history of all method executions

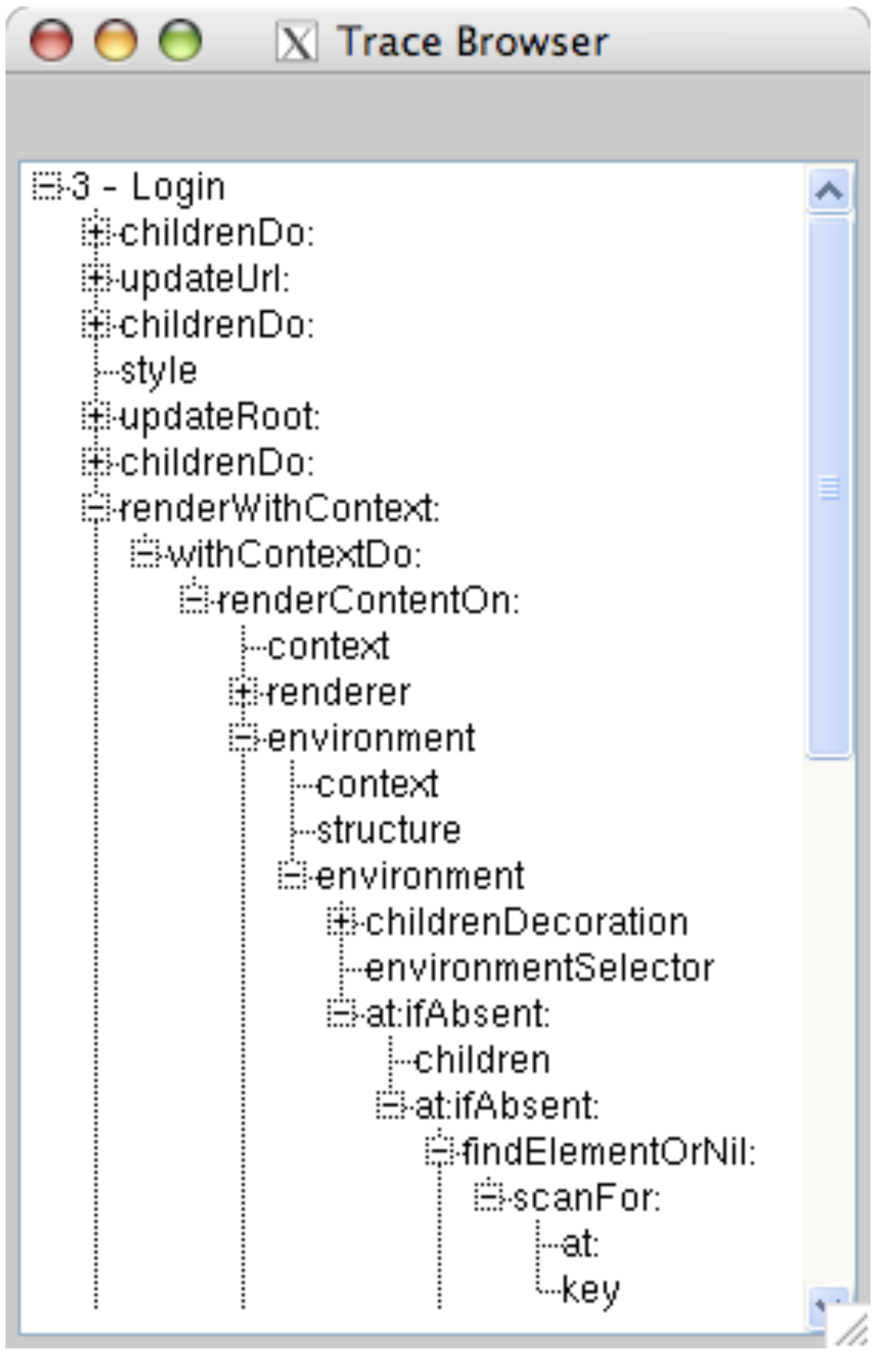
Execution Tracing

How can we capture “full” OO program execution?

Trace entry and exit of methods

Additional information:

- receiver and arguments
- class instantiations
- …?



Trace Browser

- 3 - Login
  - childrenDo:
  - updateUrl:
  - childrenDo:
  - style
  - updateRoot:
  - childrenDo:
  - renderWithContext:
    - withContextDo:
      - renderContentOn:
        - context
        - renderer
        - environment
          - context
          - structure
          - environment
            - childrenDecoration
            - environmentSelector
            - at:ifAbsent:
              - children
              - at:ifAbsent:
                - findElementOrNil:
                  - scanFor:
                    - at:
                    - key

For now we consider this approach

Sequence and nesting: construct tree structure
Additional:
-receiver and arguments
-Return values
-Object creation
-Current feature
-Distinguish process (analyze concurrency properties)

Object referencing relationships not captured.
Object graph at particular point in time cannot be reconstructed, now, how it evolves

Tracing Techniques

Instrumentation approaches
- Sourcecode modification
- Bytecode modification
- Wrapping methods (Smalltalk)

Simulate execution (using debugger infrastructure)

Sampling

At the VM level
- Execution tracing by the interpreter
- (Dynamic recompilation, JIT)

Simulate execution: slow, but very precise control possible

Sampling: mainly used for profiling

Dynamic recompilation:

- control optimizations: compile hot blocks/paths/procedures to machine code
- data optimizations: garbage collection: move objects for locality

Technical Challenges

> Instrumentation influences the behavior of the execution

> Overhead: increased execution time

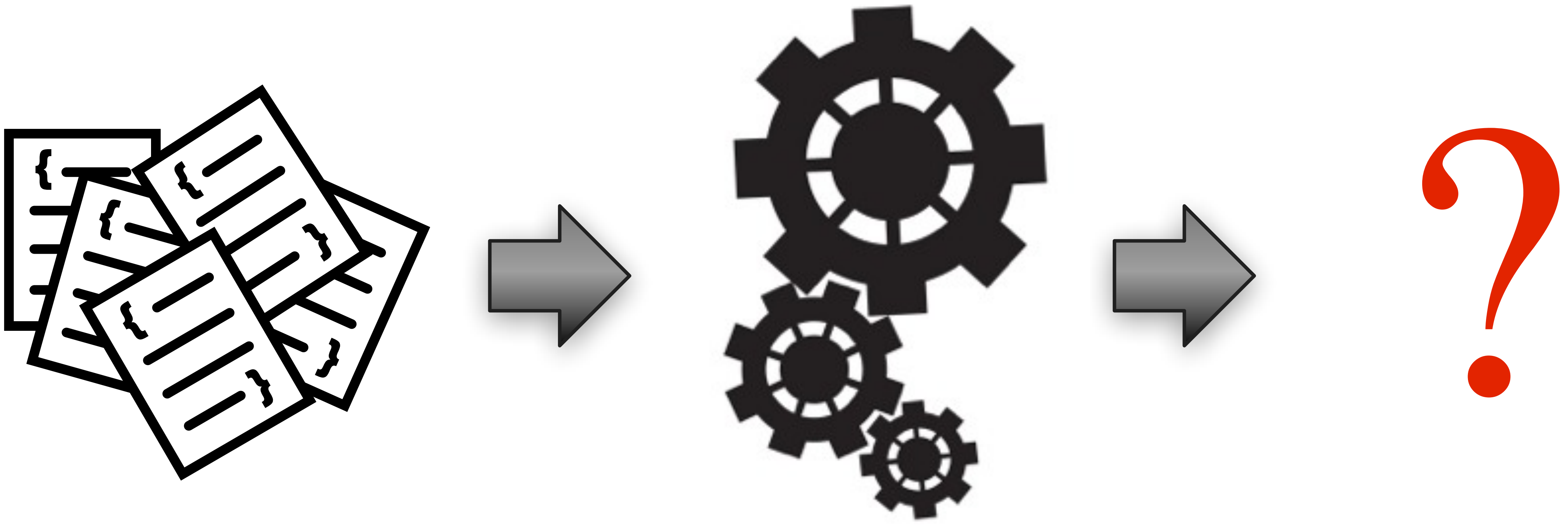
> Large amount of data

> Code also used by the tracer, library and system classes cannot be instrumented
    -> Trace at the VM level
    -> Scope instrumentation (Changeboxes)

Roadmap

- Motivation
- Sources of Runtime Information
- **Dynamic Analysis Techniques**
- Advanced Dynamic Analysis Techniques
- Dynamic analysis in a Reverse Engineering Context
- What can we achieve with all this?
- Conclusion

Feature Analysis



I have a system and I need to find the features. Which part of the system belongs to which features?

Loggers - low tech debugging

> *"…debugging statements stay with the program;*
> *debugging sessions are transient. "*
>
> *Kerningham and Pike*

```
public class Main {

public static void main(String[] args) {
        Clingon aAlien = new Clingon();
        System.out.println("in main ");
        aAlien.spendLife();
        }
}
```

Inserting log statements into your code is a low-tech method for debugging it.

It may also be the only way because debuggers are not always available or applicable.

This is often the case for distributed applications.

Loggers - low tech debugging

> *"…debugging statements stay with the program;*
> *debugging sessions are transient. "*
> *Kerningham and Pike*

Very messy!

```
public class Main {

public static void main(String[] args) {
        Clingon aAlien = new Clingon();
        System.out.println("in main ");
        aAlien.spendLife();
        }
}
```

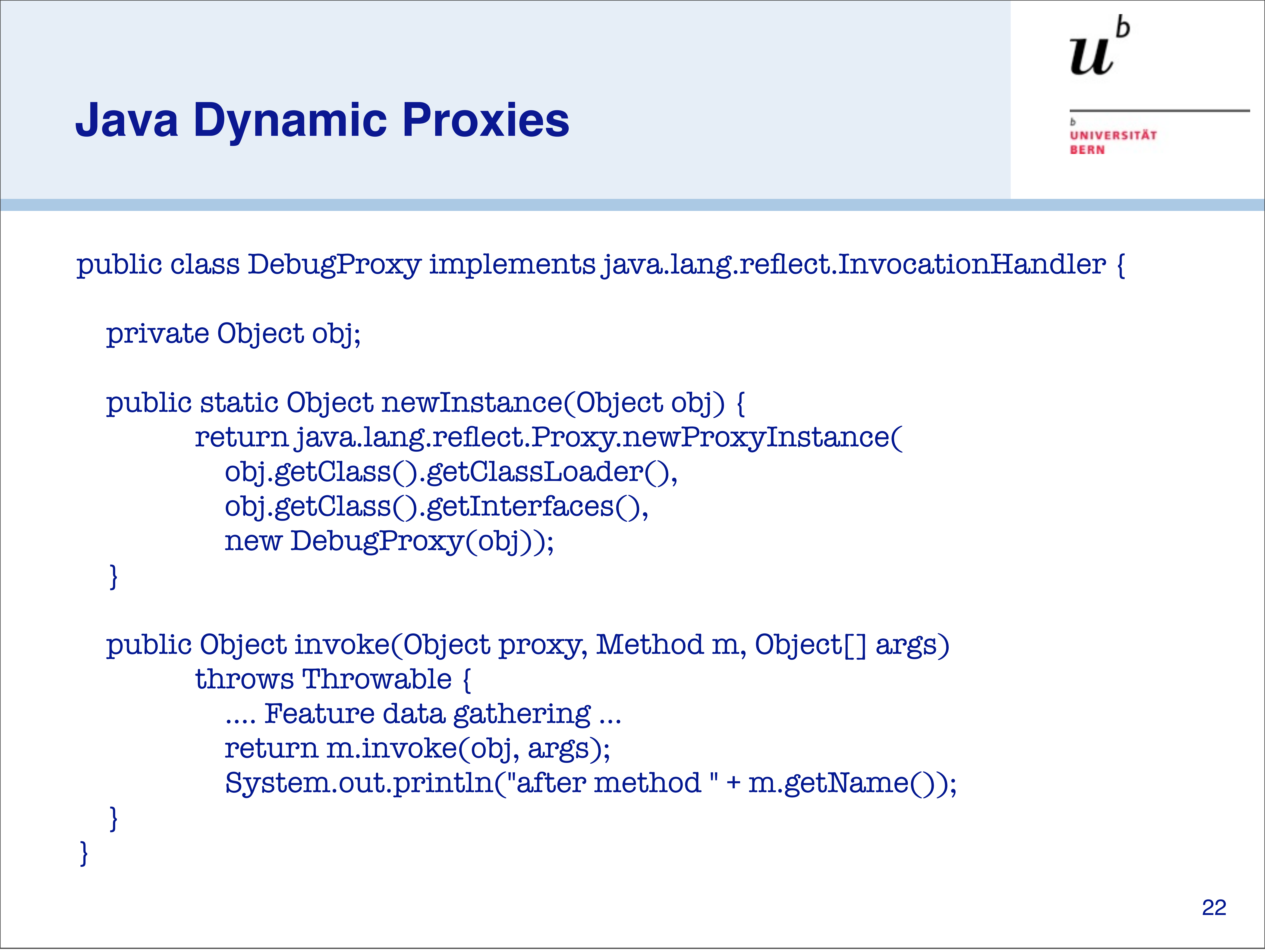
Inserting log statements into your code is a low-tech method for debugging it.

It may also be the only way because debuggers are not always available or applicable.

This is often the case for distributed applications.

Smalltalk Mechanisms

- become: function
- Method Wrappers
- Anonymous Classes



Java Dynamic Proxies

```
public class DebugProxy implements java.lang.reflect.InvocationHandler {

   private Object obj;

   public static Object newInstance(Object obj) {
            return java.lang.reflect.Proxy.newProxyInstance(
               obj.getClass().getClassLoader(),
               obj.getClass().getInterfaces(),
               new DebugProxy(obj));
   }

   public Object invoke(Object proxy, Method m, Object[] args)
            throws Throwable {
               .... Feature data gathering ...
               return m.invoke(obj, args);
               System.out.println("after method " + m.getName());
   }
}
```

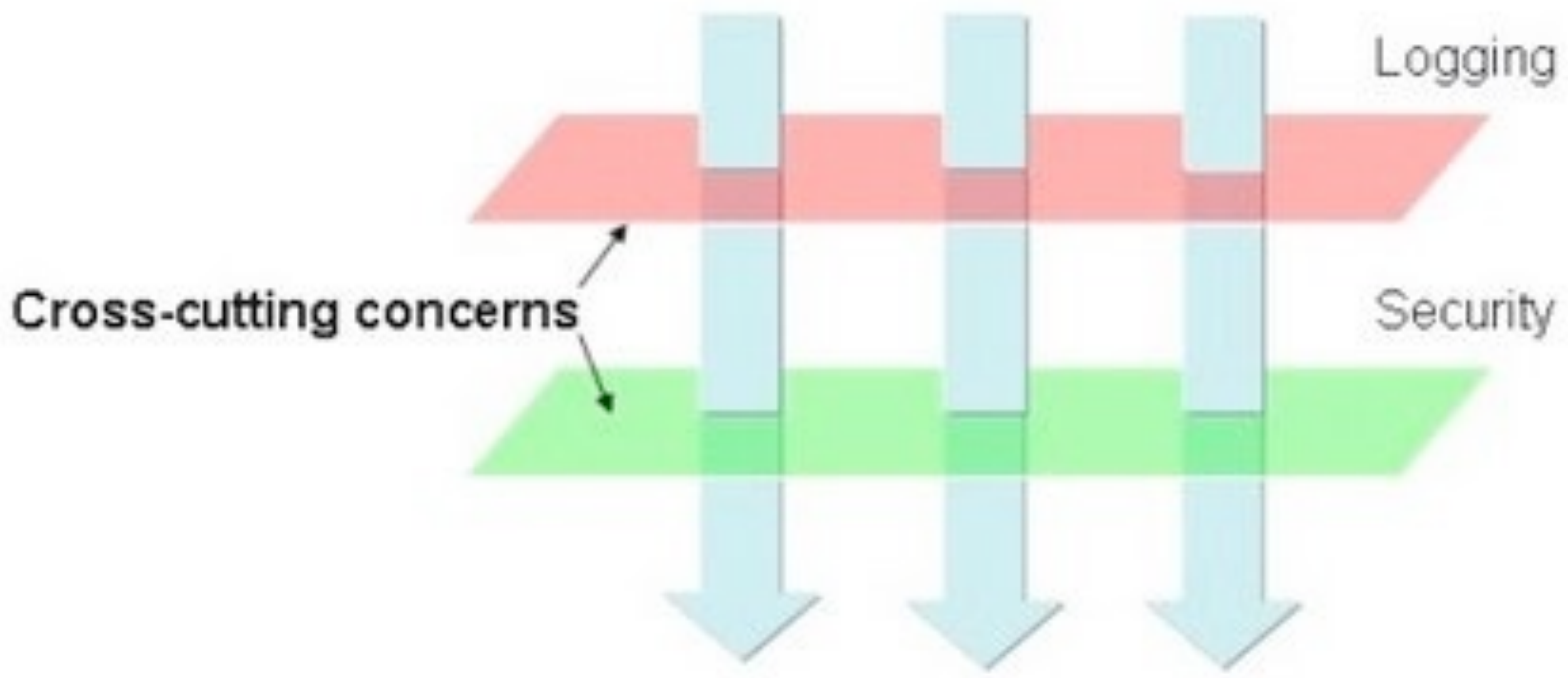
http://docs.oracle.com/javase/1.3/docs/guide/reflection/proxy.html

Aspect Oriented Programming

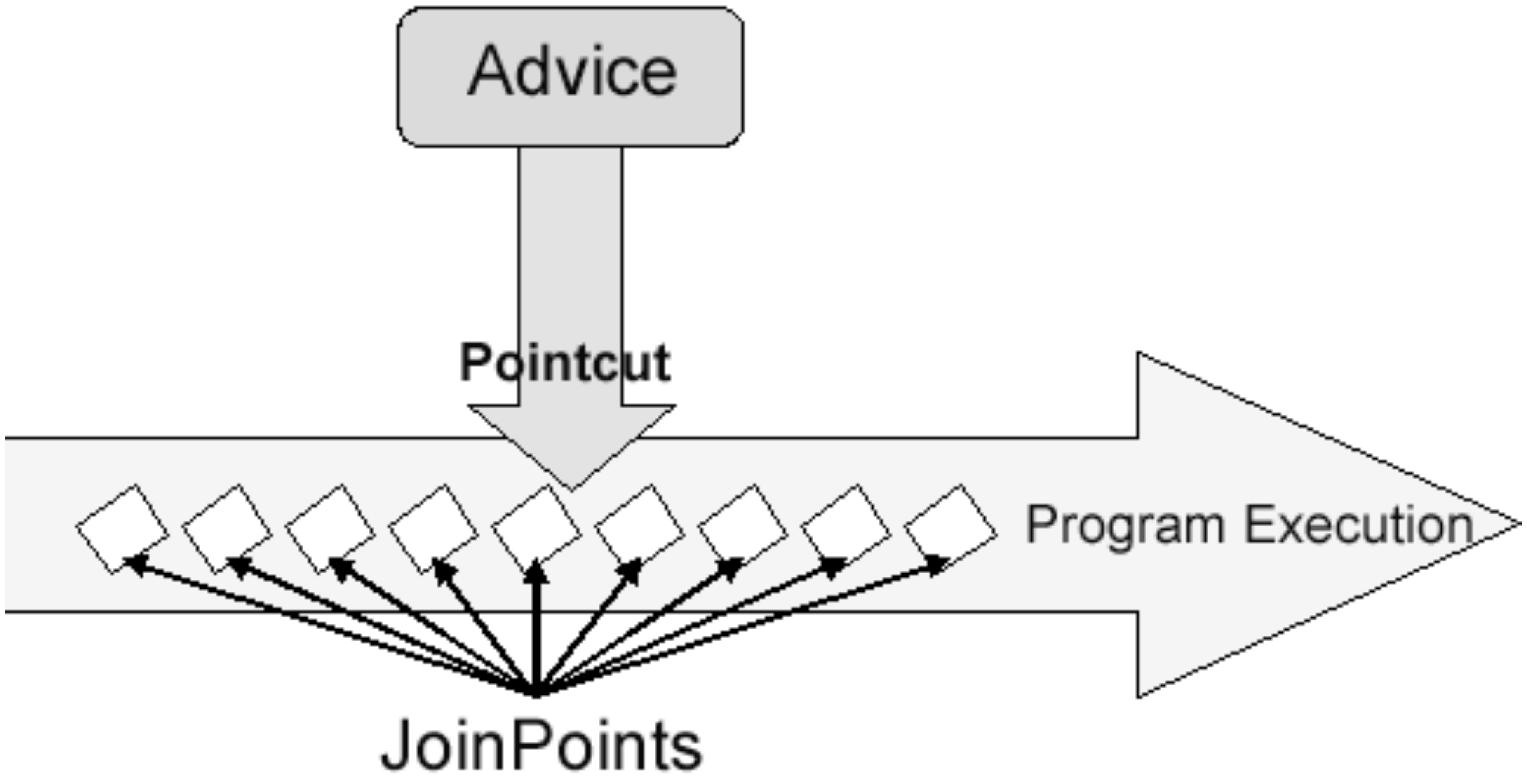
http://www.eclipse.org/aspectj/doc/next/progguide/language.html

In the pointcut-advice (PA) mechanism for aspect-oriented programming, as embodied in AspectJ and others, cross- cutting behavior is defined by means of pointcuts and advices. Points during execution at which advices may be executed are called (dynamic) join points. A pointcut identifies a set of join points, and an advice is the action to be taken at a join point matched by a pointcut. An aspect is a module that encompasses a number of pointcuts and advices. In AspectJ, the decision of whether or not to use an aspect within a program is done at build time; if so, the aspect has global scope, *i.e.* it *sees* all join points of the program execution. Restricting the scope of an aspect can be done by introducing conditions in the pointcut definitions.

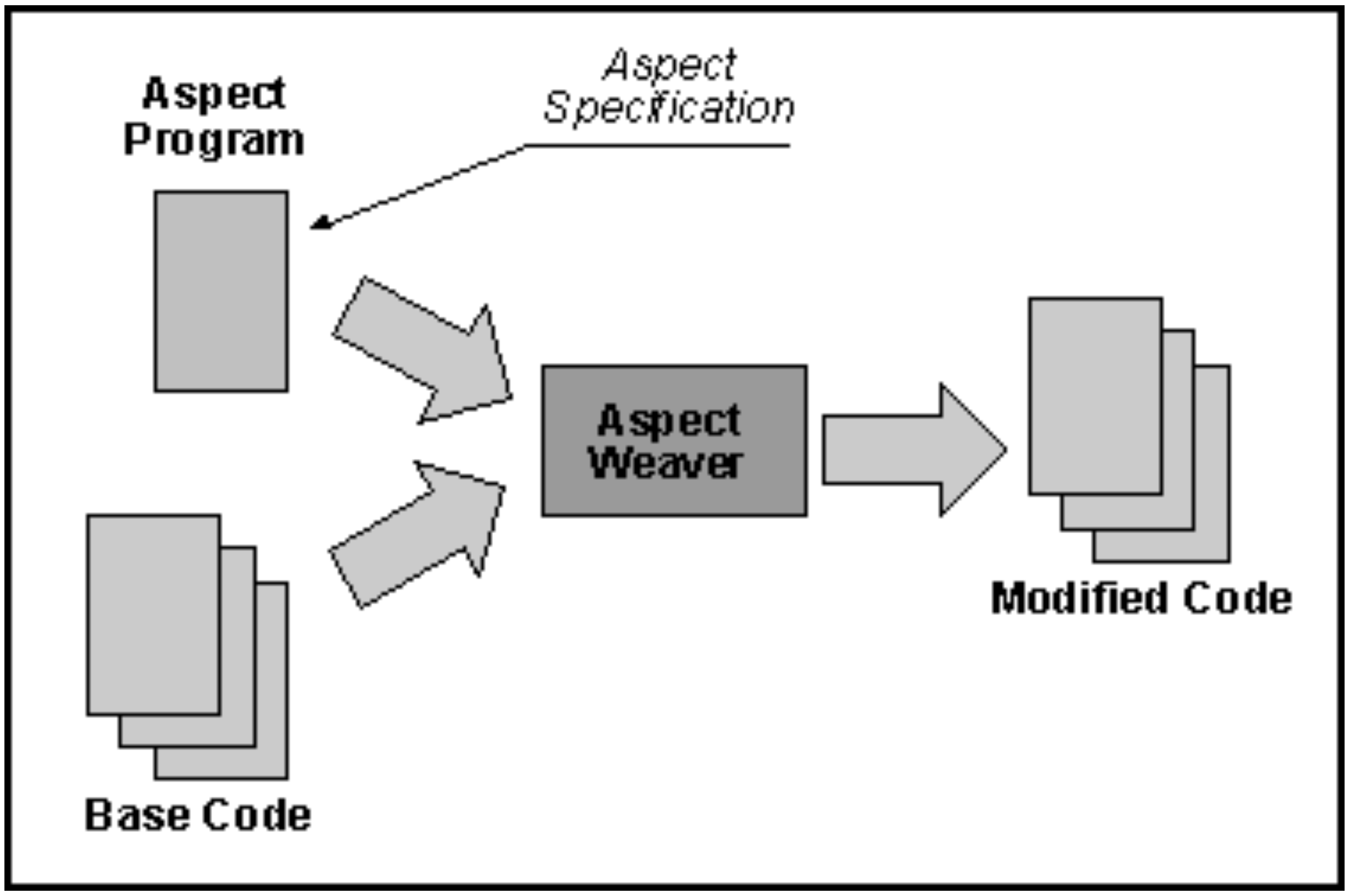
AOP



AOP



AOP


Aspect Program
Aspect Specification
Aspect Weaver
Modified Code
Base Code

AOP Example

```
public class HelloWorld {

   public static void say(String message) {
      System.out.println(message);
   }

   public static void sayToPerson(String message, String name) {
      System.out.println(name + ", " + message);
   }
}
```

AOP Example

```
public aspect Example {
   pointcut callSaySMessage() :
              call(public static void HelloWorld.say*(..));
   before() : callSayMessage() {
      System.out.println("Good day!");
   }
   after() : callSayMessage() {
      System.out.println("Thank you!");
   }
}
```

Feature Analysis AOP

```
public aspect FeatureAnalysis {
   pointcut callMessage() :
            call(public * com.mycompany..*.*(..));

   before() : callMessage() {
      ... save feature information ...
   }

}
```

Feature Analysis AOP

```
public aspect FeatureAnalysis {
   pointcut executeMessage() :
            execute(public * com.mycompany..*.*(..));

   before() : executeMessage() {
      … save feature information …
   }

}
```

So what's the difference between these join points? Well, there are a number of differences:

Firstly, the lexical pointcut declarations `within` and `withincode` match differently. At a call join point, the enclosing code is that of the call site. This means that `call(void m()) && withincode(void m())` will only capture directly recursive calls, for example. At an execution join point, however, the program is already executing the method, so the enclosing code is the method itself: `execution(void m()) && withincode(void m())` is the same as `execution(void m())`.

Secondly, the call join point does not capture super calls to non-static methods. This is because such super calls are different in Java, since they don't behave via dynamic dispatch like other calls to non-static methods.

The rule of thumb is that if you want to pick a join point that runs when an actual piece of code runs (as is often the case for tracing), use `execution`, but if you want to pick one that runs when a particular *signature* is called (as is often the case for production aspects), use `call`.

AspectJ pointcuts

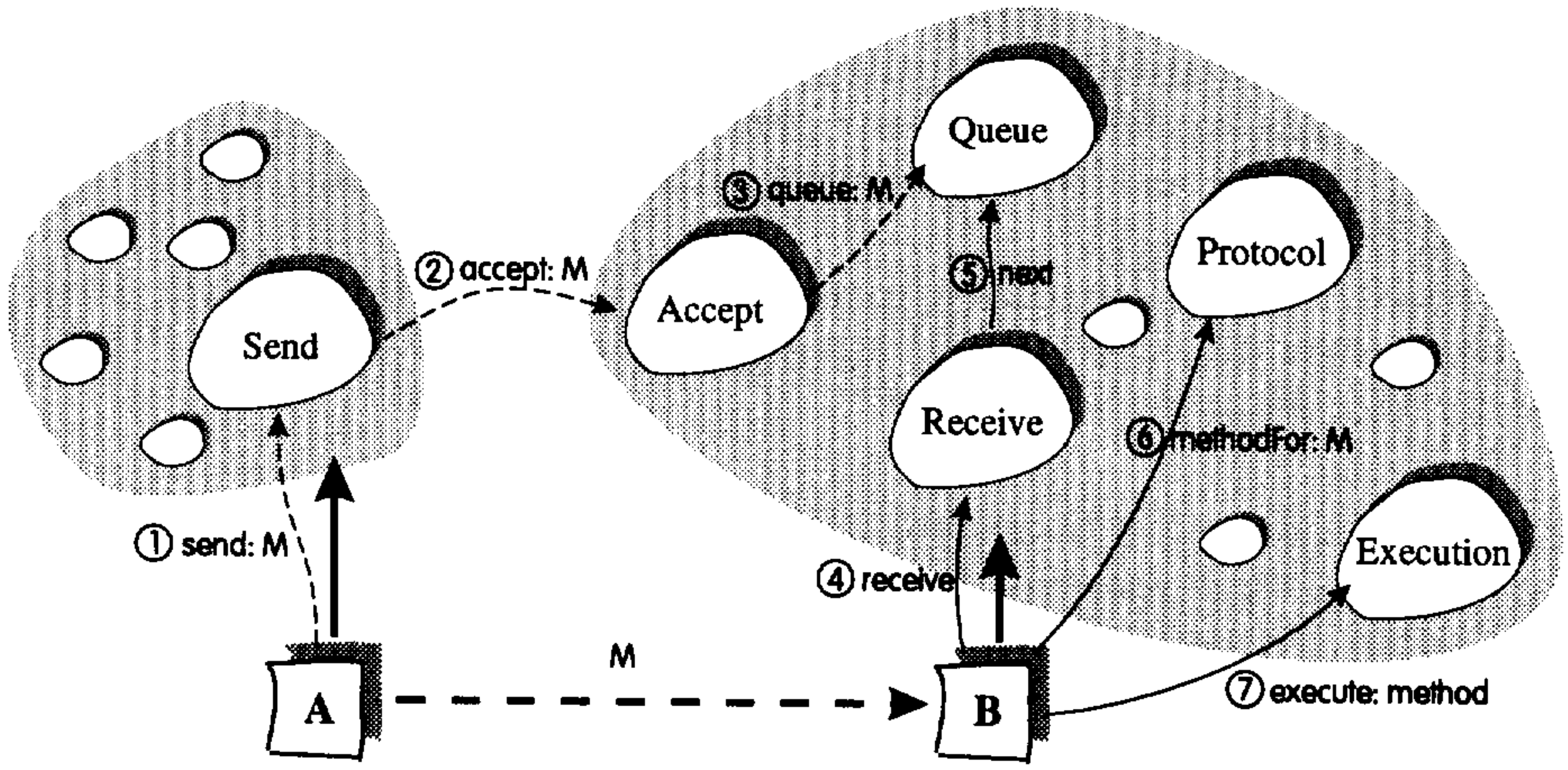
call(MethodPattern)
execution(MethodPattern)
get(FieldPattern)
set(FieldPattern)
call(ConstructorPattern)
execution(ConstructorPattern)
initialization(ConstructorPattern)
preinitialization(ConstructorPattern)
staticinitialization(TypePattern)
handler(TypePattern)
adviceexecution()
within(TypePattern)

withincode(MethodPattern)
withincode(ConstructorPattern)
cflow(Pointcut)
cflowbelow(Pointcut)
this(Type or Id)
target(Type or Id)
args(Type or Id, ...)
PointcutId(TypePattern or Id, ...
if(BooleanExpression)
! Pointcut
Pointcut0 && Pointcut1
Pointcut0 || Pointcut1

This is just and example for AspectJ, there are many other aspect languages with many different pointcuts with different objectives.

Operational Decomposition

McAffer - CodA - Meta-level Programming with CodA - ECOOP 1995



Operational Decomposition

Iguana C++, IguanaJ

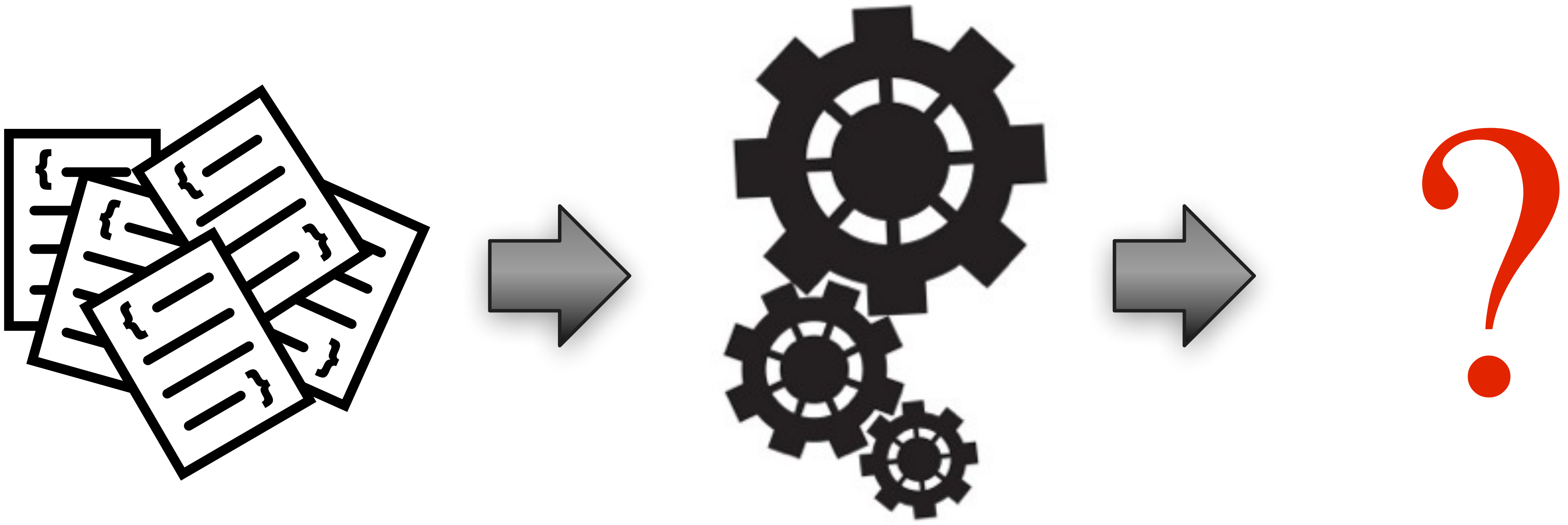
Bifröst

AOP

EAOP

AspectJ tracematches

Sub-method Feature Analysis



We need to find which statements belong to which feature.

Sub-method Feature Analysis

Bytecode Instrumentation

Bytecode Instrumentation

Smalltalk

Example: Number>>asInteger

UNIVERSITÄT BERN

- Smalltalk code:

```
Number>>asInteger
   "Answer an Integer nearest
   the receiver toward zero."

   ^self truncated
```

- Symbolic Bytecode

```
9 <70> self
10 <D0> send: truncated
11 <7C> returnTop
```

Example: Step by Step

- `9 <70> self`
  - The receiver (self) is pushed on the stack
- `10 <D0> send: truncated`
  - Bytecode 208: send litereral selector 1
  - Get the selector from the first literal
  - start message lookup in the class of the object that is on top of the stack
  - result is pushed on the stack
- `11 <7C> returnTop`
  - return the object on top of the stack to the calling method

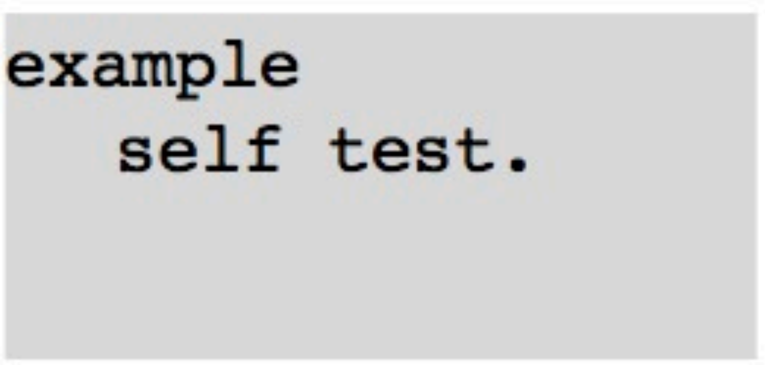
ByteSurgeon

- Library for bytecode transformation in Smalltalk
- Full flexibility of Smalltalk Runtime
- Provides high-level API
- For Pharo, but portable

- Runtime transformation needed for
  - Adaptation of running systems
  - Tracing / debugging
  - New language features (MOP, AOP)

Example: Logging

- Goal: logging message send.
- First way: Just edit the text:

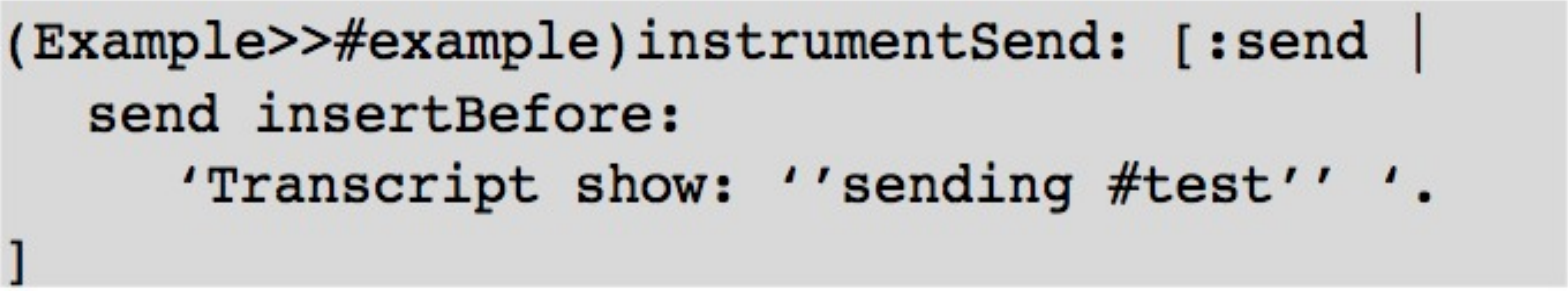


```
example
   self test.
```

```
example
   Transcript show: ‘sending #test’.
   self test.
```

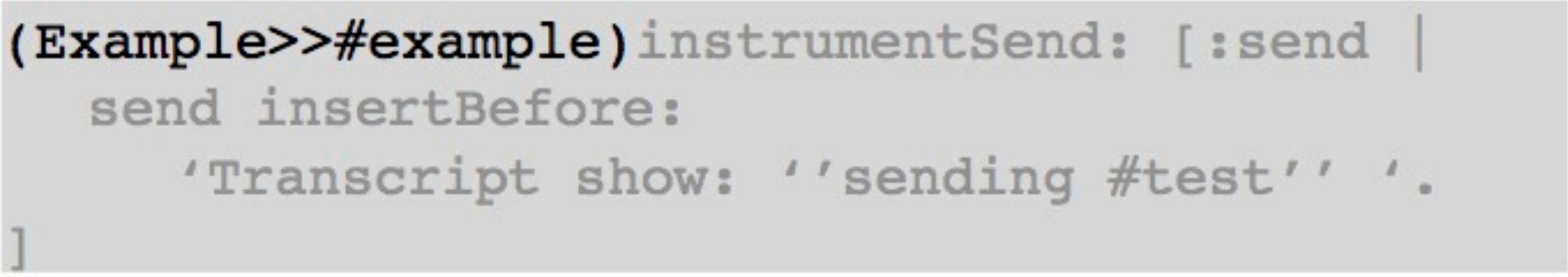
Logging with ByteSurgeon

- Goal: Change the method without changing program text
- Example:

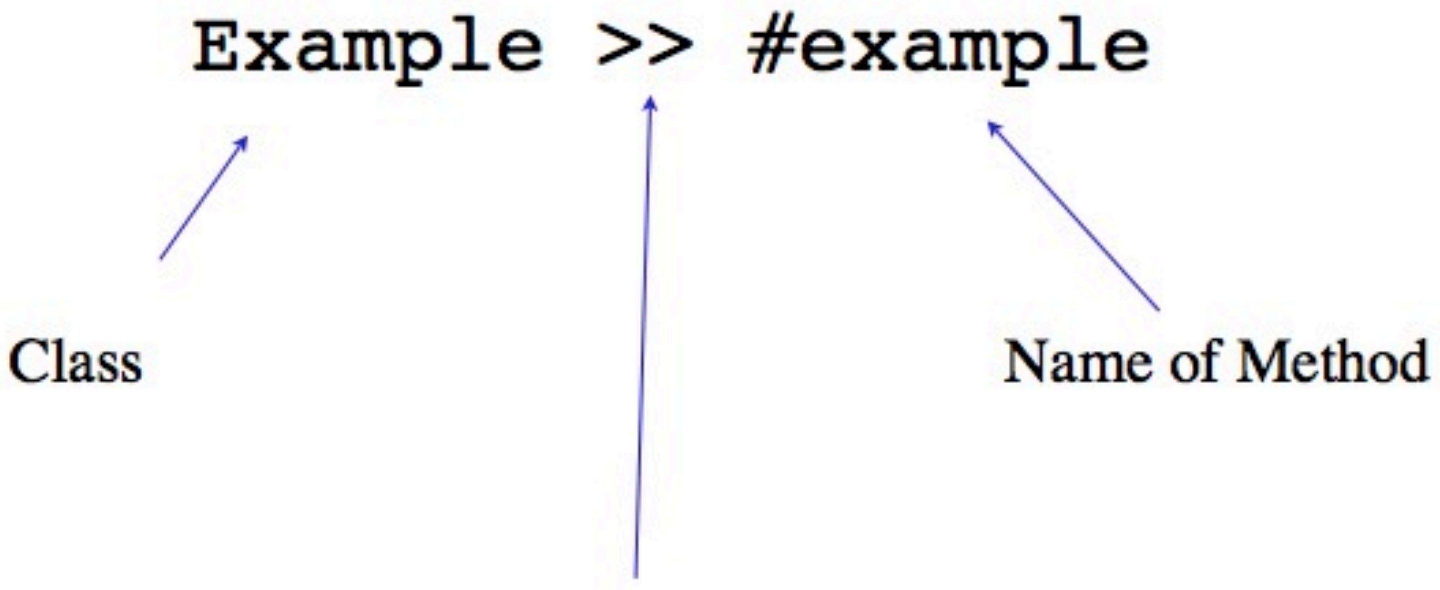


```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      ‘Transcript show: ‘’sending #test’’ ‘.
]
```

Logging: Step by Step



```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      ‘Transcript show: ‘’sending #test’’ ‘.
]
```



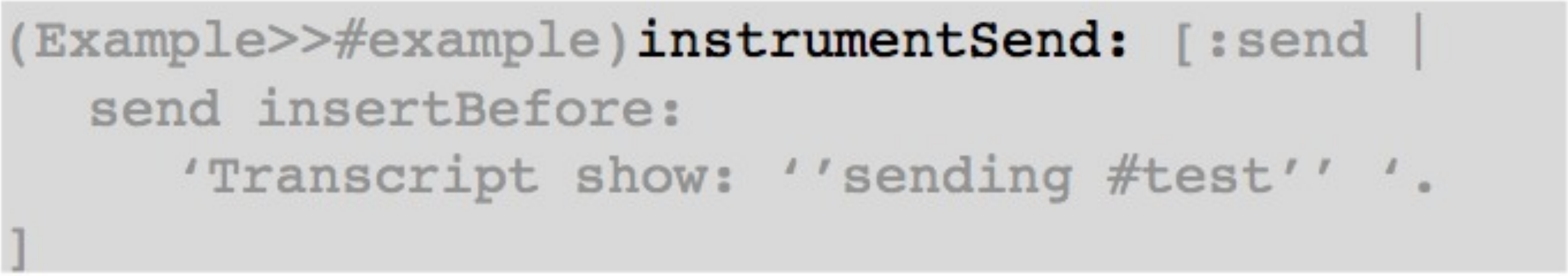
`Example >> #example`

Class

Name of Method

\>>: - takes a name of a method
- returns the CompiledMethod object

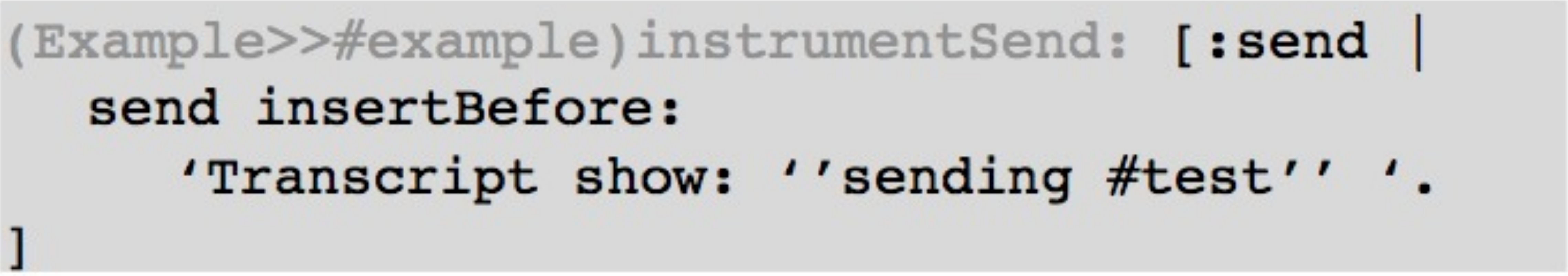
Logging: Step by Step



```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      ‘Transcript show: ‘’sending #test’’ ‘.
]
```

- instrumentSend:
  - takes a block as an argument
  - evaluates it for all send bytecodes

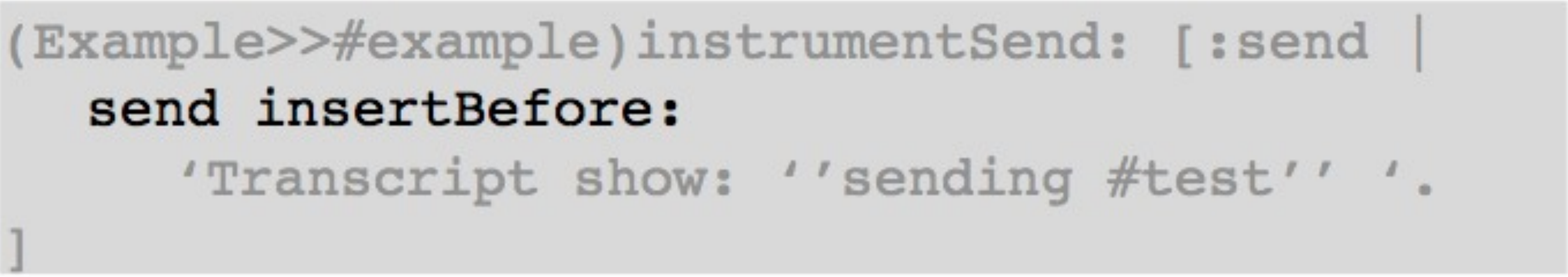
Logging: Step by Step



```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      ‘Transcript show: ‘’sending #test’’ ‘.
]
```

- The block has one parameter: send
- It is executed for each send bytecode in the method

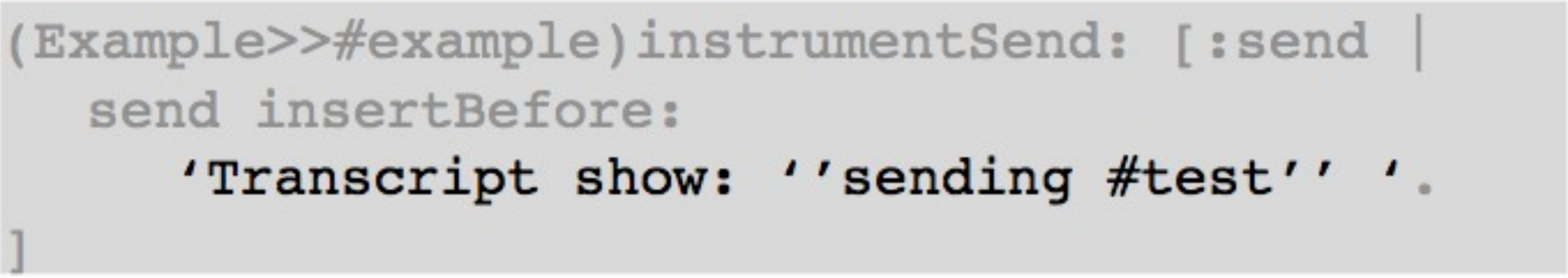
Logging: Step by Step



```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      ‘Transcript show: ‘’sending #test’’ ‘.
]
```

- Objects describing bytecode understand how to insert code
  - insertBefor
  - insertAfter
  - replace

Logging: Step by Step

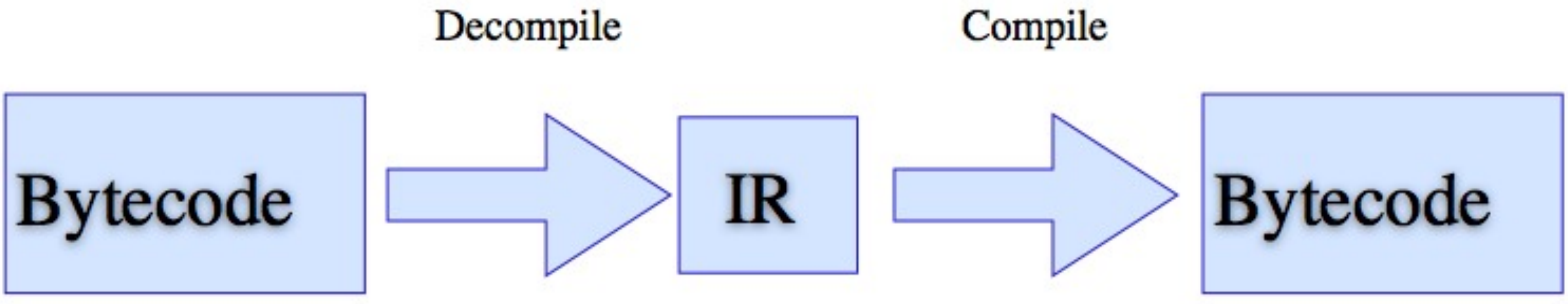


```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      ‘Transcript show: ‘’sending #test’’ ‘.
]
```

- The code to be inserted.
- Double quoting for string inside string
  - *Transcript show: ’sending #test’*

Inside ByteSurgeon

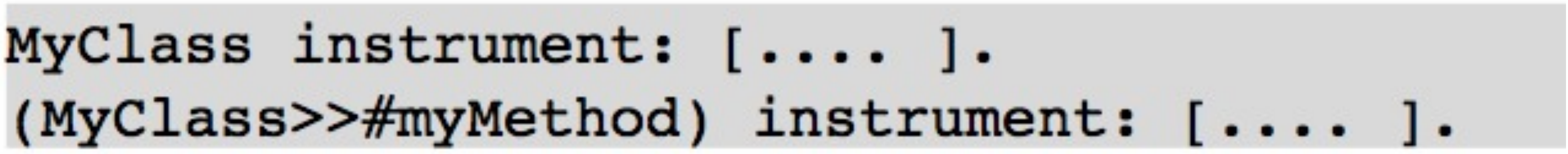
- Uses IRBuilder internally



- Transformation (Code inlining) done on IR

ByteSurgeon Usage

- On Methods or Classes:



```
MyClass instrument: [.... ].
(MyClass>>#myMethod) instrument: [.... ].
```

- Different instrument methods:
  - instrument:
  - instrumentSend:
  - instrumentTempVarRead:
  - instrumentTempVarStore:
  - instrumentTempVarAccess:
  - same for InstVar

Advanced ByteSurgeon

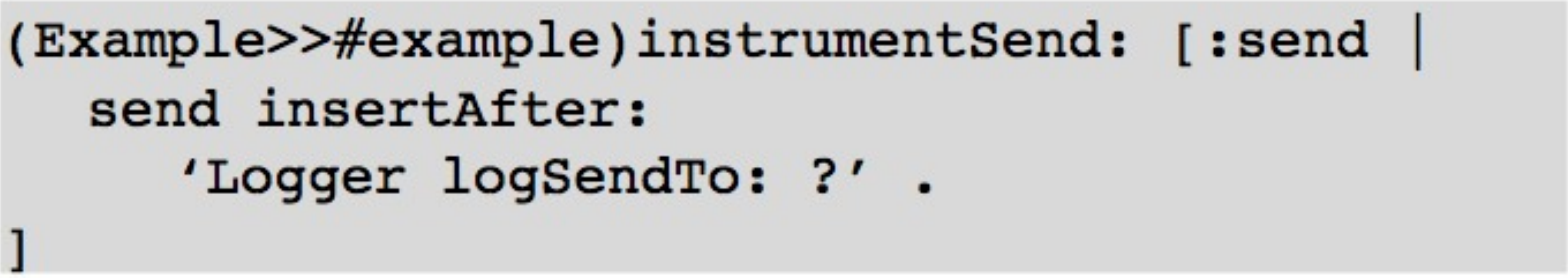
- Goal: extend a send with after logging

```
example
   self test.
```

```
example
      self test.
      Logger logSendTo: self.
```

Advanced ByteSurgeon

- With ByteSurgeon, something like:

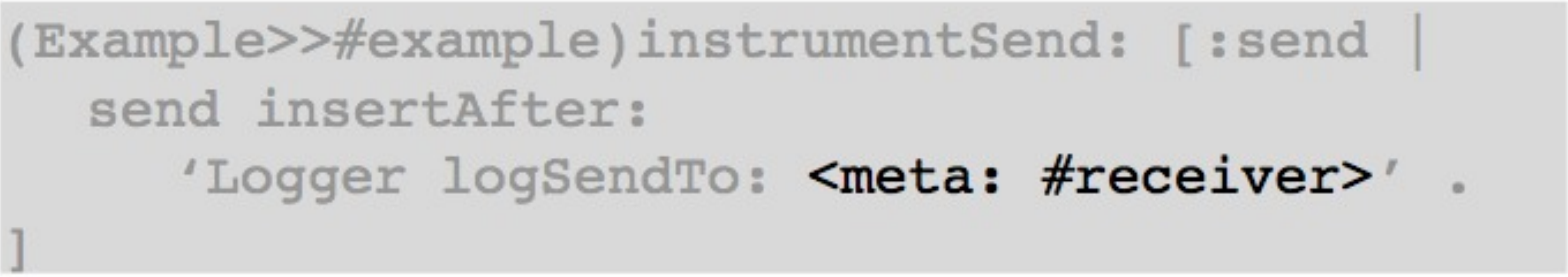


```
(Example>>#example)instrumentSend: [:send |
   send insertAfter:
      'Logger logSendTo: ?' .
]
```

- How can we access the receiver of the send?
- Solution: Metavariable

Advanced ByteSurgeon

- With Bytesurgeon, something like:



```
(Example>>#example)instrumentSend: [:send |
   send insertAfter:
      ‘Logger logSendTo: <meta: #receiver>’ .
]
```

- How can we access the receiver of the send?
- Solution: Metavariable

Bytecode Instrumentation

Java

www.javassist.org
http://commons.apache.org/bcel/
http://asm.objectweb.org/

Bytecode Manipulation

- Java
  - Javassist
    - *reflection*
    - *RMI*
  - BCEL
    - *Decompiling, Obfuscation, and Refactoring*
    - *AspectJ*
    - *FindBugs*
  - ASM
    - *Groovy*
    - *AspectWerkz*

www.javassist.org
http://commons.apache.org/bcel/
http://asm.objectweb.org/

Javassist

```
class Point {
   int x, y;
   void move(int dx, int dy) { x += dx; y += dy; }
}
```

Javassist

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
CtMethod m = cc.getDeclaredMethod("move");
m.insertBefore("{ System.out.println($1);
System.out.println($2); }");
cc.writeFile();
```

Javassist

```
class Point {
   int x, y;
   void move(int dx, int dy) {
      { System.out.println(dx); System.out.println(dy); }
      x += dx; y += dy;
   }
}
```

Javassist - Edit Body

```
CtMethod cm = ... ;
cm.instrument(
   new ExprEditor() {
      public void edit(MethodCall m)
                 throws CannotCompileException
      {
         if (m.getClassName().equals("Point")
                    && m.getMethodName().equals("move"))
            m.replace("{ $1 = 0; $_ = $proceed($$); }");
      }
   });
```

searches the method body represented by `cm` and replaces all calls to `move()` in class `Point` with a block:

- `{ $1 = 0; $_ = $proceed($$); }`

Problems with Bytecode Instrumentation

- Bytecode is not a good meta model

- Lost of management infrastructure is needed
  - Hook composition
  - Synthesized elements (hooks) vs original code
  - Mapping to source elements

- Bytecode is optimized
  - e.g. no ifTrue:

Simulation

Parsing and Interpretation

- First step: *Parse bytecode*
  - enough for easy analysis, pretty printing, decompilation
- Second step: *Interpretation*
  - needed for simulation, complex analyis (e.g., profiling)
- Pharo provides frameworks for both:
  - InstructionStream/InstructionClient (parsing)
  - ContextPart (Interpretation)

The InstructionStream Hierarchy

```
InstructionStream
  ContextPart
    BlockContext
    MethodContext
  Decompiler
  InstructionPrinter
  InstVarRefLocator
  BytecodeDecompiler
```

InstructionStream

- Parses the byte-encoded instructions
- State:
  - pc: program counter
  - sender: the method (bad name!)

```
Object subclass: #InstructionStream
  instanceVariableNames: 'sender pc'
  classVariableNames: 'SpecialConstants'
  poolDictionaries: ''
  category: 'Kernel-Methods'
```

Usage

- Generate an instance:

```
instrStream := InstructionStream on: aMethod
```

- Now we can step through the bytecode with:

```
instrStream interpretNextInstructionFor: client
```

- Calls methods on a client object for the type of bytecode, e.g.
  - `pushReceiver`
  - `pushConstant: value`
  - `pushReceiverVariable: offset`

InstructionClient

- Abstract superclass
  - Defines empty methods for all methods that InstructionStream calls on a client
- For convenience:
  - Clients don’t need to inherit from this class

```
Object subclass: #InstructionClient
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Methods'
```

Example: A test

```
InstructionClientTest>>testInstructions
   "just interpret all of methods of Object"
   | methods client scanner|

   methods := Object methodDict values.
   client := InstructionClient new.

   methods do: [:method |
      scanner := (InstructionStream on: method).
      [scanner pc <= method endPC] whileTrue: [
            self shouldnt:
                  [scanner interpretNextInstructionFor: client]
               raise: Error.
      ].
   ].
```

Example: Printing Bytecode

> InstructionPrinter:
  - Print the bytecodes as human readable text

> Example:
  - print the bytecode of *`Number`*`>>asInteger`:

```
String streamContents:
   [:str | (InstructionPrinter on: Number>>#asInteger)
            printInstructionsOn: str ]
```

```
'9 <70> self
10 <D0> send: truncated
11 <7C> returnTop
'
```

InstructionPrinter

- Class Definition:

```
InstructionClient subclass: #InstructionPrinter
   instanceVariableNames: 'method scanner
                           stream indent'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Methods'
```

InstructionPrinter

> Main Loop:

```
InstructionPrinter>>printInstructionsOn: aStream
   "Append to the stream, aStream, a description
   of each bytecode in the instruction stream."
   | end |
   stream := aStream.
   scanner := InstructionStream on: method.
   end := method endPC.
   [scanner pc <= end]
      whileTrue: [scanner interpretNextInstructionFor: self]
```

InstructionPrinter

- Overwrites methods from `InstructionClient` to print the bytecodes as text
- e.g. the method for `pushReceiver`

```
InstructionPrinter>>pushReceiver
   "Print the Push Active Context's Receiver
   on Top Of Stack bytecode."

   self print: 'self'
```

Example: InstVarRefLocator

```
InstructionClient subclass: #InstVarRefLocator
   instanceVariableNames: 'bingo'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Methods'

InstVarRefLocator>>interpretNextInstructionUsing: aScanner
   bingo := false.
   aScanner interpretNextInstructionFor: self.
   ^bingo

InstVarRefLocator>>popIntoReceiverVariable: offset
   bingo := true

InstVarRefLocator>>pushReceiverVariable: offset
   bingo := true

InstVarRefLocator>>storeIntoReceiverVariable: offset
   bingo := true
```

InstVarRefLocator

- Analyse a method, answer true if it references an instance variable

```
CompiledMethod>>hasInstVarRef
   "Answer whether the receiver references an instance variable."

   | scanner end printer |

   scanner := InstructionStream on: self.
   printer := InstVarRefLocator new.
   end := self endPC.

   [scanner pc <= end] whileTrue:
      [ (printer interpretNextInstructionUsing: scanner)
         ifTrue: [^true]. ].
   ^false
```

InstVarRefLocator

- Example for a simple bytecode analyzer
- Usage:

```
aMethod hasInstVarRef
```

- (has reference to variable testSelector)

```
(TestCase>>#debug) hasInstVarRef
```

```
true
```

- (has no reference to a variable)

```
(Integer>>#+) hasInstVarRef
```

```
false
```

ContextPart: Semantics for Execution

- Sometimes we need more than parsing
  - “stepping” in the debugger
  - system simulation for profiling

```
InstructionStream subclass: #ContextPart
   instanceVariableNames: 'stackp'
   classVariableNames: 'PrimitiveFailToken QuickStep'
   poolDictionaries: ''
   category: 'Kernel-Methods'
```

Simulation

- Provides a complete Bytecode interpreter

- Run a block with the simulator:

```
(ContextPart runSimulated: [3 factorial])
```

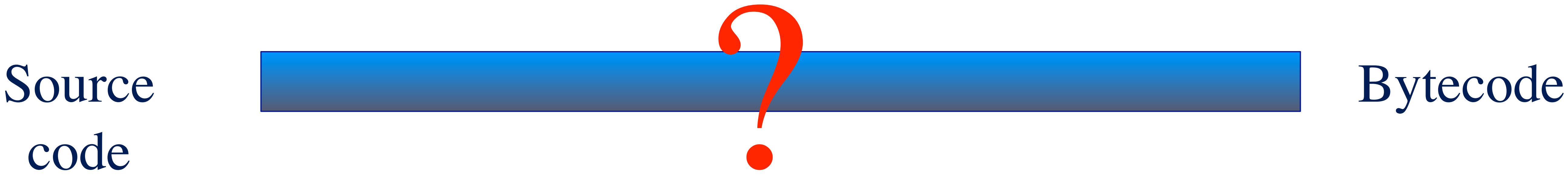
```
6
```

What is the big picture?

Source code

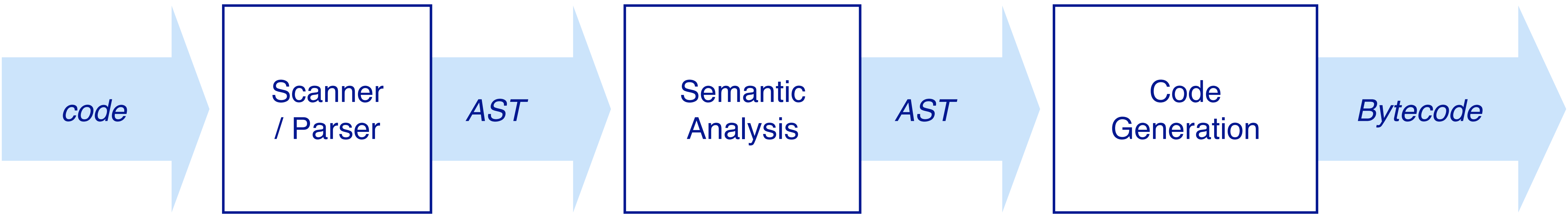
Bytecode

What is the big picture?


Source code

Bytecode

AST Instrumentation



code

Scanner / Parser

AST

Semantic Analysis

AST

Code Generation

Bytecode

Reflectivity

- Marcus Denker
  - Pharo Smalltalk
  - Geppetto 2
  - Phersephone

- Using Partial Behavioral Reflection Model
  - Reflex, Tanter etal.

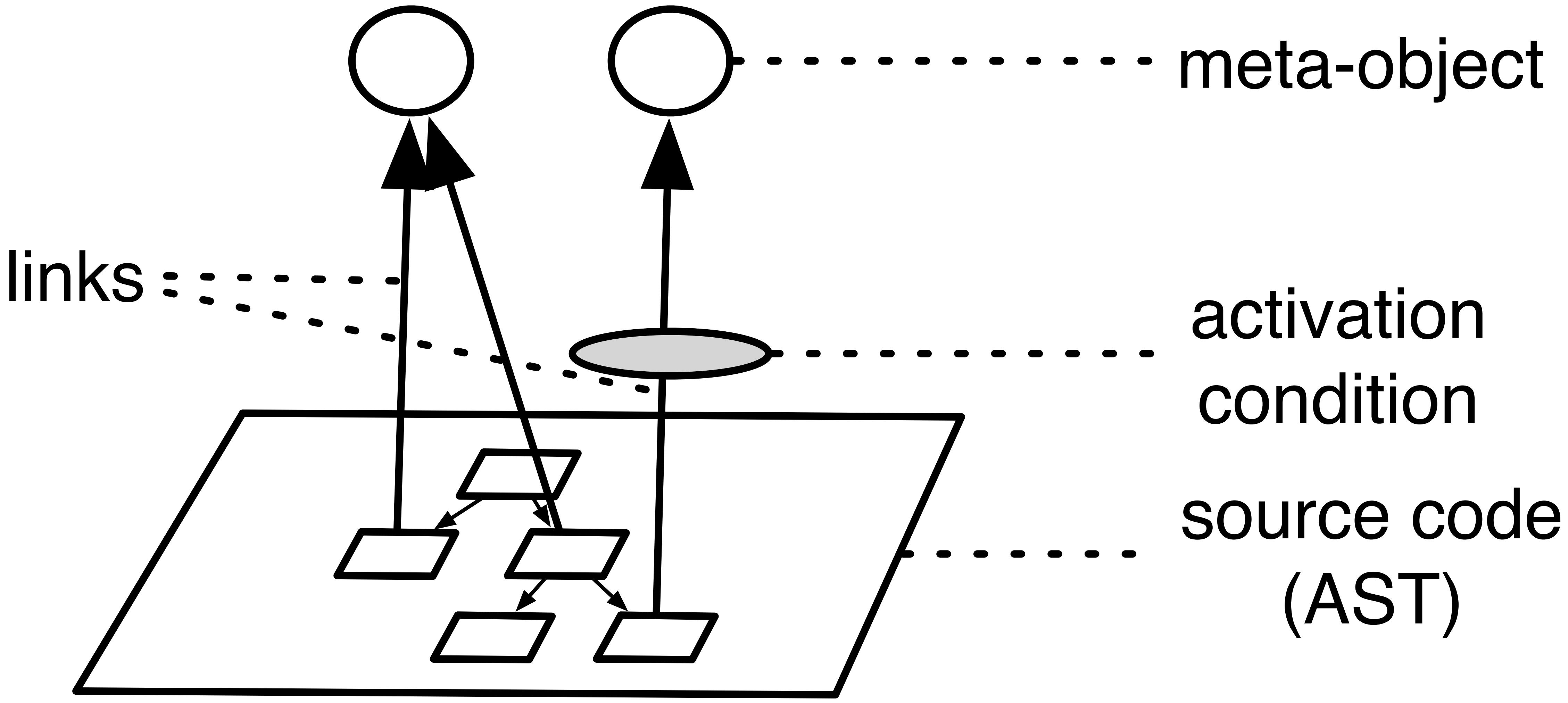
Compiler: AST

> AST: Abstract Syntax Tree

- Encodes the Syntax as a Tree
- No semantics yet!
- Uses the RB Tree:
  - *Visitors*
  - *Backward pointers in ParseNodes*
  - *Transformation (replace/add/delete)*
  - *Pattern-directed TreeRewriter*
  - *PrettyPrinter*

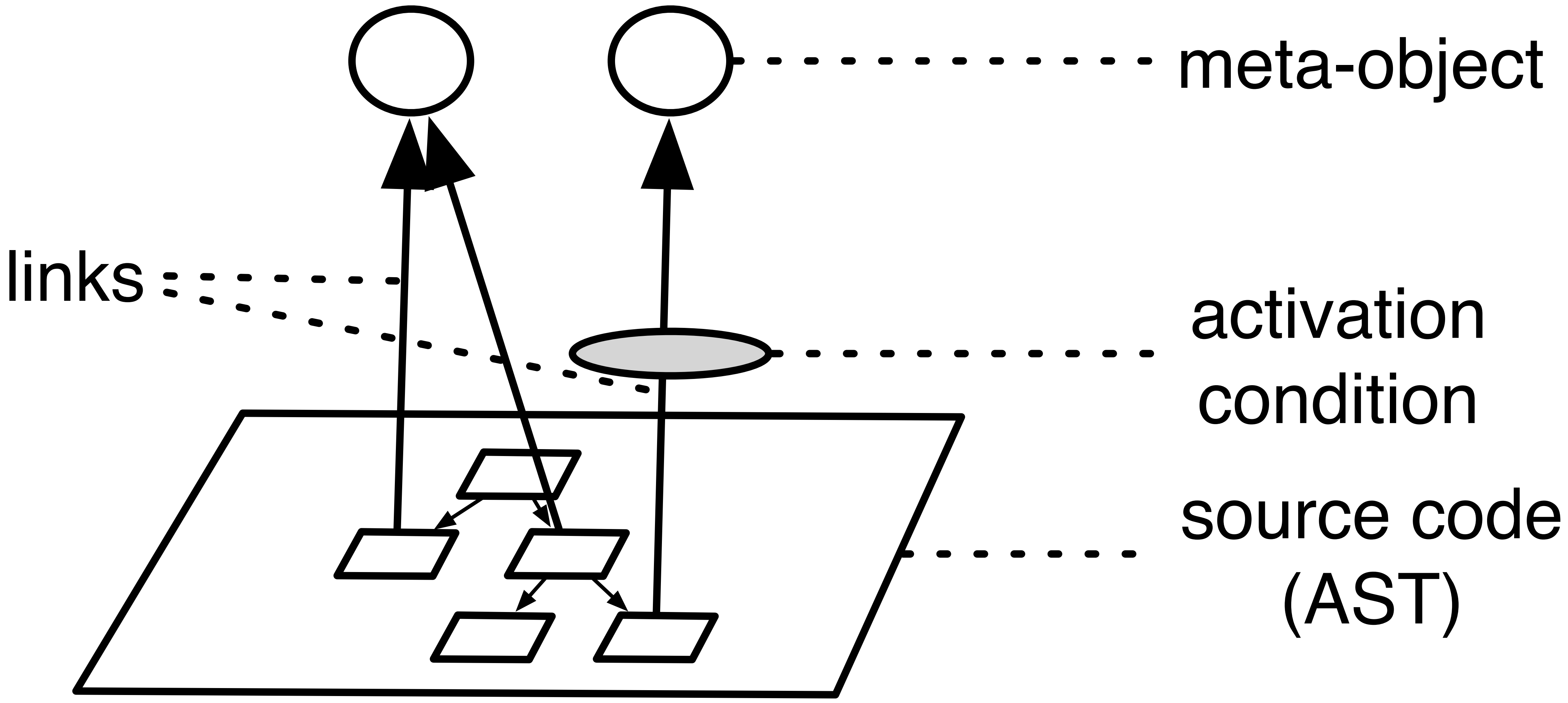
```
RBProgramNode
   RBDoItNode
   RBMethodNode
   RBReturnNode
   RBSequenceNode
   RBValueNode
      RBArrayNode
      RBAssignmentNode
      RBBlockNode
      RBCascadeNode
      RBLiteralNode
      RBMessageNode
      RBOptimizedNode
      RBVariableNode
```

Reflectivity



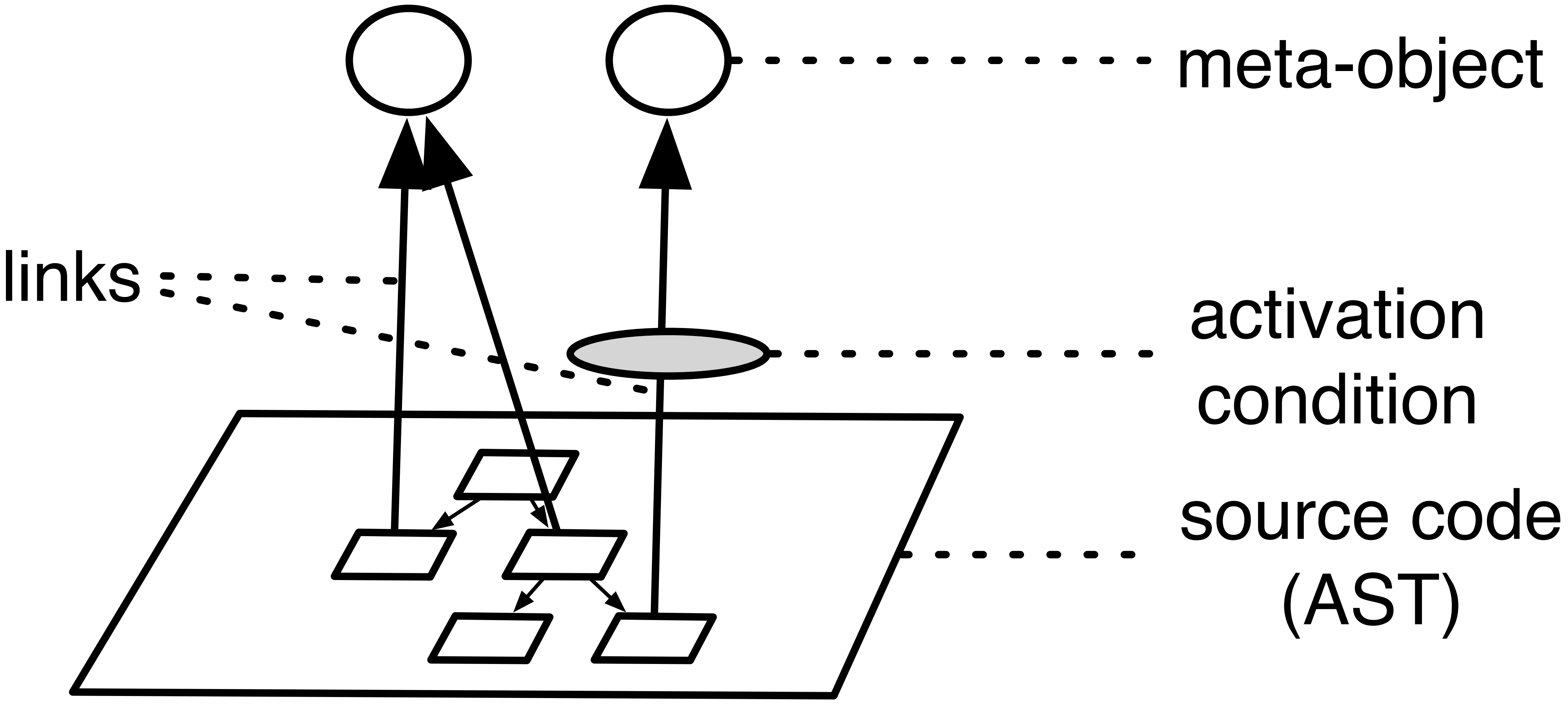
Links for AST nodes

Reflectivity



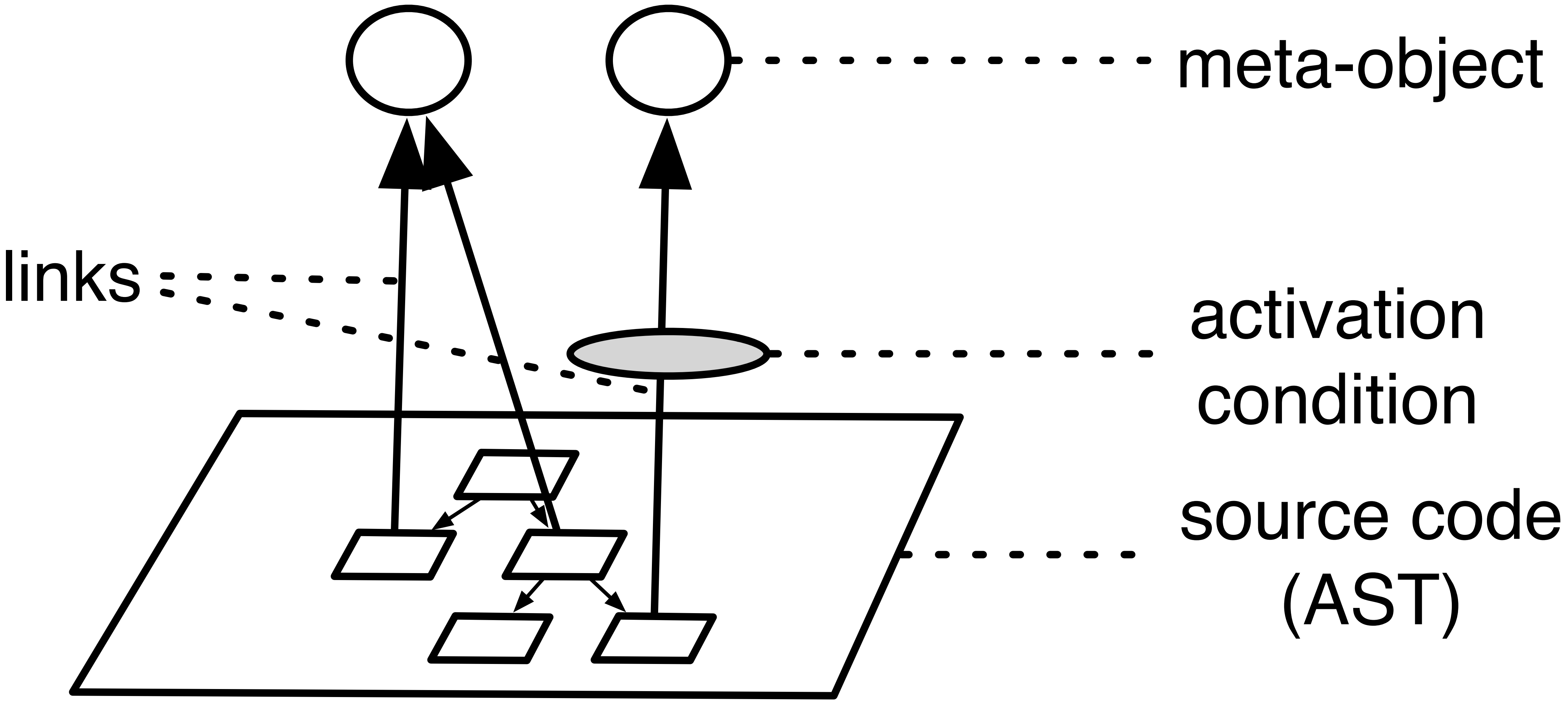
Links for AST nodes

Reflectivity



Links for AST nodes

Reflectivity



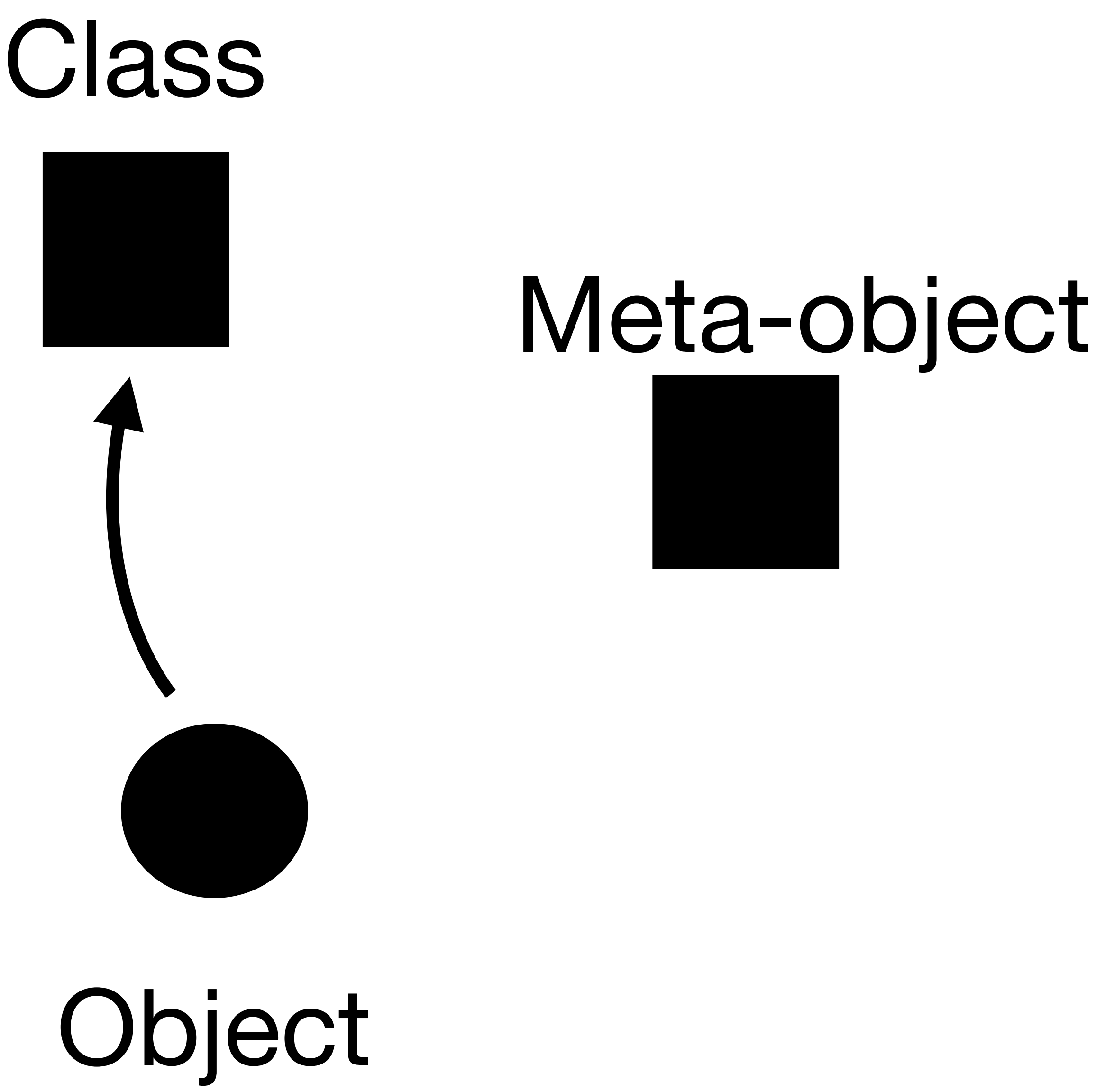
Links for AST nodes

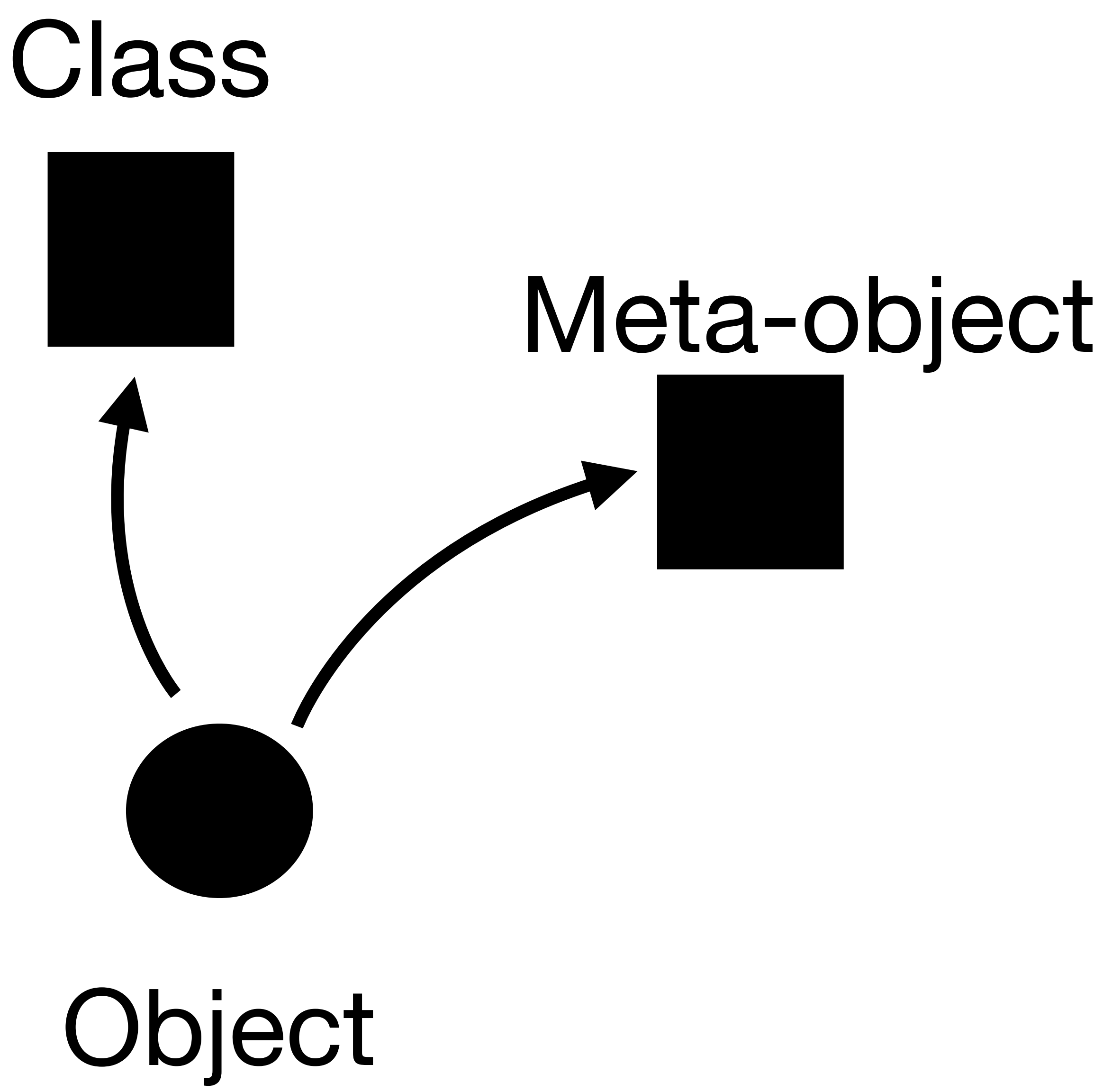
Reflective Architecture

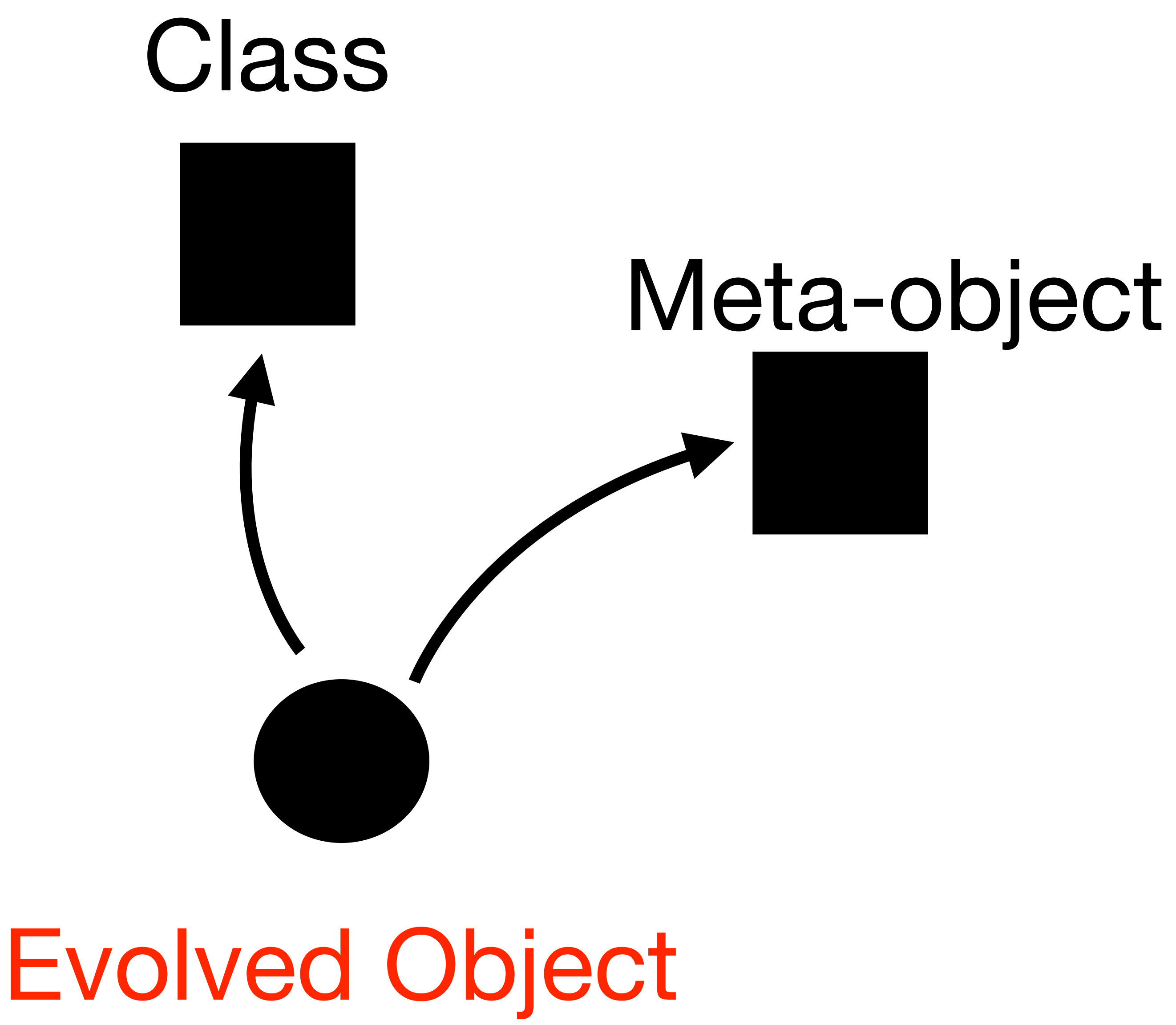
Bifröst

Bifröst

- Organize the meta-level
- Explicit meta-object
- Structural and Behavioral reflection
- Partial Reflection
- Unanticipation
- Selective Reifications
- No VM requirements


Class
Meta-object
Object


Class
Meta-object
Object


Class
Meta-object
Evolved Object

Feature Analysis

```
| aMetaObject |
aMetaObject := BFBehavioralMetaObject new.
aMetaObject
    when: (ASTExecutionEvent new)
    do: [ ... feature information gathering ...].
aMetaObject bindTo: self
```

http://scg.unibe.ch/research/bifrost

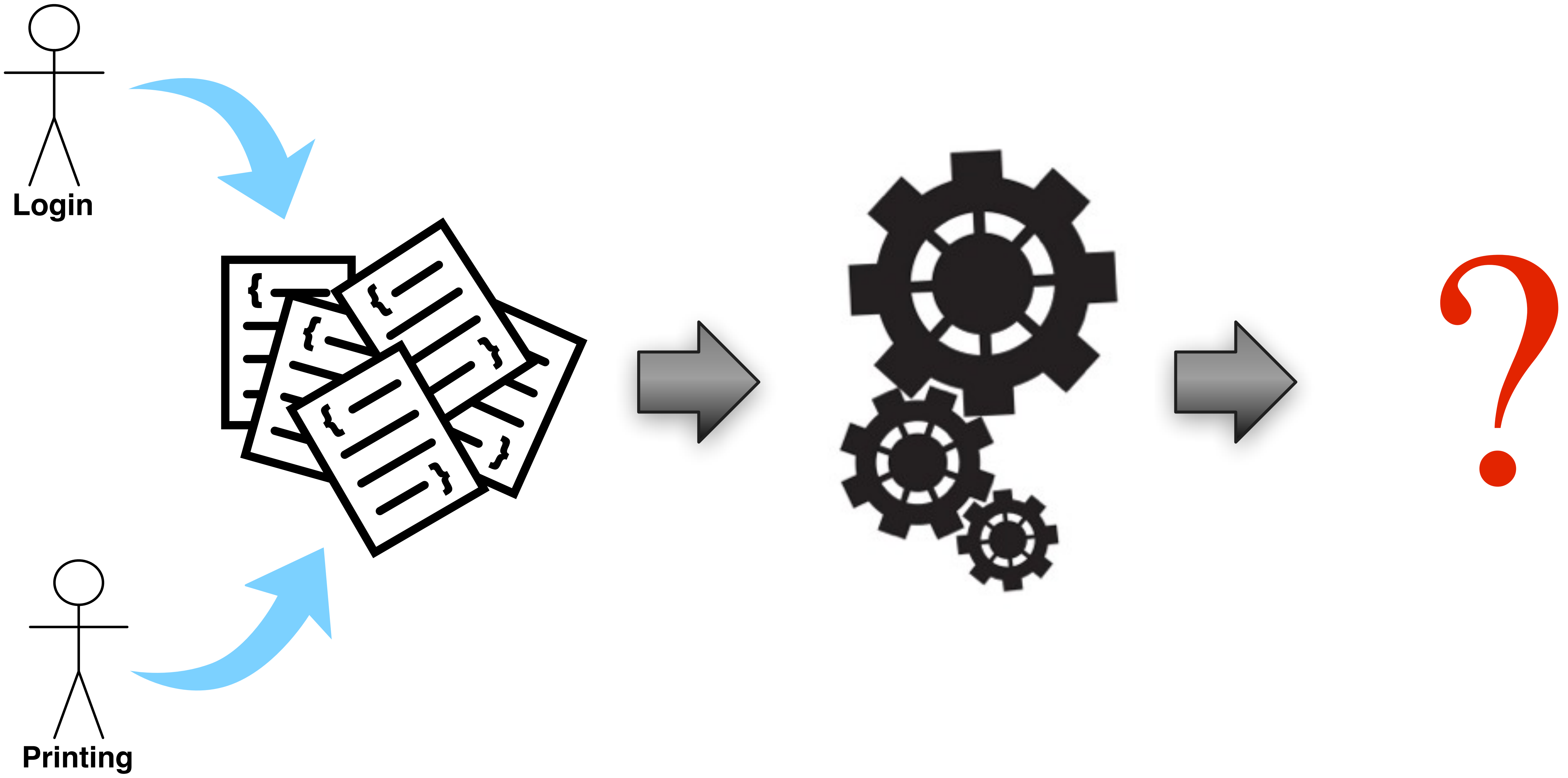
Implicit Problems

- Partial Reflection
  - We want to reflect on portions of the system
- Unanticipation
  - We want to reflect without having to anticipate where in the system
- Selective Reifications
  - We want to have runtime reifications available
- Composition
  - We want to be able to compose different analysis

Roadmap

- Motivation
- Sources of Runtime Information
- Dynamic Analysis Techniques
- **Advanced Dynamic Analysis Techniques**
- Dynamic analysis in a Reverse Engineering Context
- What can we achieve with all this?
- Conclusion

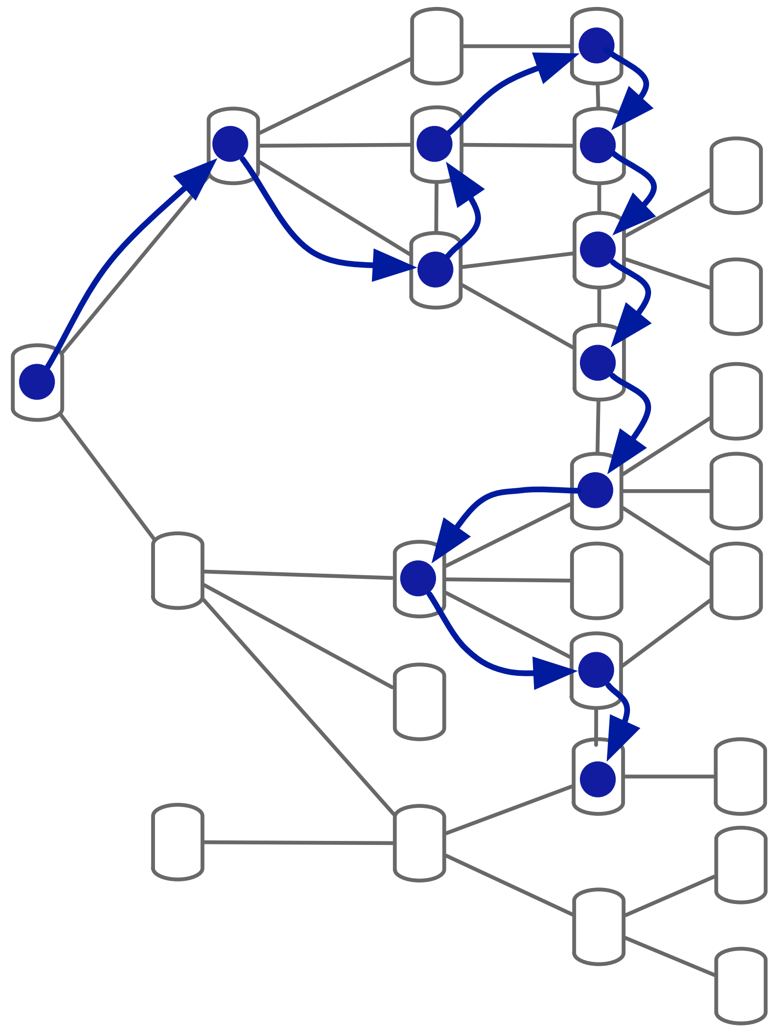
Simultaneous Feature Analysis



Simultaneous Feature Analysis

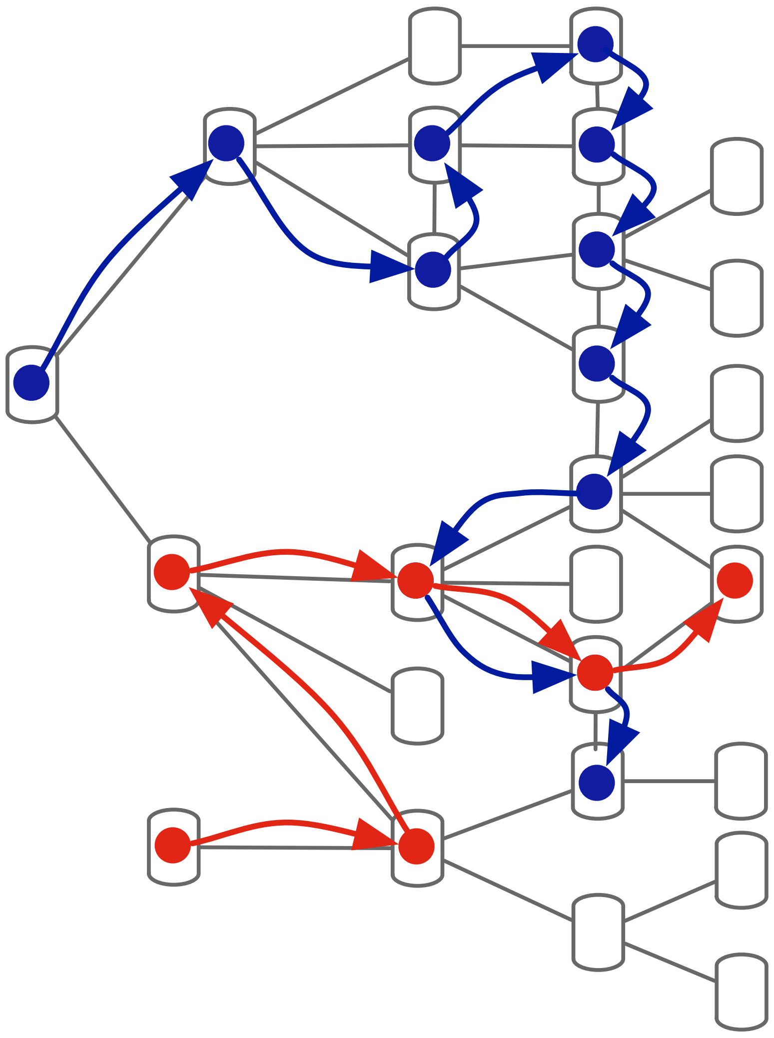
Dynamic Scope

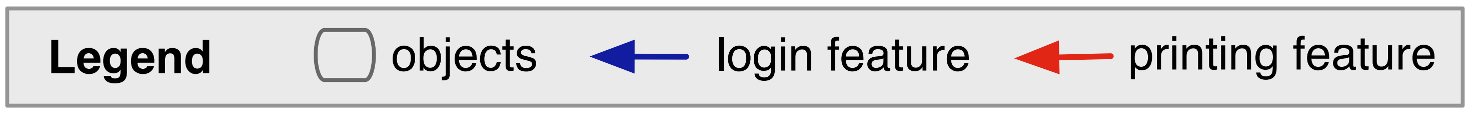
Simultaneous Feature Analysis



Legend ▢ objects ← dynamic scope

Simultaneous Feature Analysis





CaesarJ

Dynamically scoped aspects

deploy(a){block}

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development,* volume 3880 of *Lecture Notes in Computer Science,* pages 135–173. Springer-Verlag, February 2006.

whereby the aspect instance a sees all join points produced in the dynamic extent of the execution of block

Dynamically scoped aspects

- AspectScheme
- CaesarJ
- AspectS

Deployment Strategies

depl(a, **δ**⟨**c**, **d**, **f** ⟩, e)

a is an aspect

δ is the strategy

- c stack propagation function
- d object propagation function
- f joint point filter

e is an expression

Éric Tanter. Expressive Scoping of Dynamically-Deployed Aspects. In Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008), p. 168—179, ACM Press, Brussels, Belgium, April 2008.

Deployment Strategies

```
deploy[true,-,if(cars_sp.contains(jp.args(1)))](sp){
  next.process(batch);
}
```

Example from a car factory. Only some cars with a special package should get this adaptation

Scoping Strategies

Propagation and Activation Problem

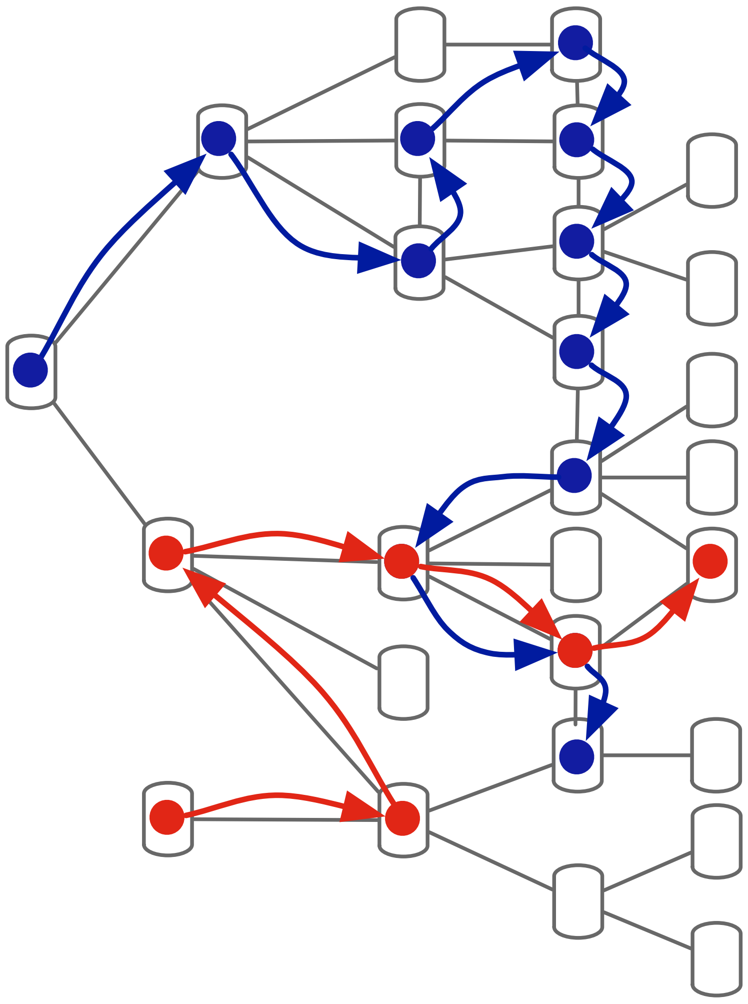
Éric Tanter. Beyond static and dynamic scope. In Proceedings of the 5th symposium on Dynamic languages, DLS '09 p. 3—14, ACM, New York, NY, USA, 2009.

Dynamic Scoping

Prisma

Dynamic Scoping

Simultaneous Feature Analysis

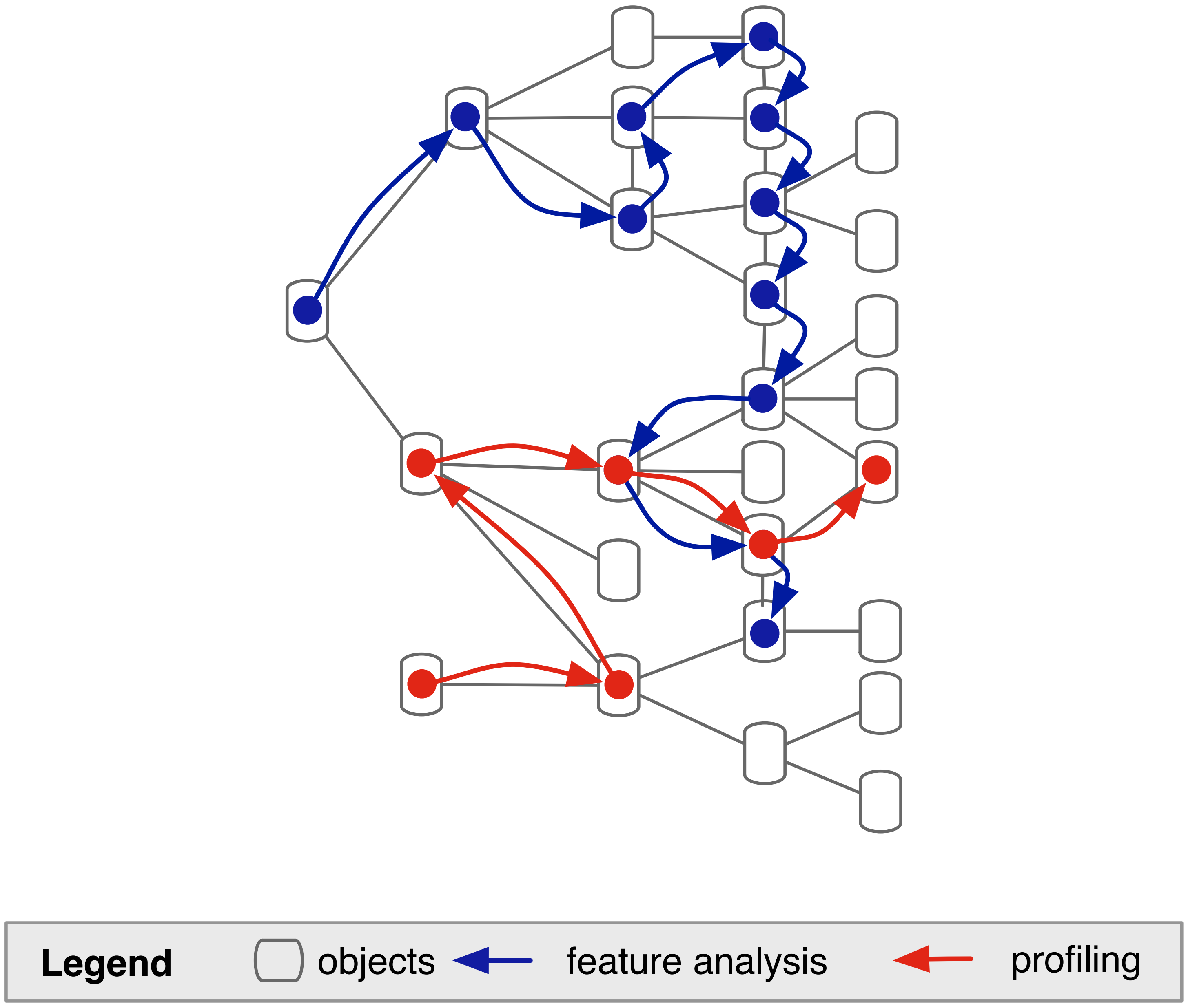


Legend ▢ objects ⟵ login feature ⟵ printing feature

Dynamic Scoping

- Prisma
  - Execution Reification
  - Reflective Architecture
  - Execution composed of meta-objects
  - Reuse of Execution
  - Execution is not tied to threads
  - Broadening of Scope
  - Dynamic change of conditions

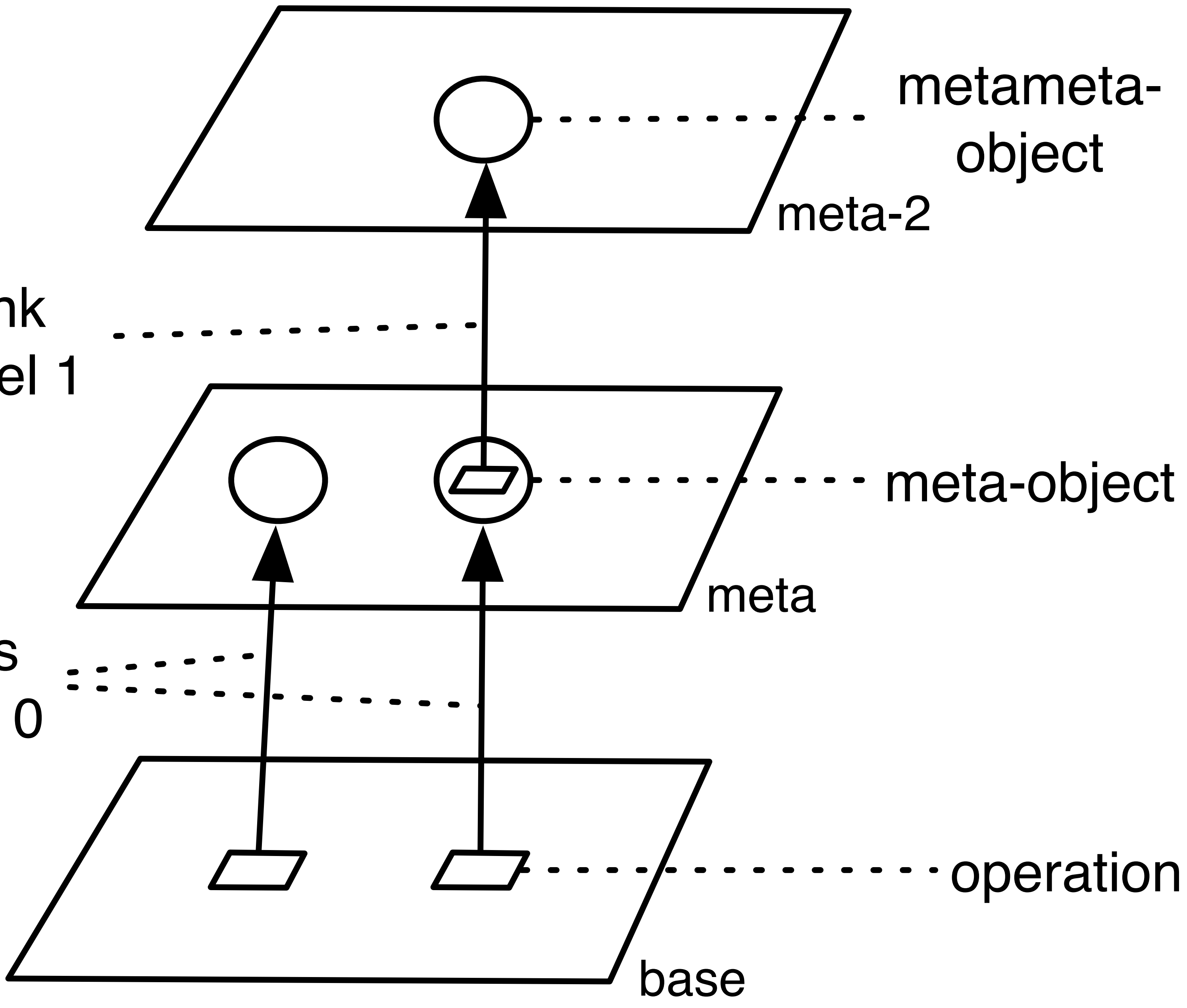
Execution levels



Legend ▭ objects ← feature analysis ← profiling

Execution levels

Denker etal. Meta Context



Execution levels

- Polymorphic Bytecode Instrumentation (PBI)
  - Dynamic dispatch amongst several, possibly independent instrumentations
  - Instrumentations are saved and indexed by a version identifier
  - Implemented over BCEL
  - JVM
  - Scala, JRuby, etc.
  - Execution levels
  - Monitoring
  - Mixin Layers
  - Promising performance

Philippe Moret, Walter Binder, and Éric Tanter. Polymorphic bytecode instrumentation. In Proceedings of the tenth international conference on Aspect-oriented software development, AOSD '11 p. 129—140, ACM, New York, NY, USA, 2011.

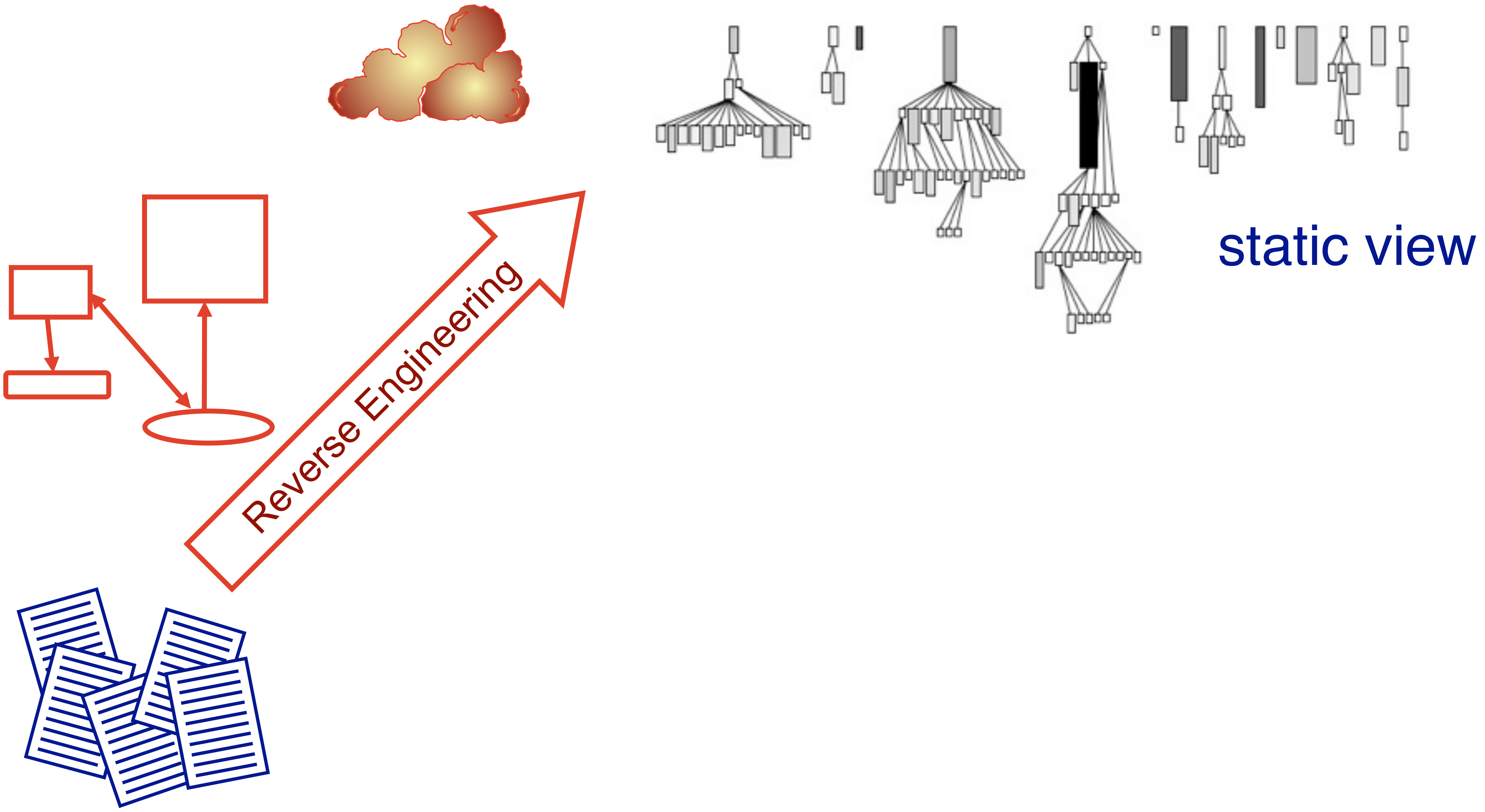
Scoping Dimensions

- **Nature of Adaptation.** A structural adaptation depicts the addition or change of a structural element, like refinements in Classboxes. A behavioral adaptation execute some action when specific runtime events are triggered.
- **Scoped Definition.** The boundaries of the scope are defined by the entry and exit points. These boundaries can be implicit or explicit.
- **Scope Information Exposure.** Some approaches allow to bind a value to a variable which is bound to the scope. This trait is particularly important to provide reusable adaptations.
- **Scope Binding.** There are two binding dimensions. The adaptation can be defined at compile time or at runtime, this is call binding time. The binding mode describes wether an adaptation can be undone/redone during execution, if so the binding mode is said to be dynamic otherwise is static.
- **Thread Locality.** The scope can be defined locally to a single thread. For example, cflow in AspectJ is by default thread local, while tracematch in AspectJ extension is by default global.

Roadmap

- Motivation
- Sources of Runtime Information
- Dynamic Analysis Techniques
- Advanced Dynamic Analysis Techniques
- **Dynamic analysis in a Reverse Engineering Context**
- What can we achieve with all this?
- Conclusion

Reverse Engineering



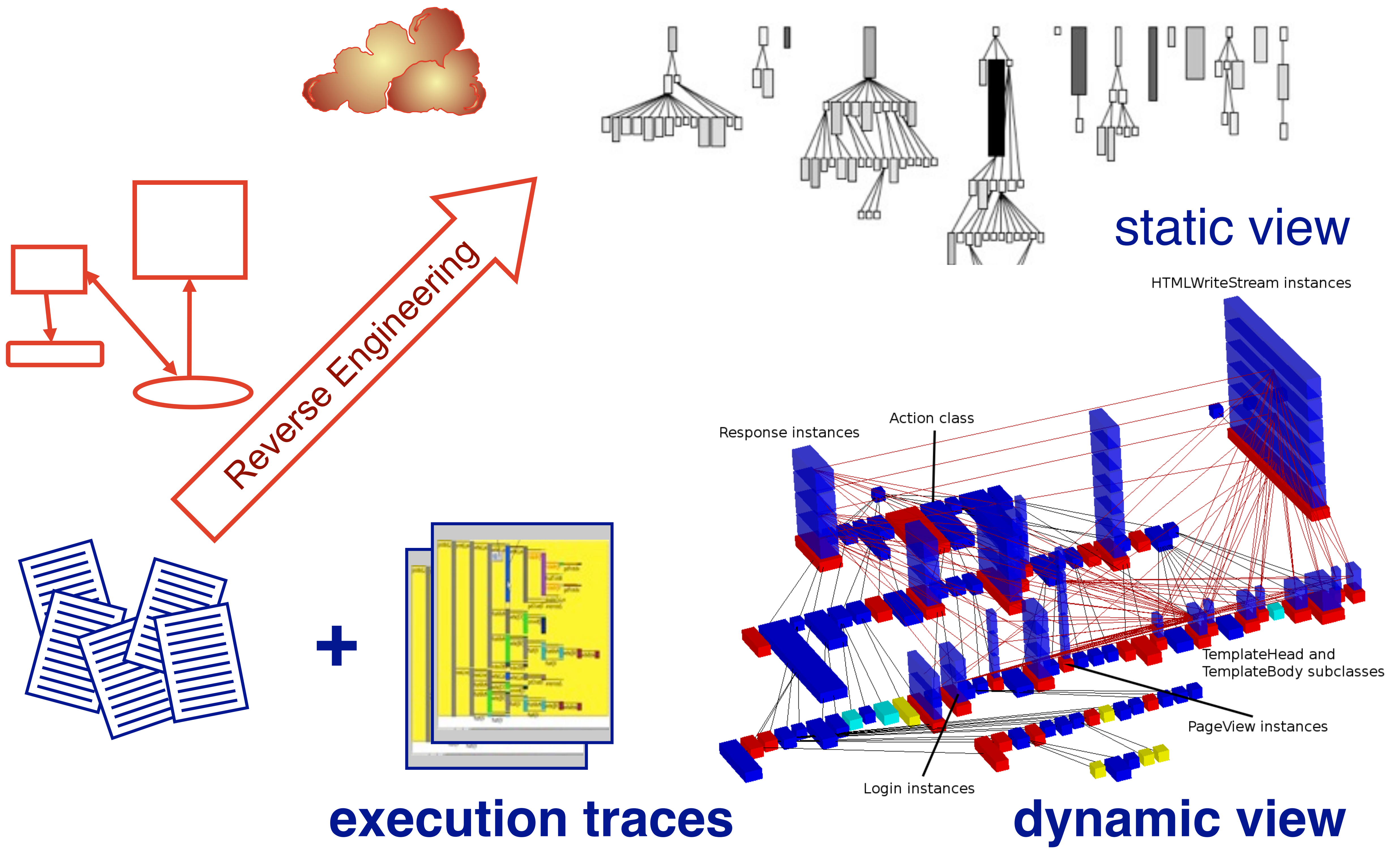
In this course you have been introduced to the concepts of reverse engineering. Reverse engineering
Reverse engineering abstracts high level abstractions that support system understanding [Chikofsky and Cross, 1990].

“Object-oriented language characteristics such as inheritance,
dynamic binding and polymorphism mean that the behavior
of a system can only be determined at runtime.” [Jerding 1996,
Demeyer2003a]

A static perspective of the system over looks semantic knowledge of the problem domain of a system. The semanic knowledge Should not be ignored. We need a way to enrich the static views with information about their intent. Which features do they Participate in at runtime? Are they specific to one part of the system, one feature, or is it general functionality that implements sone infrastructural functionality?

So lets extend our analysis by incorporating dynamic data captured while executing the features.

Reverse Engineering + Dynamic Analysis



In this course you have been introduced to the concepts of reverse engineering. Reverse engineering
Reverse engineering abstracts high level abstractions that support system understanding [Chikofsky and Cross, 1990].

“Object-oriented language characteristics such as inheritance, dynamic binding and polymorphism mean that the behavior of a system can only be determined at runtime.” [Jerding 1996, Demeyer2003a]

A static perspective of the system over looks semantic knowledge of the problem domain of a system. The semanic knowledge Should not be ignored. We need a way to enrich the static views with information about their intent. Which features do they Participate in at runtime? Are they specific to one part of the system, one feature, or is it general functionality that implements sone infrastructural functionality?

So lets extend our analysis by incorporating dynamic data captured while executing the features.

Dynamic Analysis for Program Comprehension

Post Mortem Analysis of execution traces
Metrics Based Approaches

-Frequency Analysis [Ball, Zaidman]
-Runtime Coupling Metrics based on Web mining techniques to detect key classses in a trace.
[Zaidman 2005]
-High-Level Polymetric Views of Condensed Run-Time Information [Ducasse, Lanza and Bertoulli 2004]

-Query-based approaches
Recoverind high-level views from runtime data
[Richner and Ducasse 1999]

.They define an execution scenario to maximize coverage of the system and ‘preciseness’. To execute all the features.

Frequency analysis - small number of methods are responsible for a large amount of the trace. They focus on call relationships between methods to learn something about a system.

Coupling metrics:

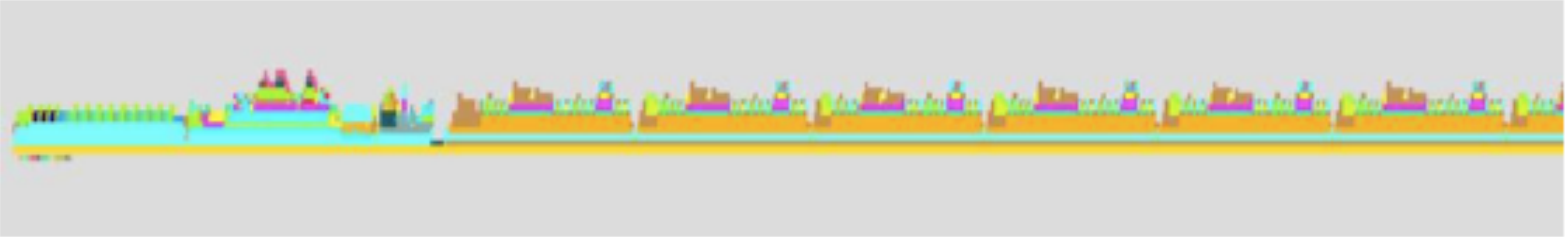
Runtime metrics

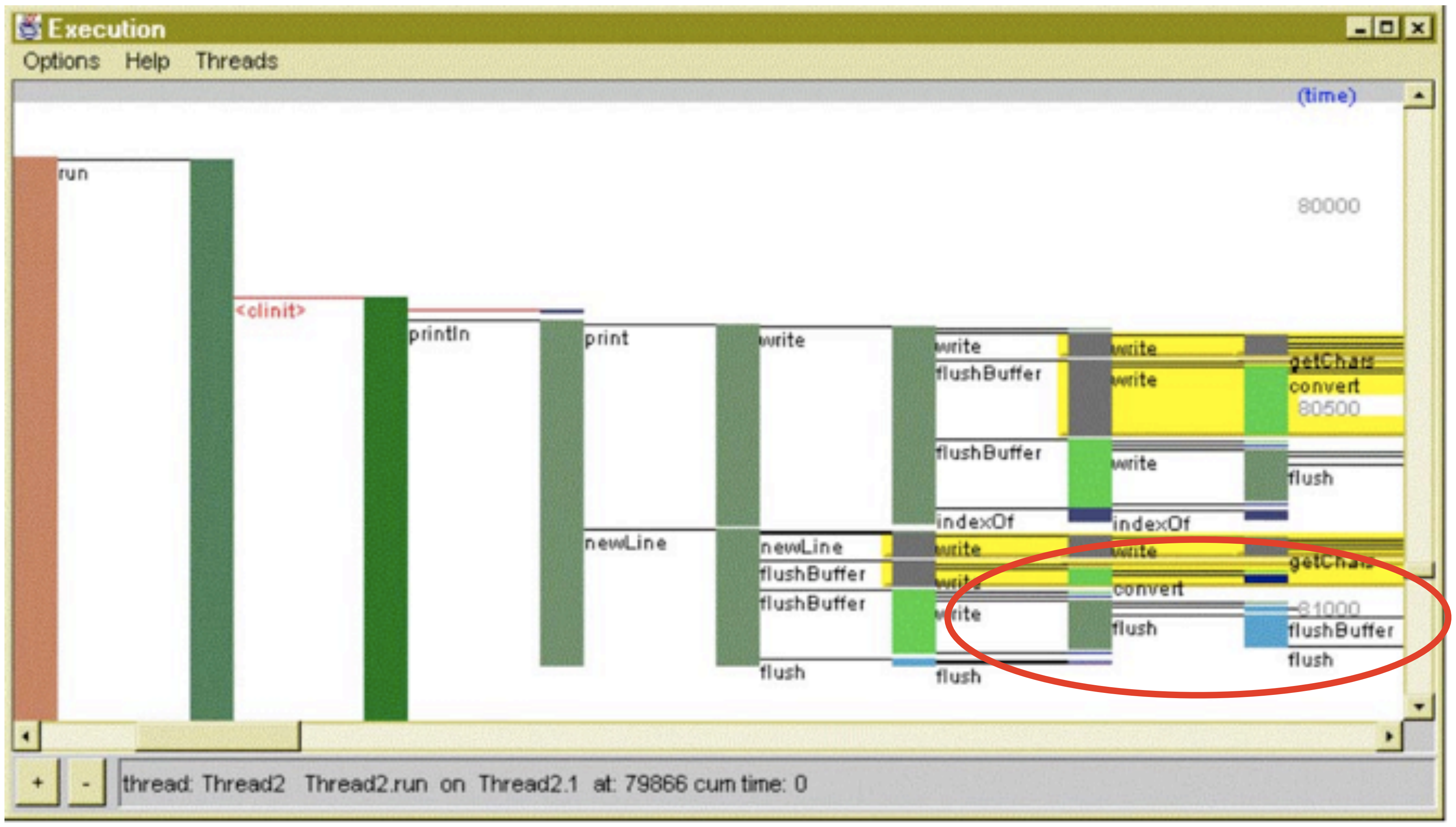
- how many methods of a class were invoked during the execution of a system.

-which classes create objects

-Which classes communicate with each other

Visualization of Runtime Behavior





thread: Thread2 Thread2.run on Thread2.1 at: 79866 cum time: 0

Problem
of
Large
traces

[JinSight, De Pauw 1993]

Traces of execution behavior lead to huge execution traces of tens of thousands of events. This makes them difficult to interpret or to

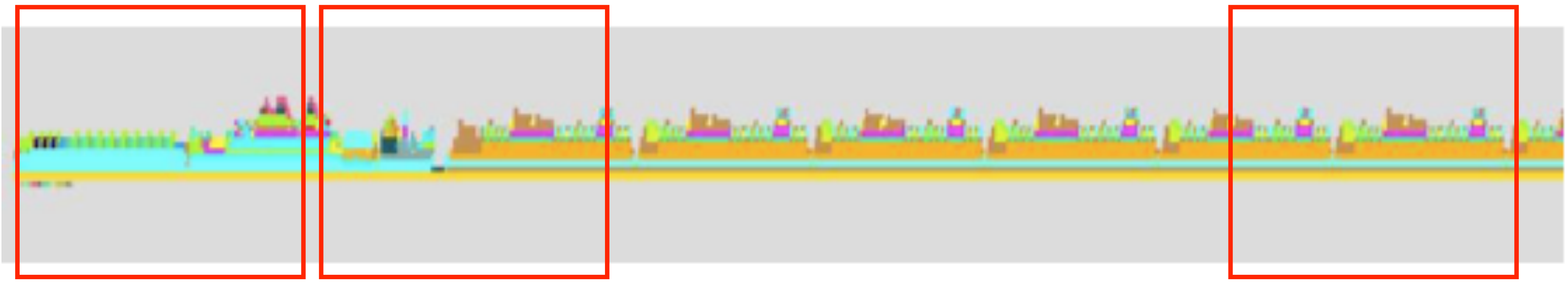
Extract high level views. We need techniques to reduce the volume of information without loss of details needed to answer a specific research question. For example: “Which classes and methods implement the save contact feature?”

Wim dePauw [JinSight, De Pauw 1993].

Other compression approaches

Use graph algorithms to detect patterns and reduce the volume of data. Use patterns to learn something about the system behavior.

Dividing a trace into features



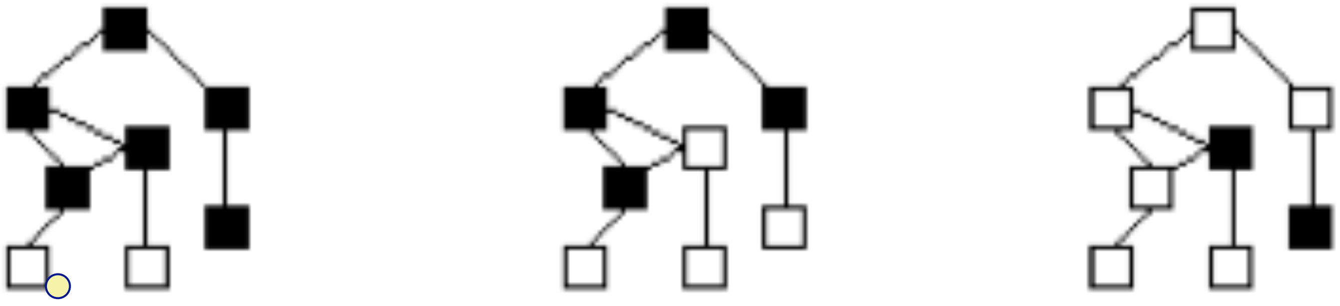
Feature 1

Feature 2

Feature n

Feature Identification is a technique to map features to source code.

"A feature is an observable unit of behavior of a system triggered by the user" [Eisenbarth etal. 2003]

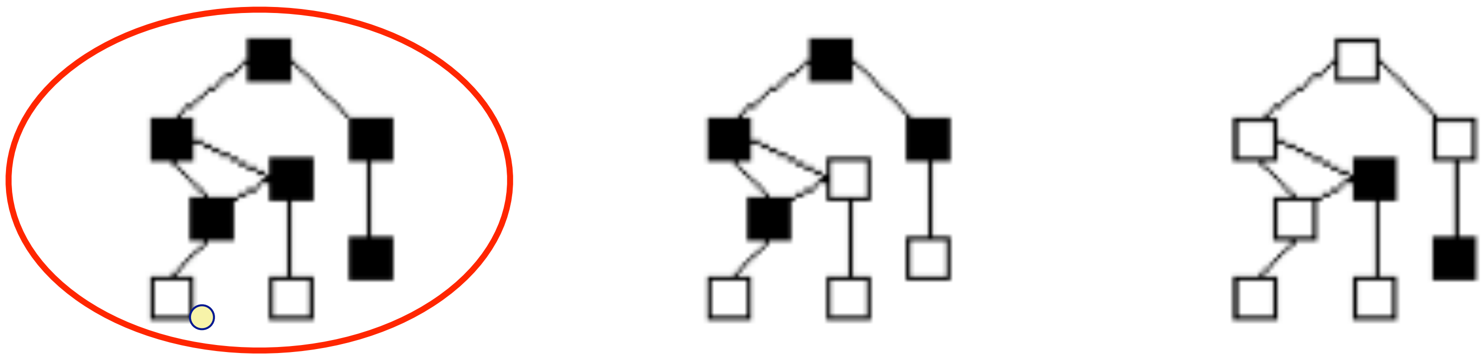


Software Reconnaissance [Wilde and Scully]
Run a (1) feature exhibiting scenario and a
(2) non-exhibiting scenario and compare the traces.
Then browse the source code.

Other researchers had devised variations of software reconnaissance - Antoniol, Eisenberg etc.

Feature Identification is a technique to map features to source code.

"A feature is an observable unit of behavior of a system triggered by the user" [Eisenbarth etal. 2003]

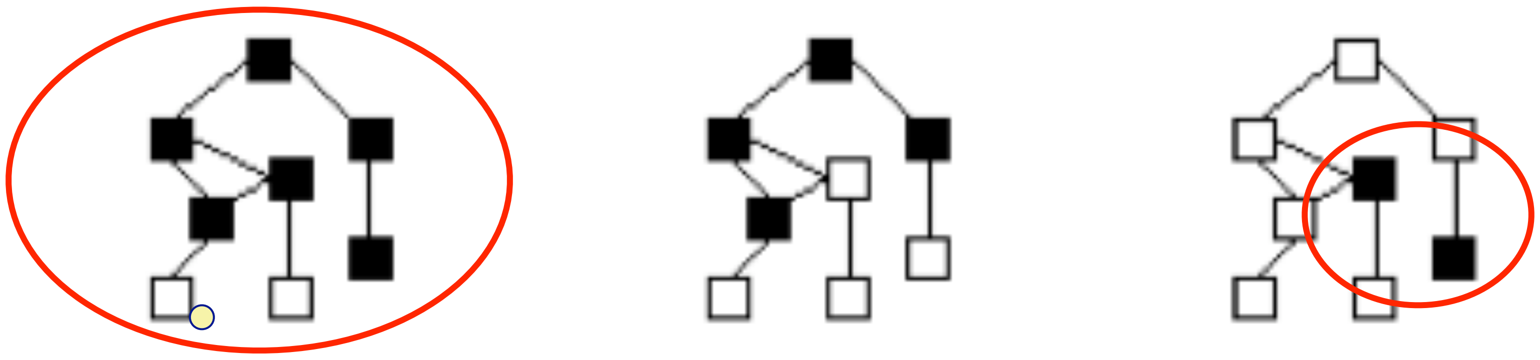


Software Reconnaissance [Wilde and Scully]
Run a (1) feature exhibiting scenario and a
(2) non-exhibiting scenario and compare the traces.
Then browse the source code.

Other researchers had devised variations of software reconnaissance - Antoniol, Eisenberg etc.

Feature Identification is a technique to map features to source code.

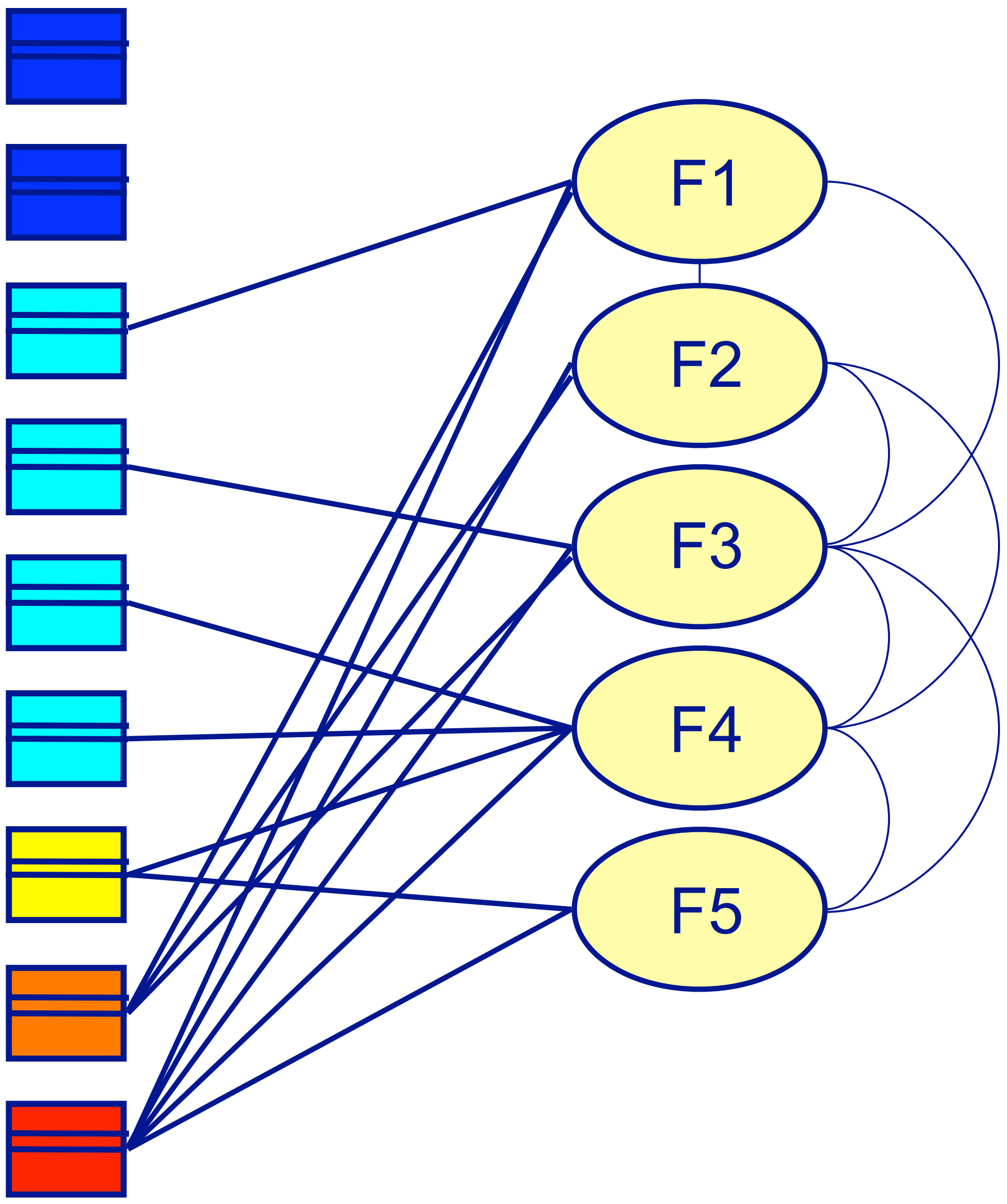
“A feature is an observable unit of behavior of a system triggered by the user” [Eisenbarth etal. 2003]



Software Reconnaissance [Wilde and Scully]
Run a (1) feature exhibiting scenario and a
(2) non-exhibiting scenario and compare the traces.
Then browse the source code.

Other researchers had devised variations of software reconnaissance - Antoniol, Eisenberg etc.

Feature-Centric Analysis:
3 Complementary Perspectives



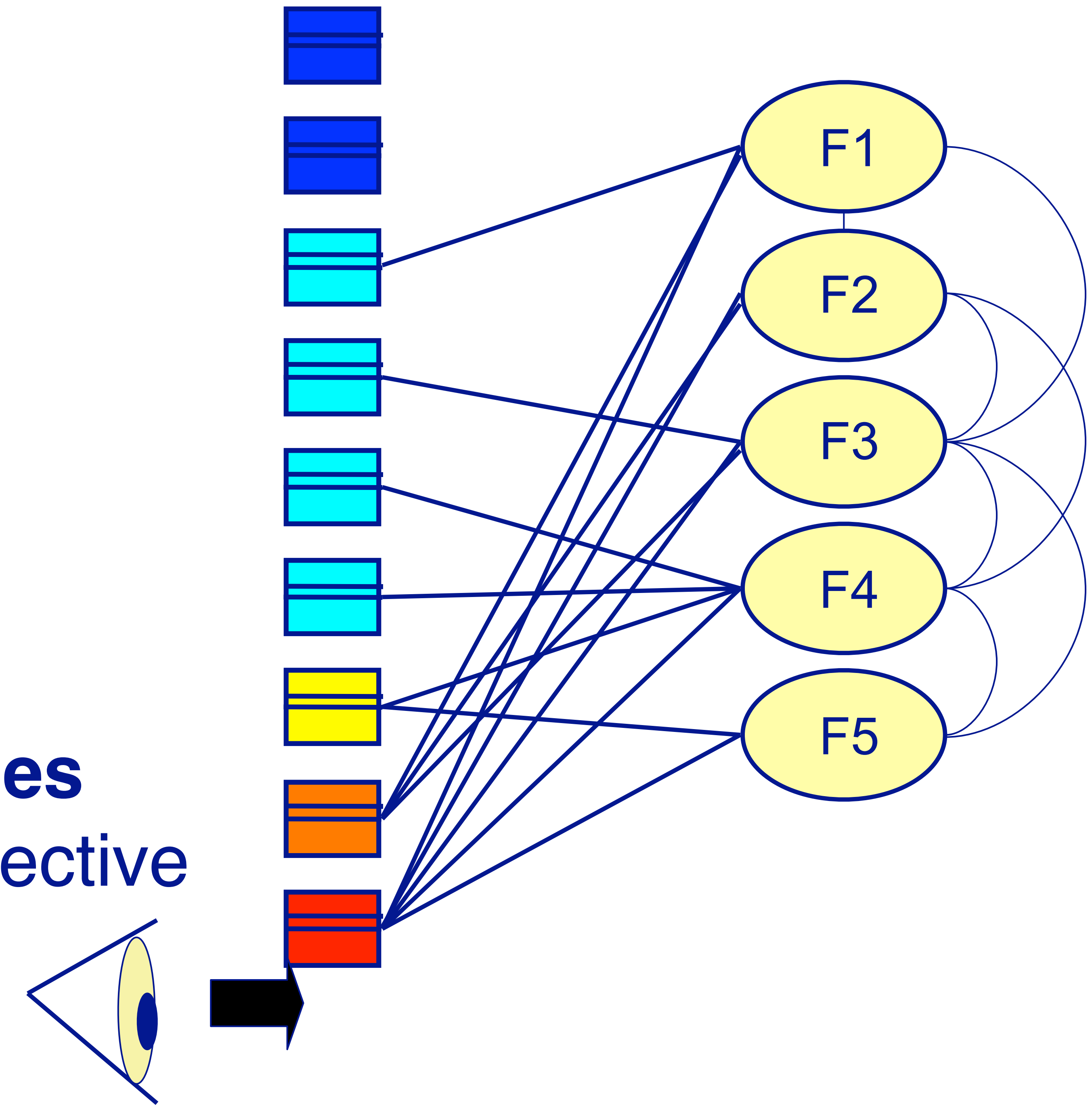
1) How are classes related to features?

2) How are features related to classes?

3) How are features related to each other?

We define a Feature-Affinity metric to distinguish between various levels of characterization of classes.

Feature-Centric Analysis: 3 Complementary Perspectives



Classes
Perspective

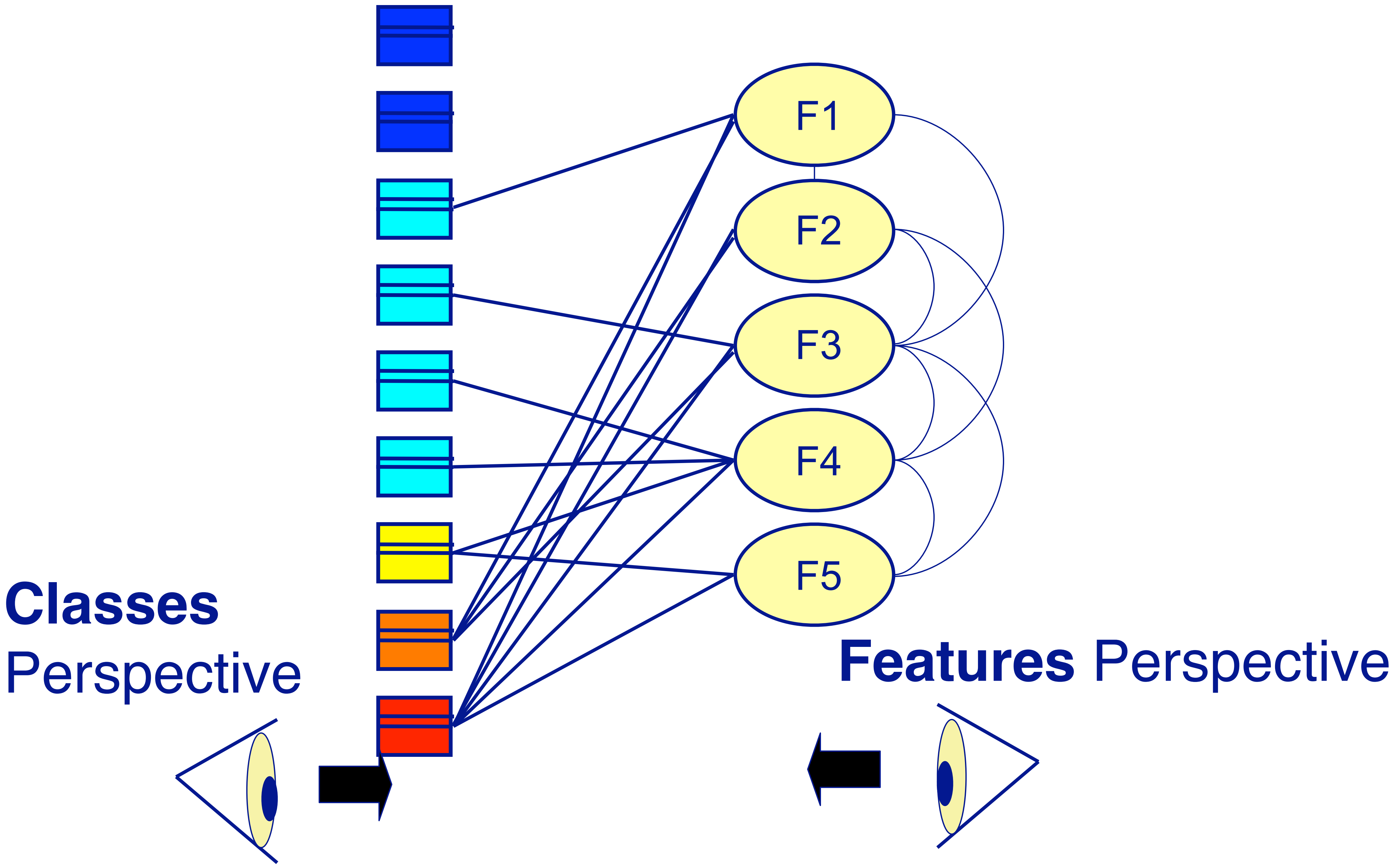
1) How are classes related to features?

2) How are features related to classes?

3) How are features related to each other?

We define a Feature-Affinity metric to distinguish between various levels of characterization of classes.

Feature-Centric Analysis:
3 Complementary Perspectives



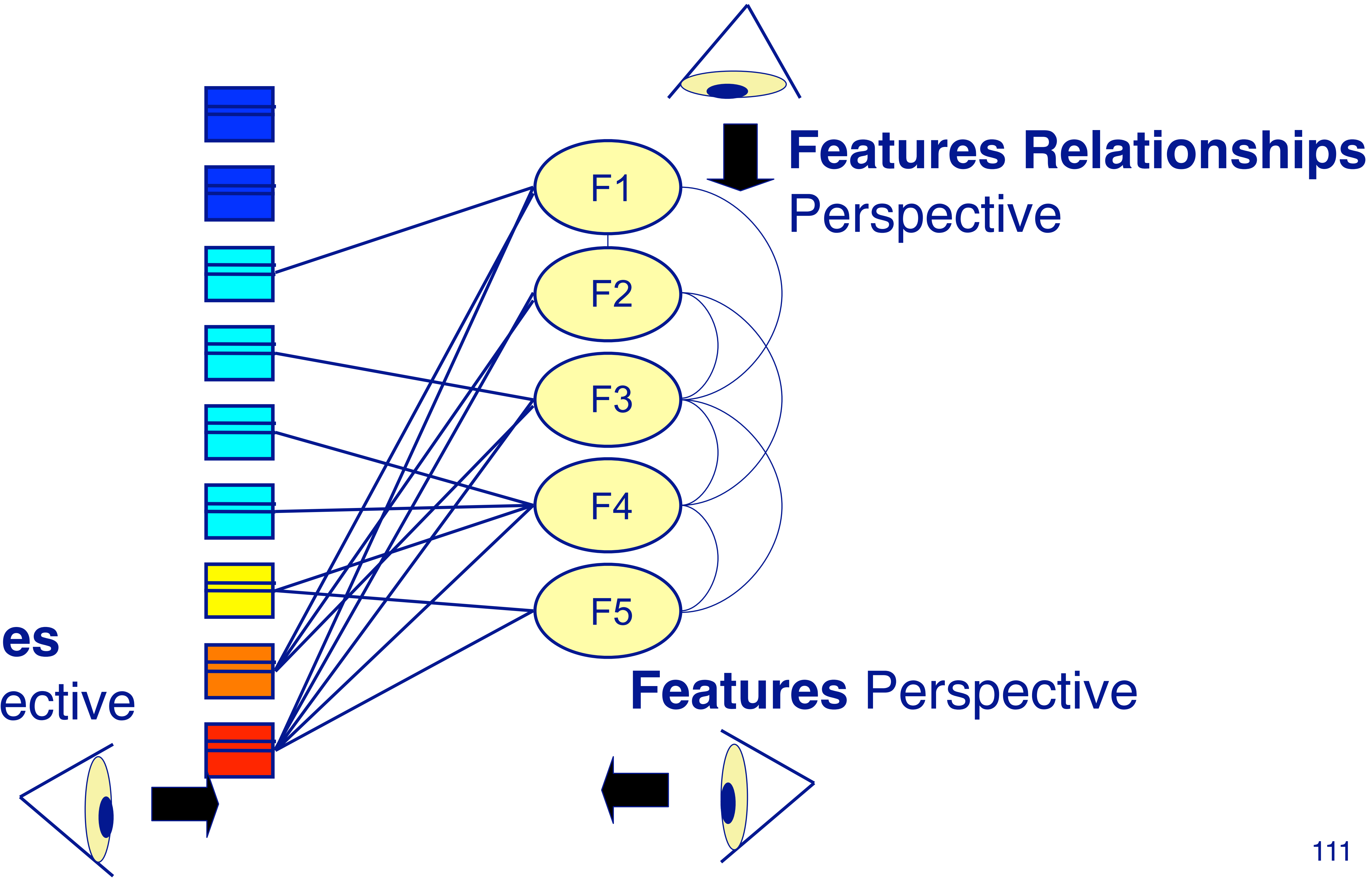
1) How are classes related to features?

2) How are features related to classes?

3) How are features related to each other?

We define a Feature-Affinity metric to distinguish between various levels of characterization of classes.

Feature-Centric Analysis: 3 Complementary Perspectives



Features Relationships Perspective

Classes Perspective

Features Perspective

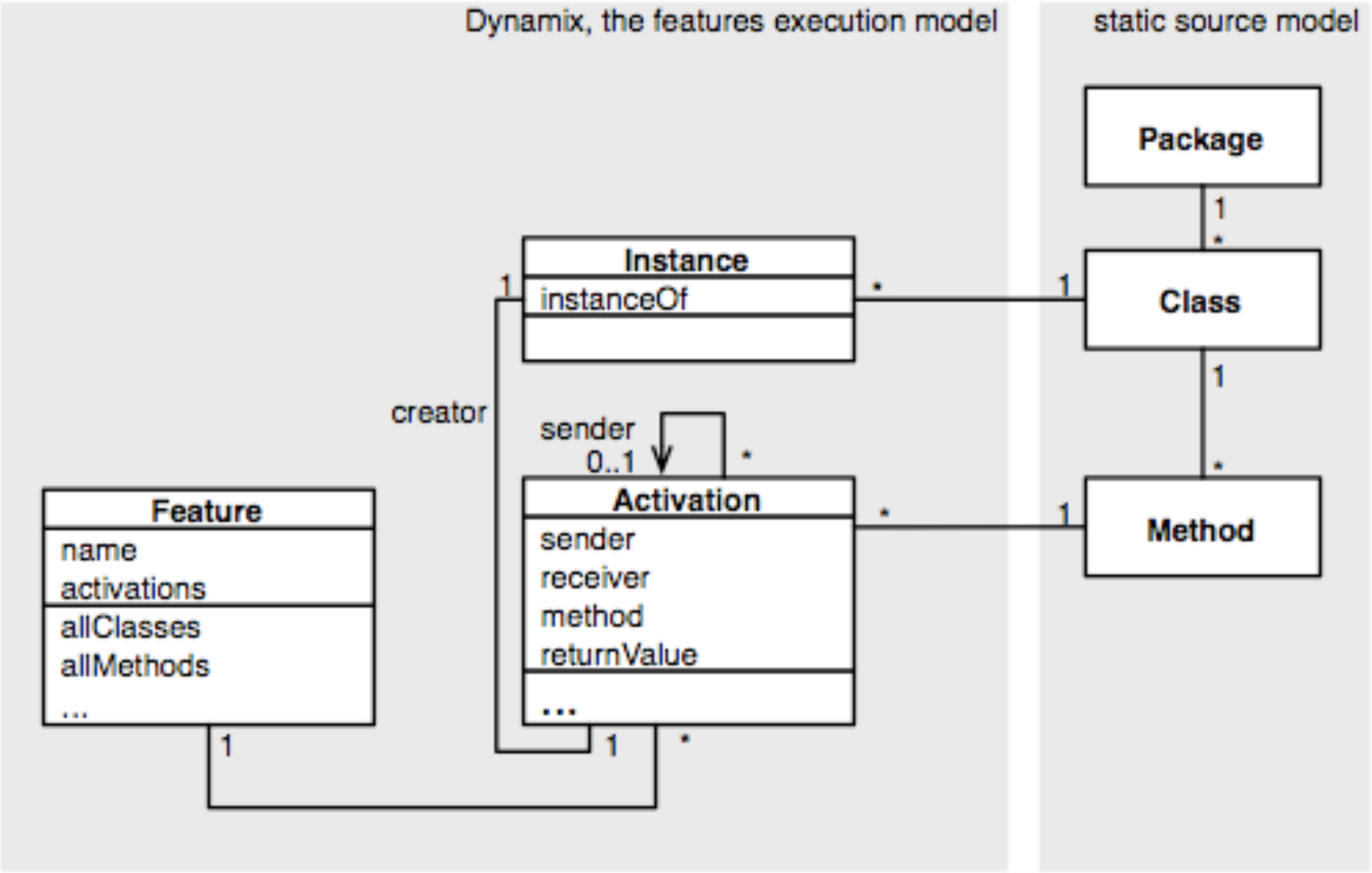
1) How are classes related to features?

2) How are features related to classes?

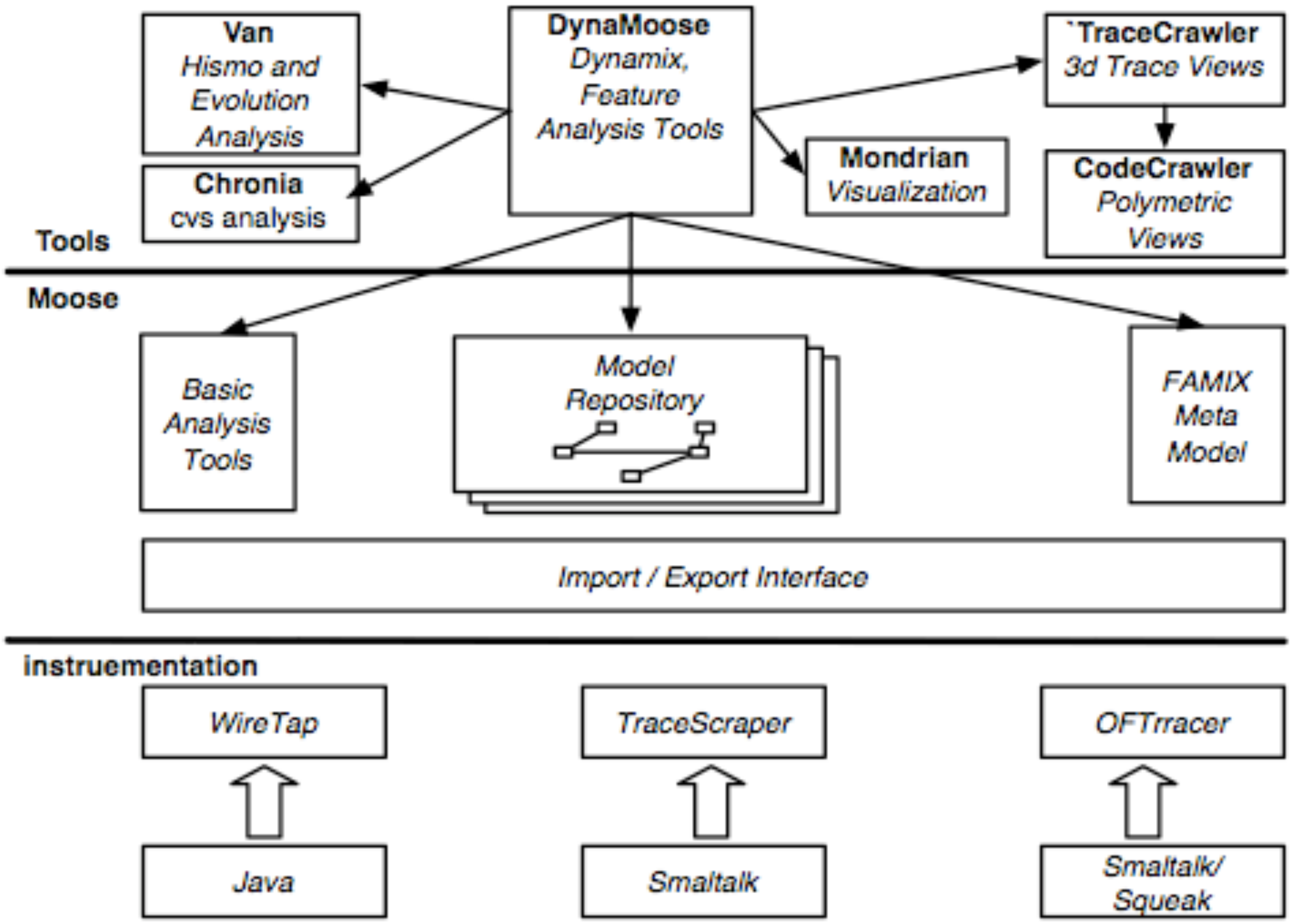
3) How are features related to each other?

We define a Feature-Affinity metric to distinguish between various levels of characterization of classes.

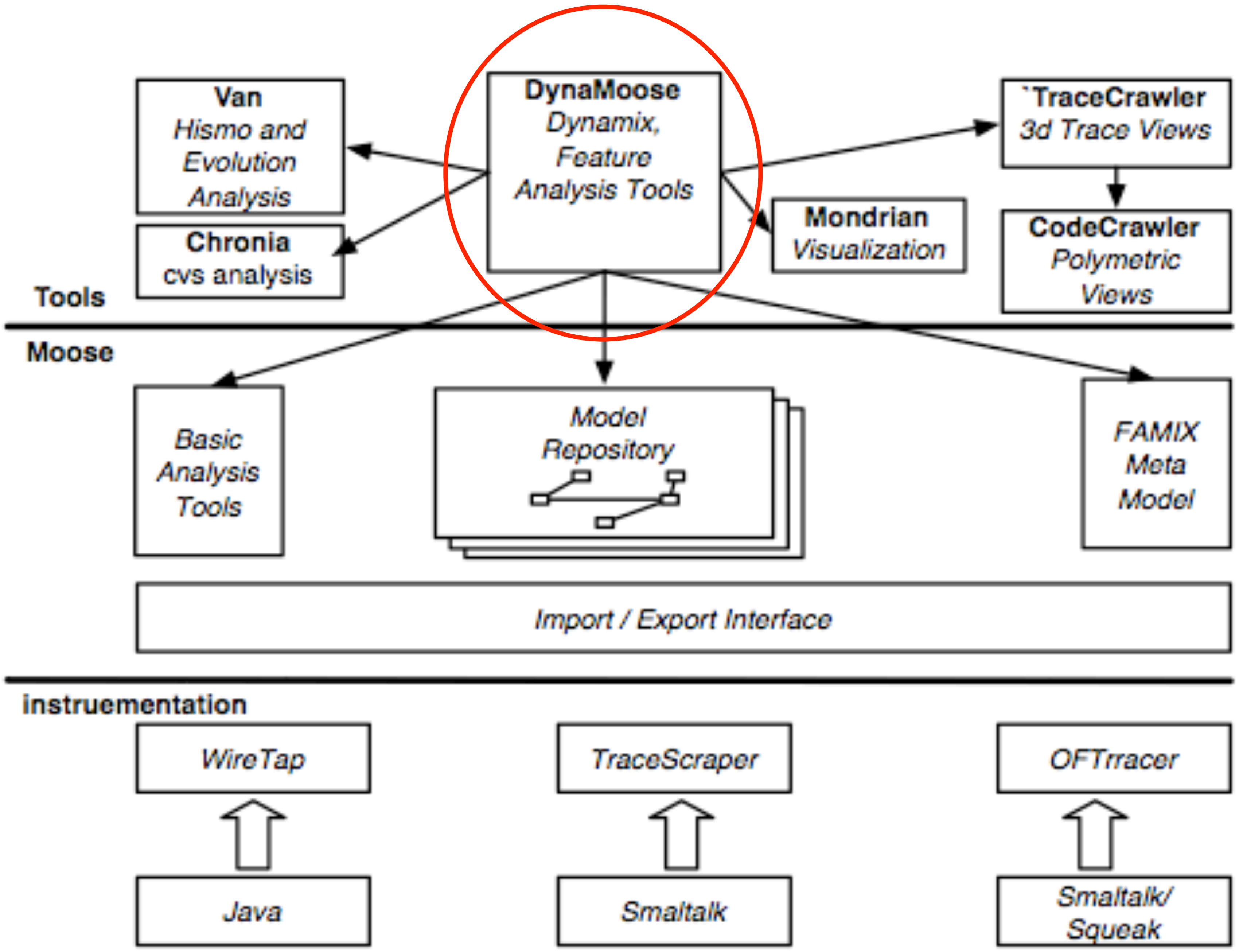
Dynamix - A Model for Dynamic Analysis



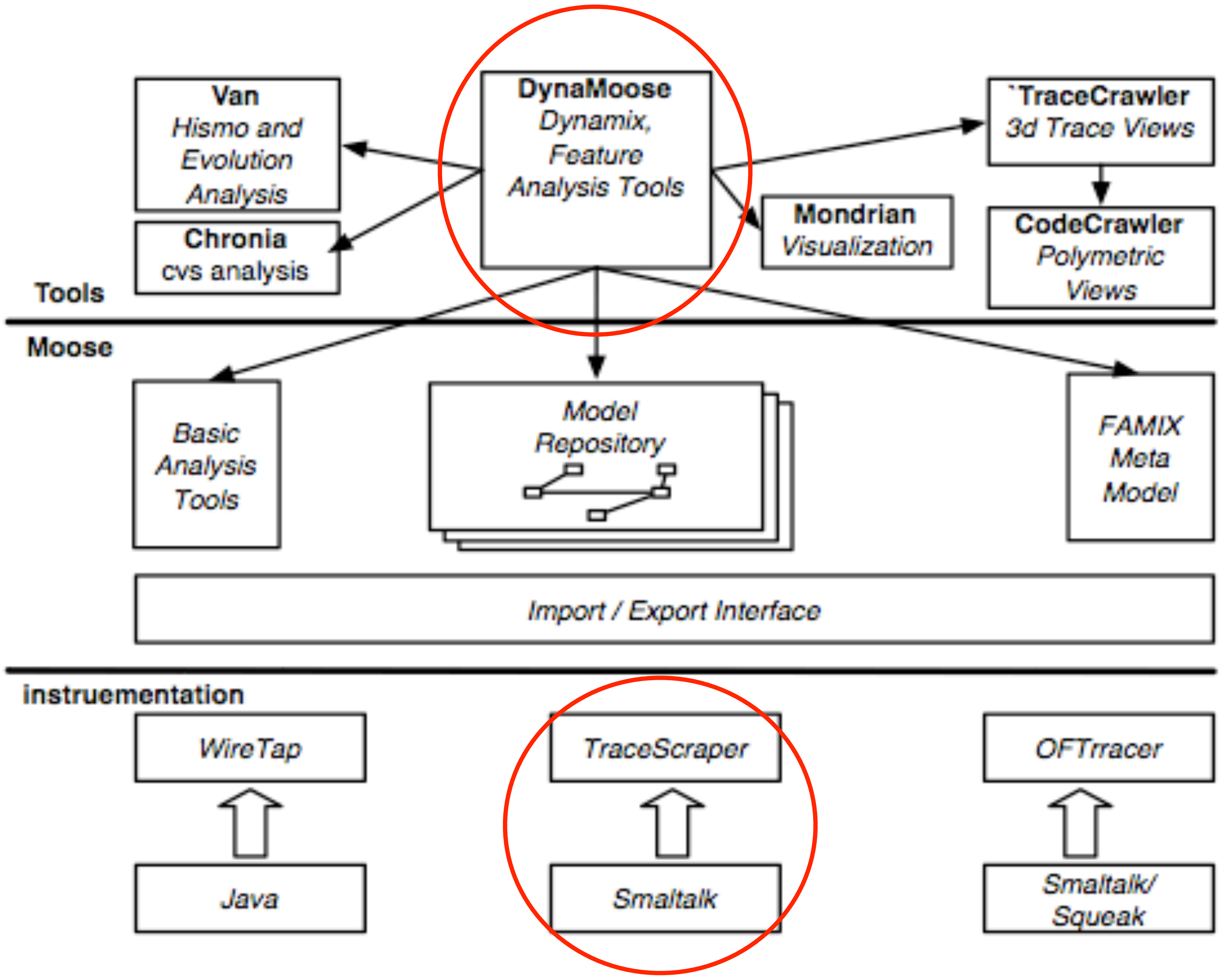
DynaMoose - An Environment for Feature Analysis



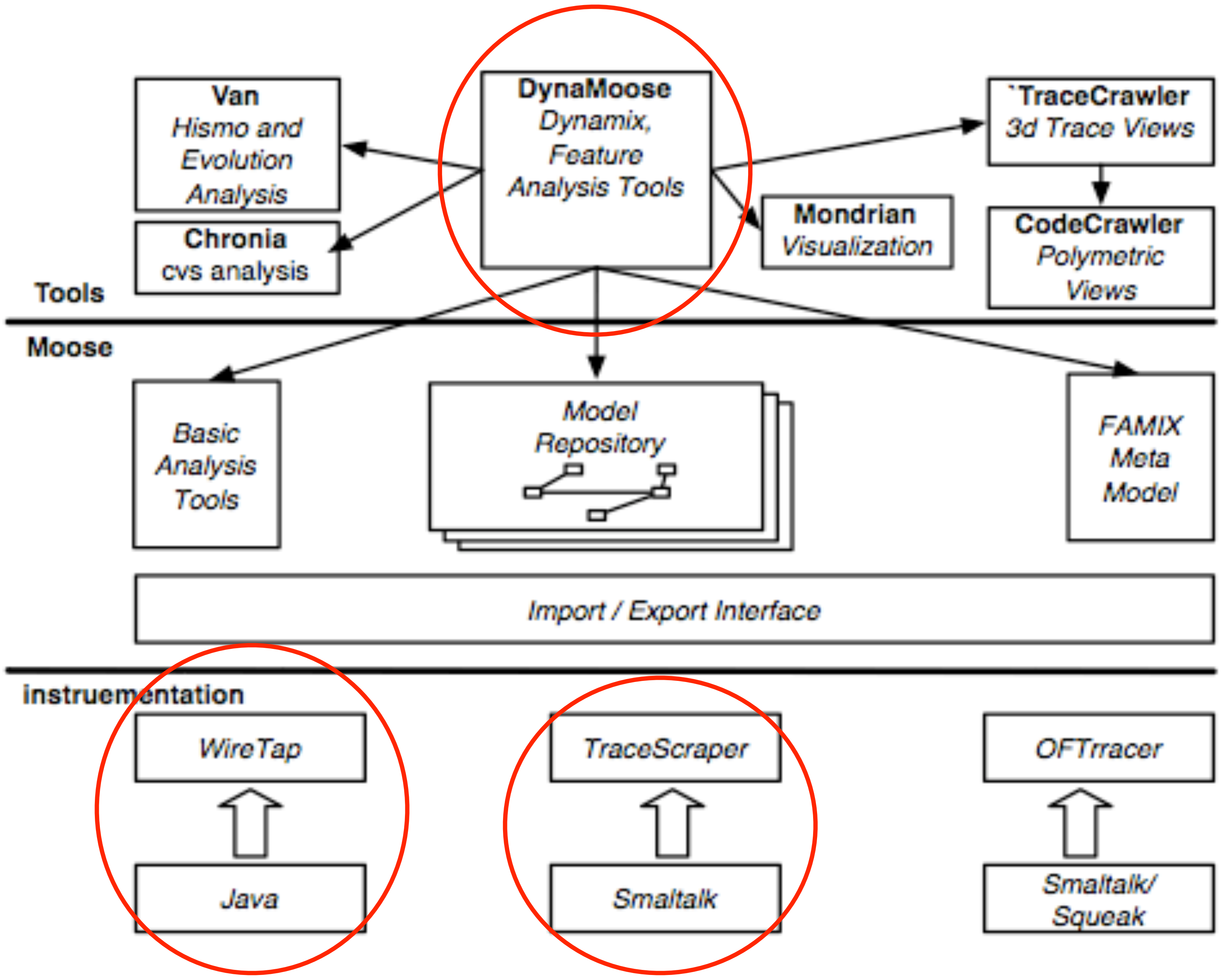
DynaMoose - An Environment for Feature Analysis



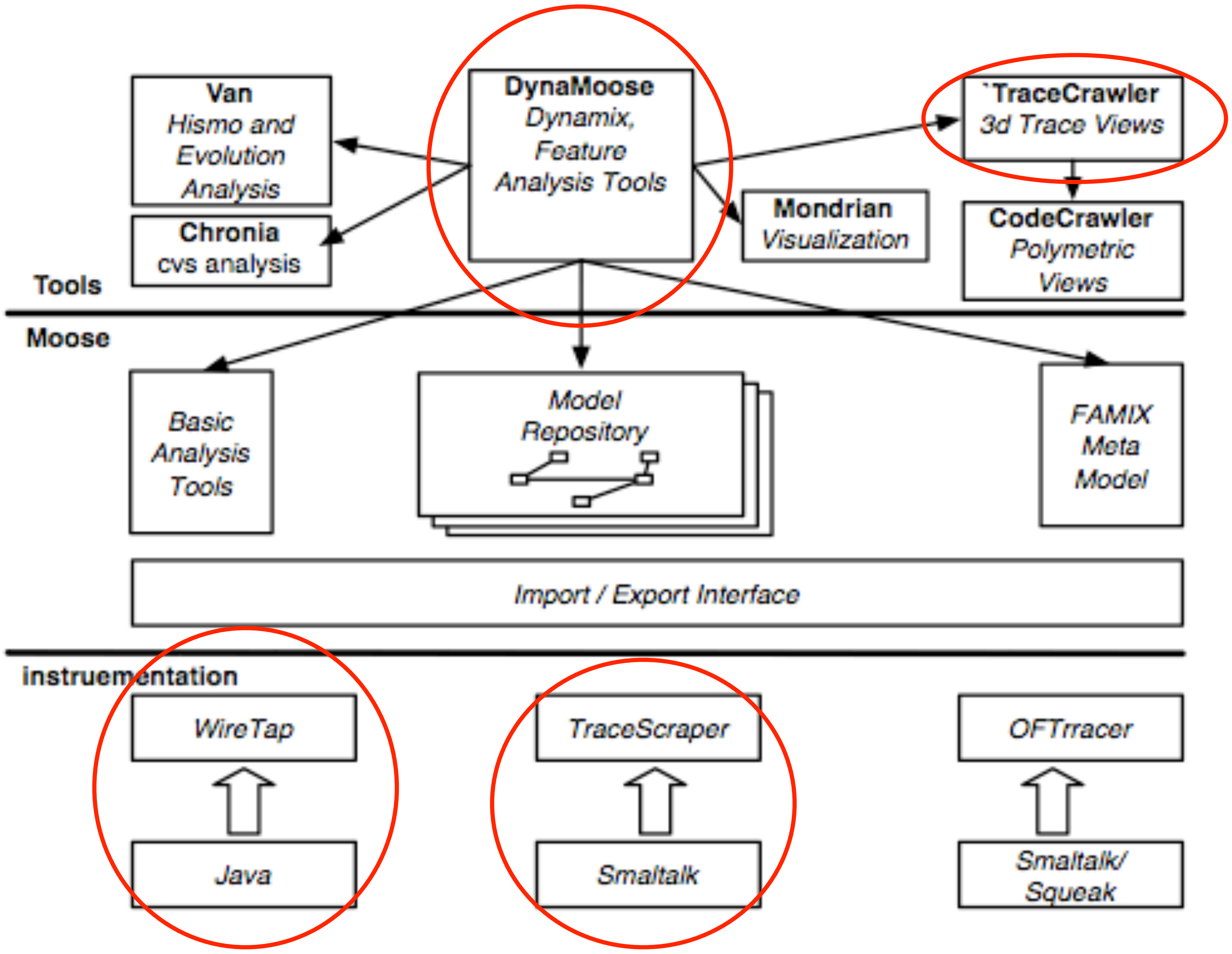
DynaMoose - An Environment for Feature Analysis



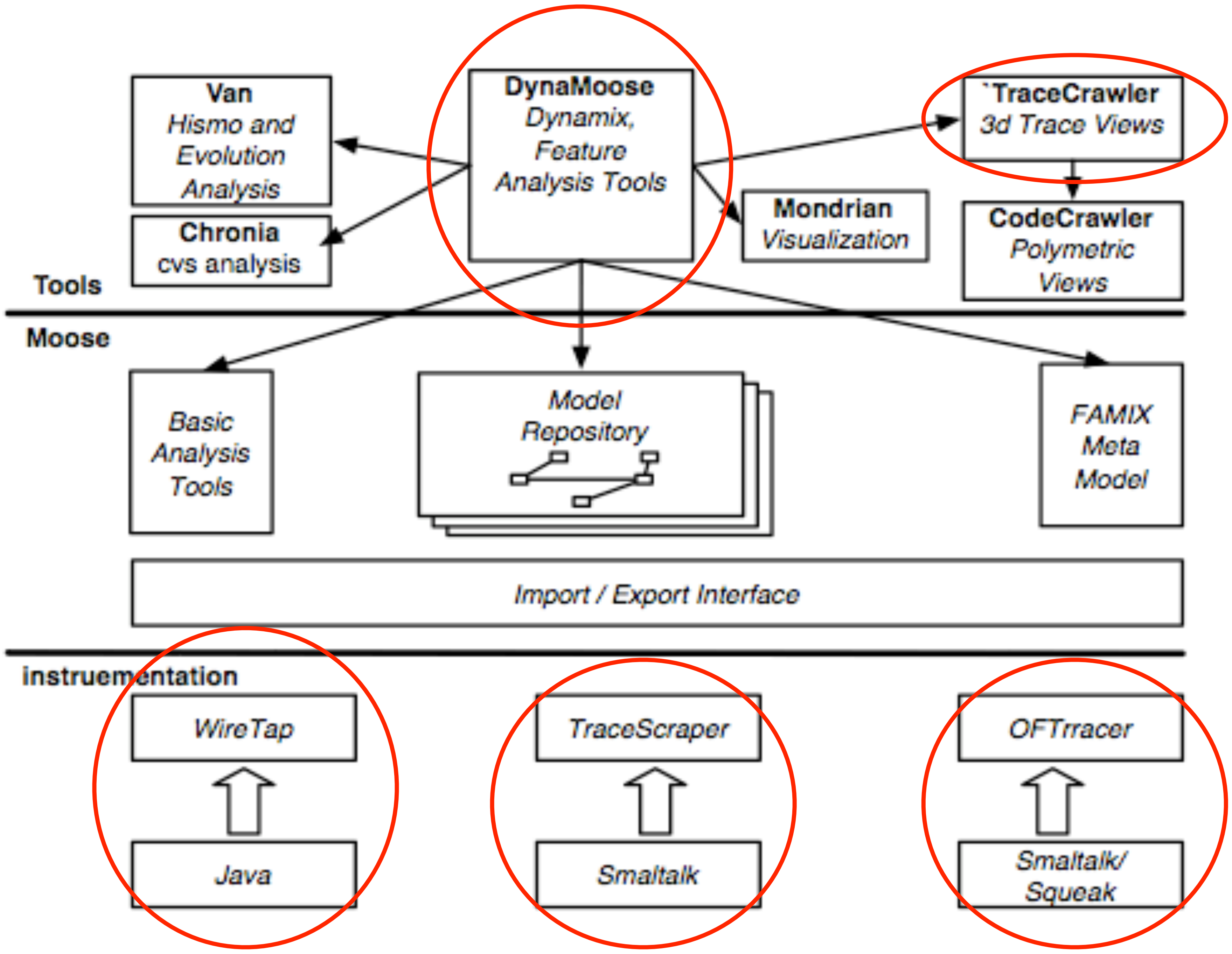
DynaMoose - An Environment for Feature Analysis



DynaMoose - An Environment for Feature Analysis

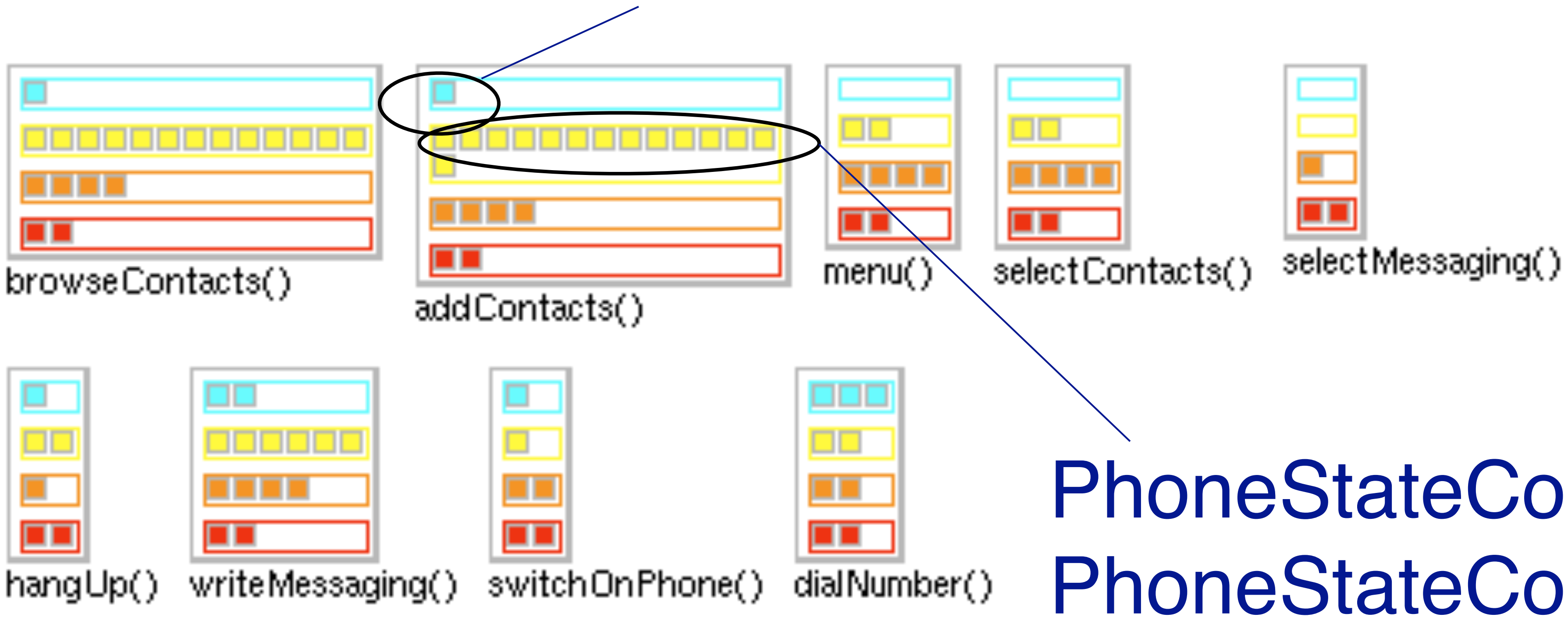


DynaMoose - An Environment for Feature Analysis



Demo of Feature Analysis - Feature Views of Classes

PhoneButtonEventBackSpace



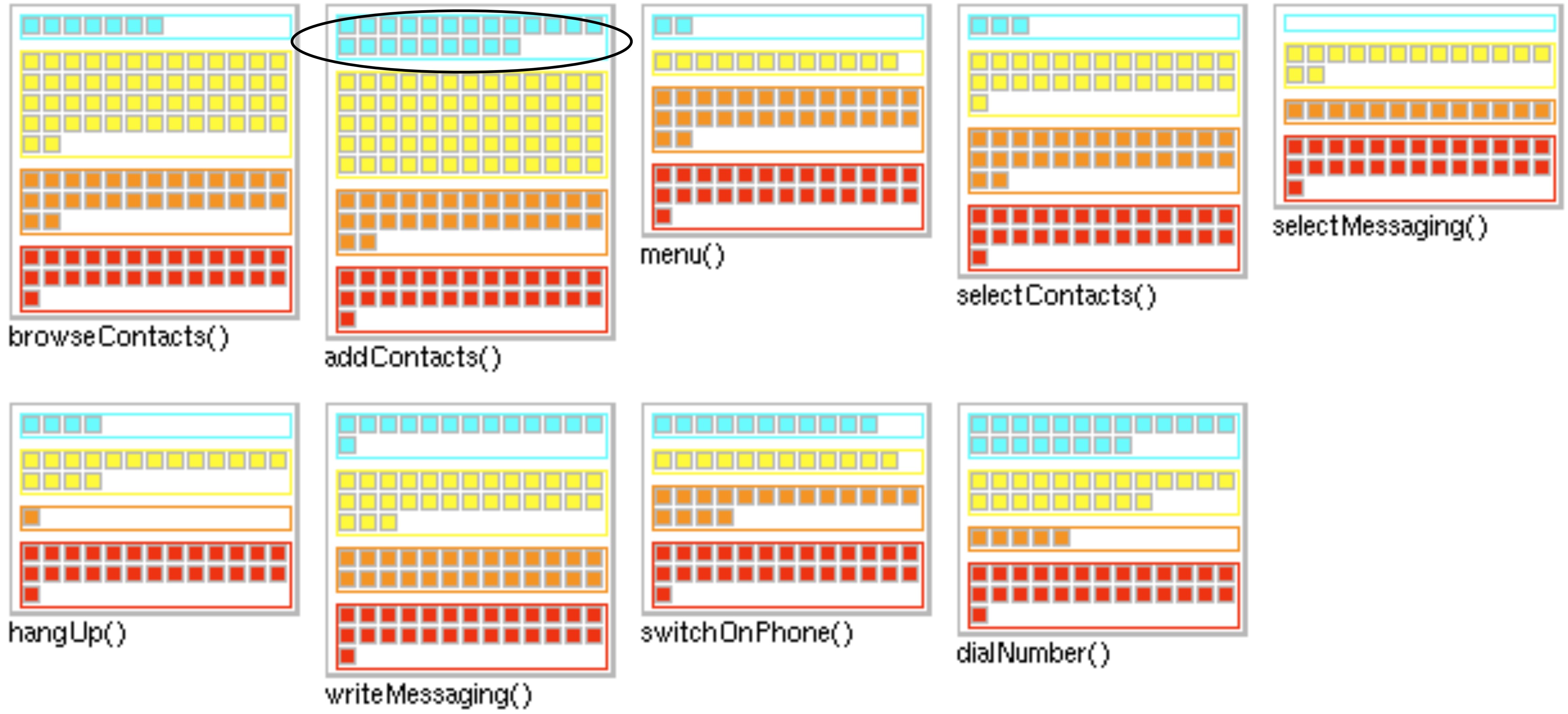
PhoneStateContact
PhoneStateContactForm
EditableText
CustomTextArea

Feature Views of PhoneSim Classes

Here we see the feature views (of classes)

Our question was “Which classes participate in the addContacts feature?”

Feature Views of ‘Phonesim’ Methods



Feature Views of PhoneSim Methods

Which methods participate in the feature ‘addContacts()’?

22 single feature methods

Object Flow Analysis

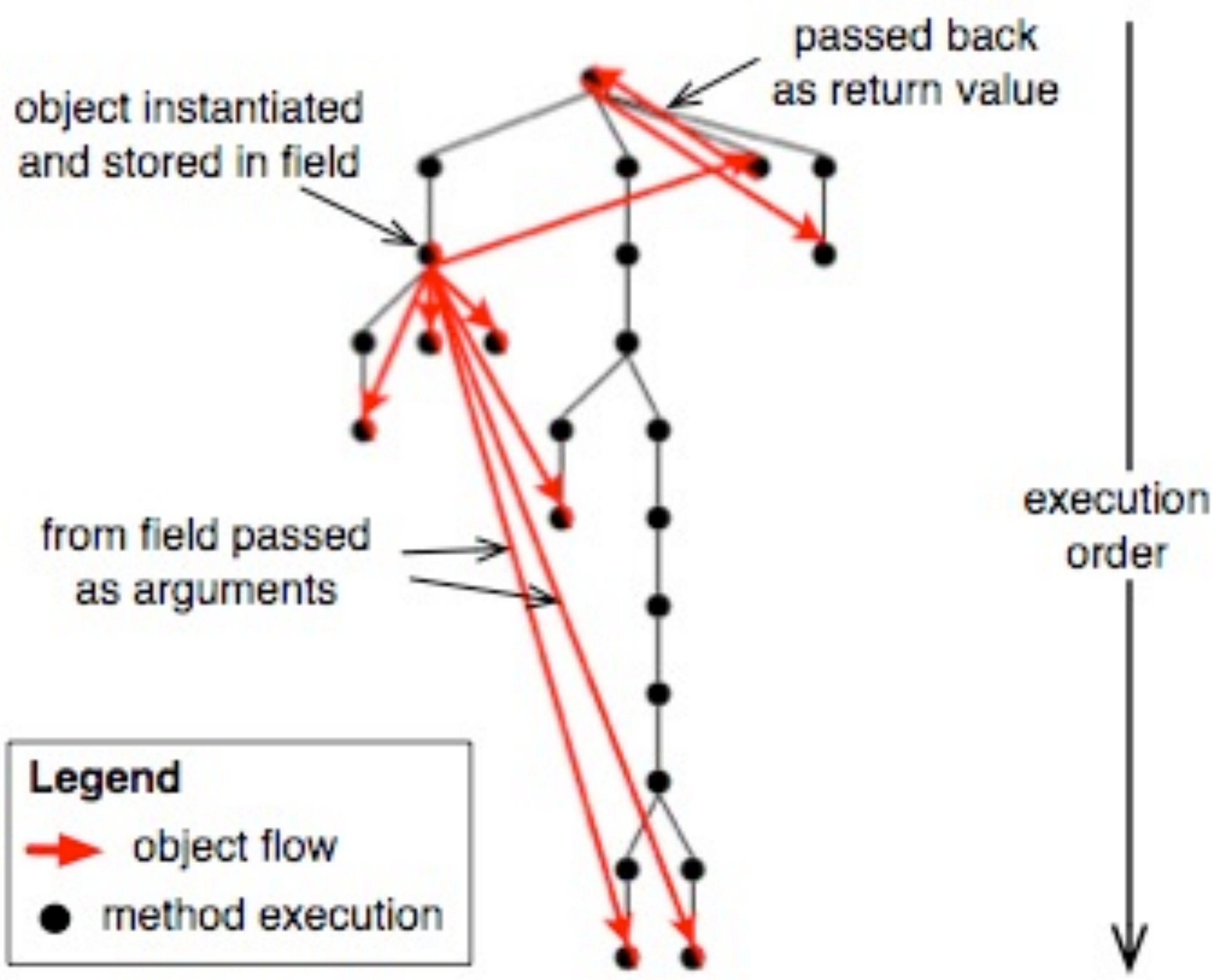
Method execution traces do not reveal how

… objects refer to each other

… object references evolve

Trace and analyze object flow

- Object-centric debugger:_Trace back flow from errors to code that produced the objects
- Detect object dependencies between features



Roadmap

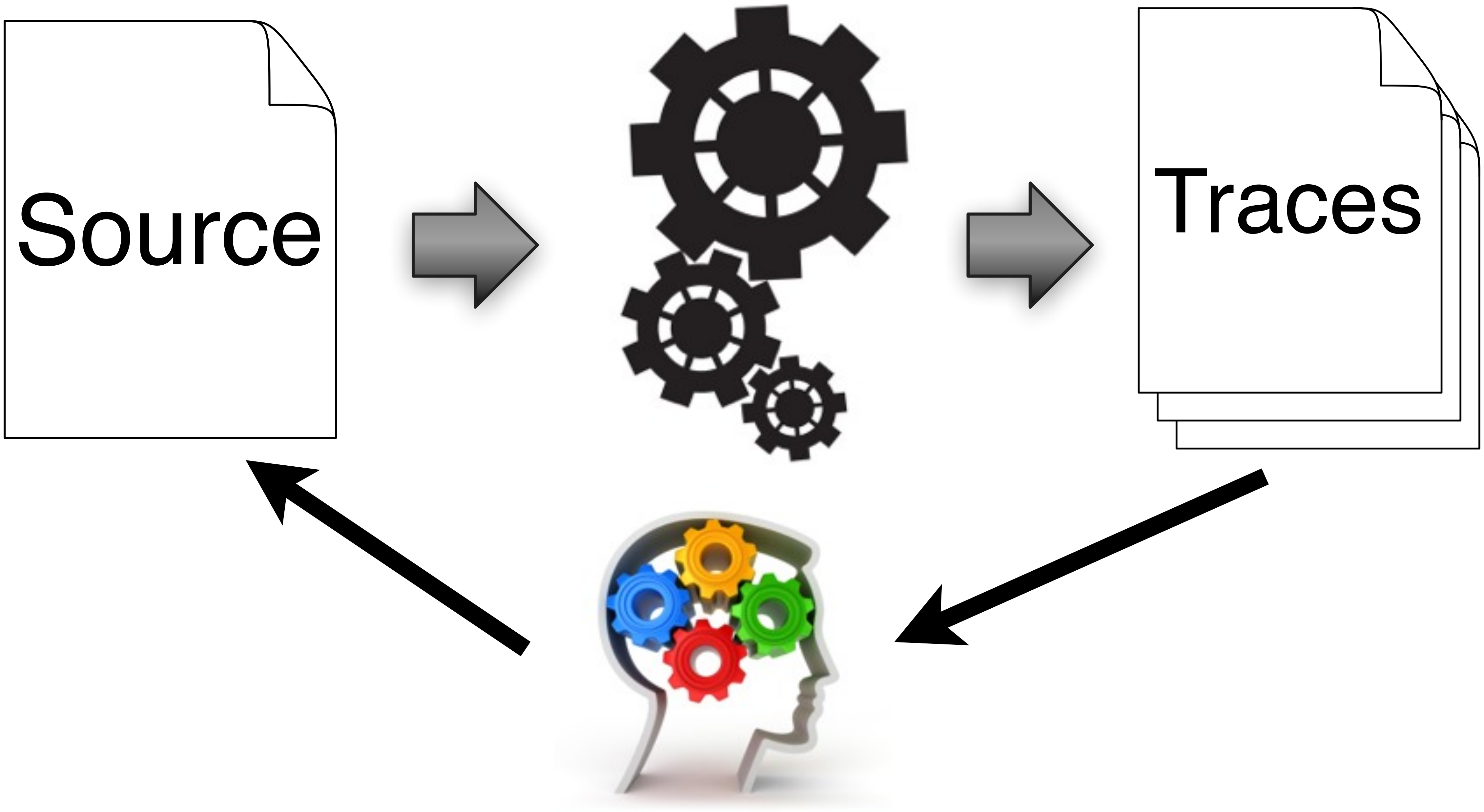
- > Motivation
- > Sources of Runtime Information
- > Dynamic Analysis Techniques
- > Advanced Dynamic Analysis Techniques
- > Dynamic analysis in a Reverse Engineering Context
- > **What can we achieve with all this?**
- > Conclusion

What is the future?

Live Feature Analysis

Live Feature Analysis

Denker etal.



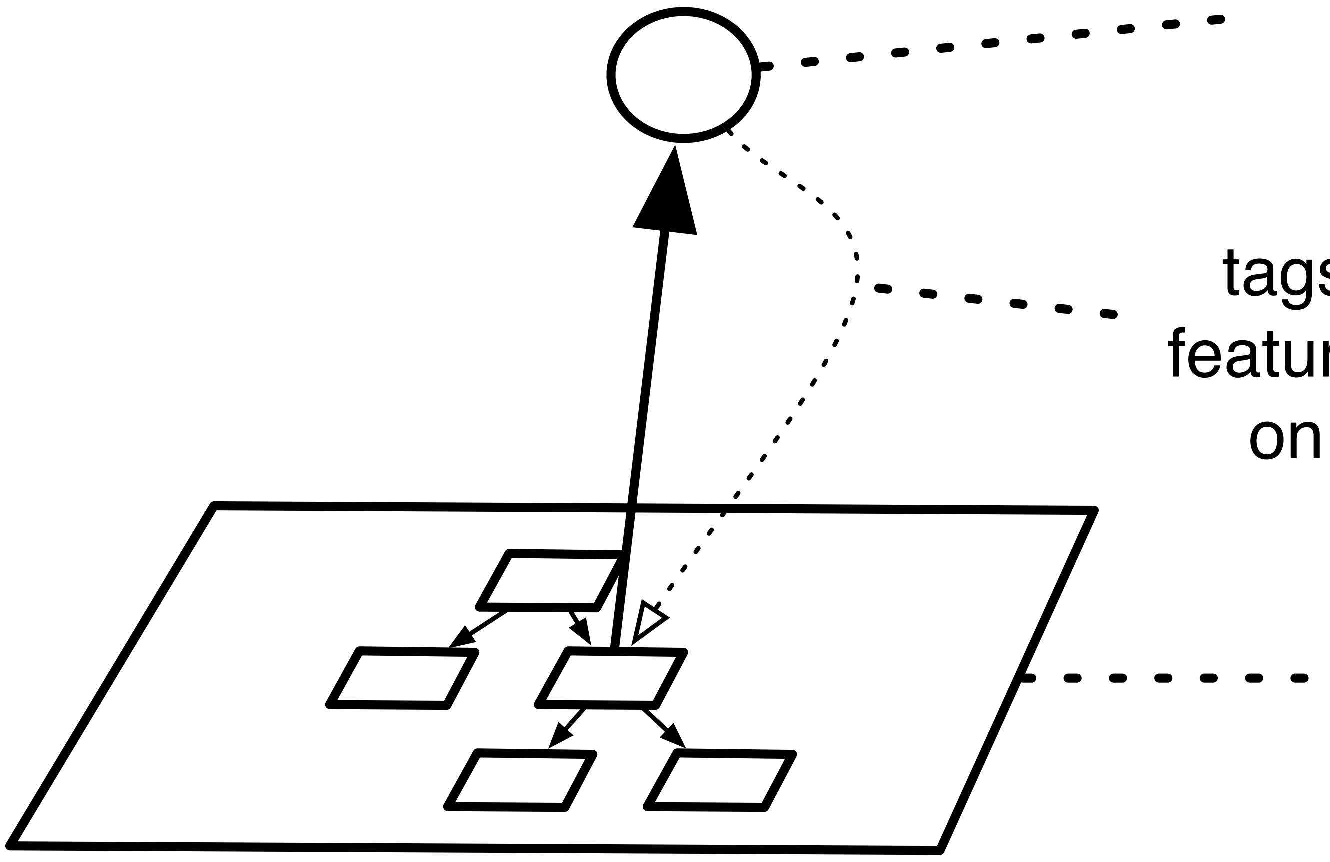
Marcus Denker, Jorge Ressia, Orla Greevy, and Oscar Nierstrasz. Modeling Features at Runtime. In Proceedings of MODELS 2010 Part II, LNCS 6395 p. 138—152, Springer-Verlag, October 2010.

Live Feature Analysis

Denker etal.

Marcus Denker, Jorge Ressia, Orla Greevy, and Oscar Nierstrasz. Modeling Features at Runtime. In Proceedings of MODELS 2010 Part II, LNCS 6395 p. 138—152, Springer-Verlag, October 2010.

Live Feature Analysis



Feature tagger
meta-object

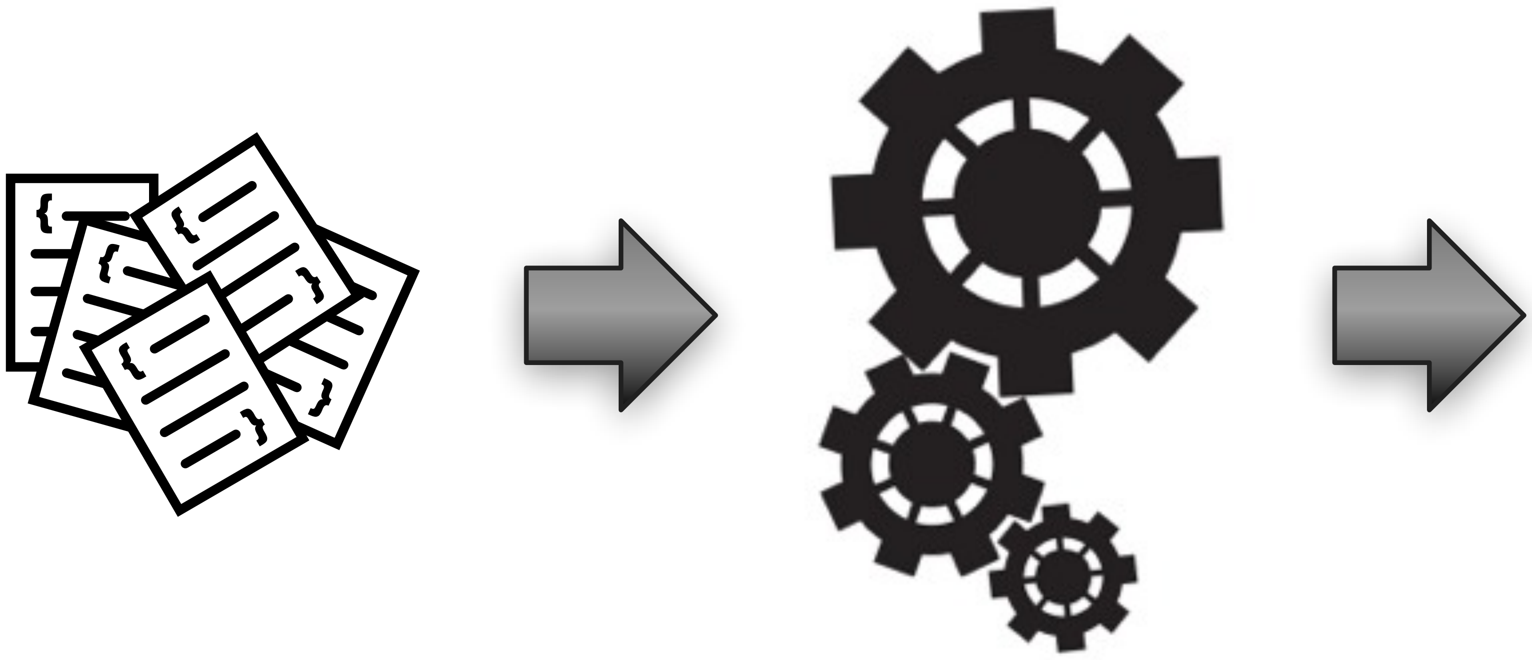
tags node with
feature annotation
on execution

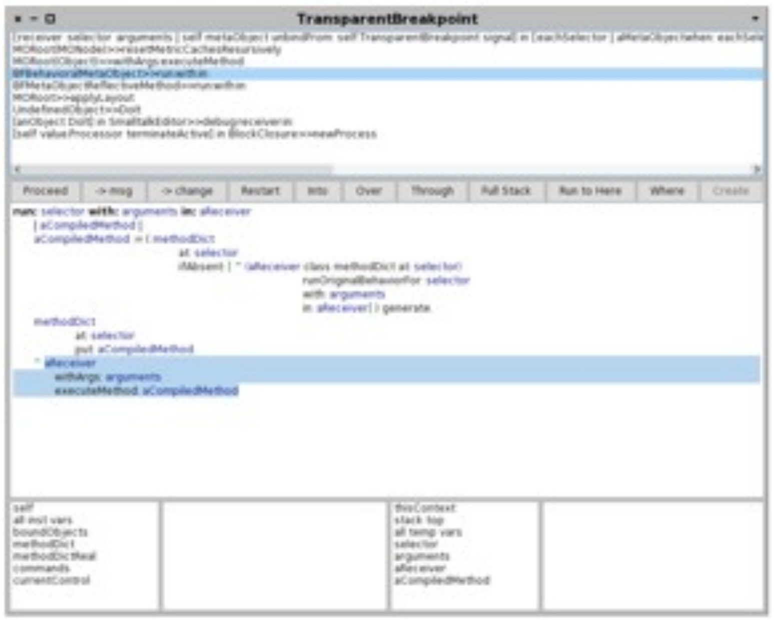
source code
(AST)

What is the future?

Object Centric Debugging

Object Centric Debugging



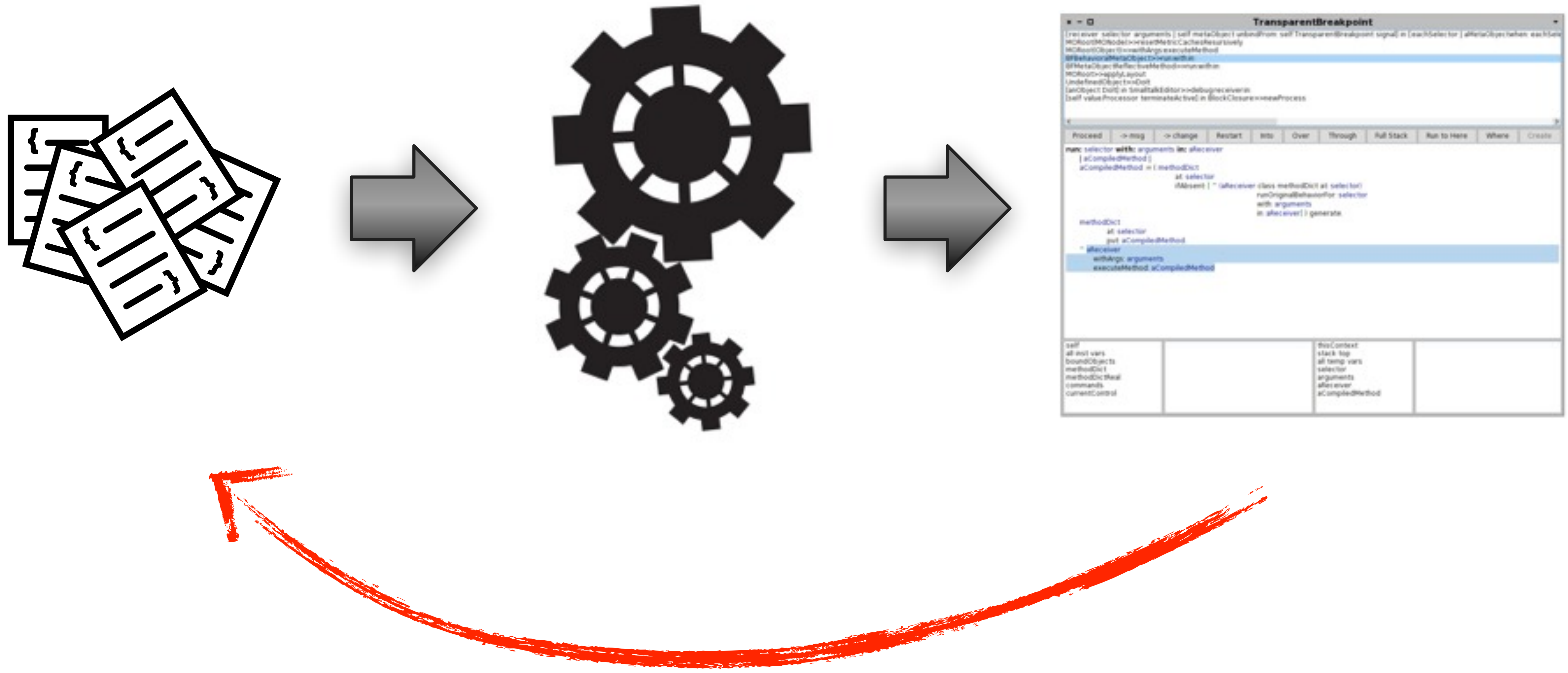


```
run: selector with: arguments in: aReceiver
    | aCompiledMethod |
    aCompiledMethod := ( methodDict
                    at: selector
                    ifAbsent: [ ^ (aReceiver class methodDict at: selector)
                                runOriginalBehaviorFor: selector
                                with: arguments
                                in: aReceiver] ) generate
    methodDict
        at: selector
        put: aCompiledMethod
    ^ aReceiver
        withArgs: arguments
        executeMethod: aCompiledMethod
```

http://scg.unibe.ch/research/bifrost/OCD

Marcus Denker, Jorge Ressia, Orla Greevy, and Oscar Nierstrasz. Modeling Features at Runtime. In Proceedings of MODELS 2010 Part II, LNCS 6395 p. 138—152, Springer-Verlag, October 2010.

Object Centric Debugging



```
run: selector with: arguments in: aReceiver
    | aCompiledMethod |
    aCompiledMethod := ( methodDict
                            at: selector
                            ifAbsent: [ ^ (aReceiver class methodDict at: selector)
                                            runOriginalBehaviorFor: selector
                                            with: arguments
                                            in: aReceiver] ) generate.
    methodDict
        at: selector
        put: aCompiledMethod.
    ^ aReceiver
        withArgs: arguments
        executeMethod: aCompiledMethod
```

http://scg.unibe.ch/research/bifrost/OCD

Marcus Denker, Jorge Ressia, Orla Greevy, and Oscar Nierstrasz. [Modeling Features at Runtime](). In Proceedings of MODELS 2010 Part II, LNCS 6395 p. 138—152, Springer-Verlag, October 2010.

Object Centric Debugging

http://scg.unibe.ch/research/bifrost/OCD

Marcus Denker, Jorge Ressia, Orla Greevy, and Oscar Nierstrasz. [Modeling Features at Runtime](). In Proceedings of MODELS 2010 Part II, LNCS 6395 p. 138—152, Springer-Verlag, October 2010.

Object Centric Debugging

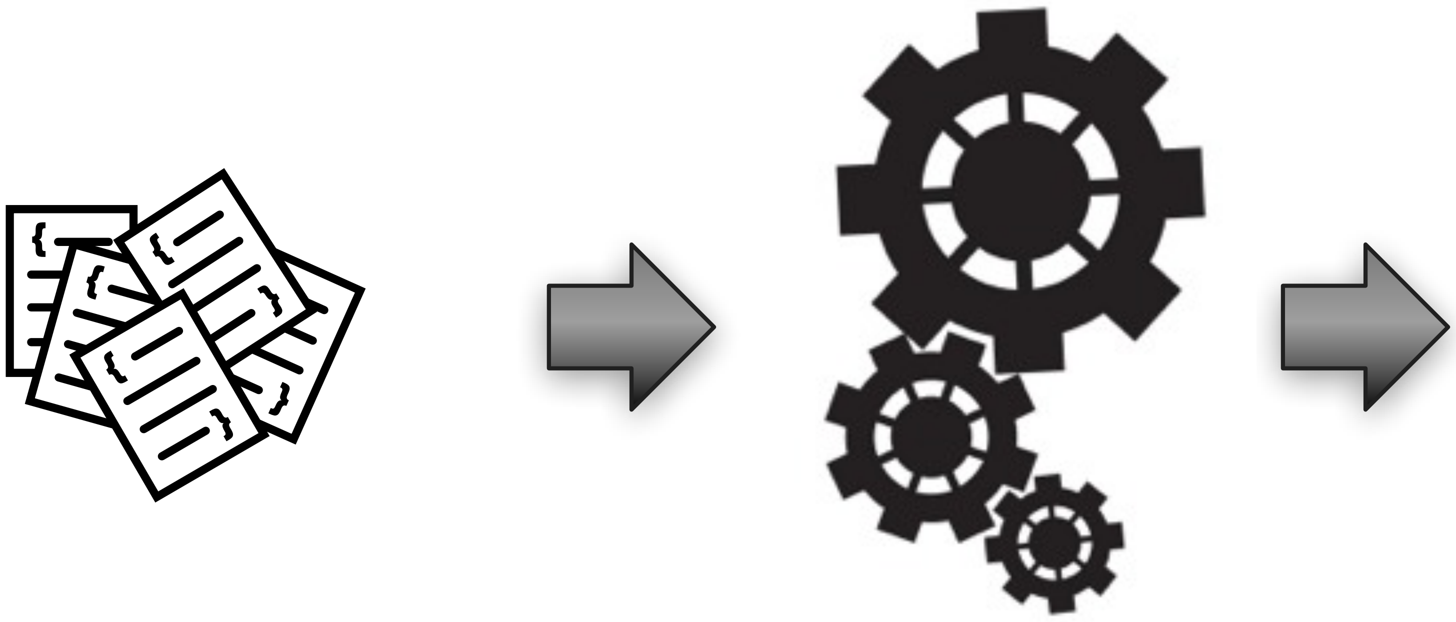
```
Object>>haltAtNextMessage
   | aMetaObject |
   aMetaObject := BFBehavioralMetaObject new.
   aMetaObject
      when: (BFMessageReceiveEvent new)
      do: [ self metaObject unbindFrom: self.
            TransparentBreakpoint signal ].
   aMetaObject bindTo: self
```

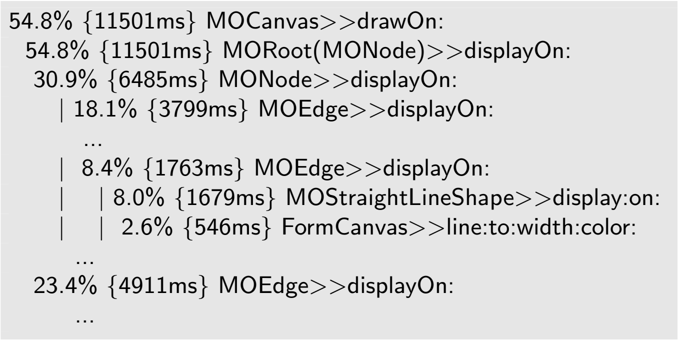
What is the future?

MetaSpy
Domain-specific
Profiling

Domain-specific Profiling

Profile





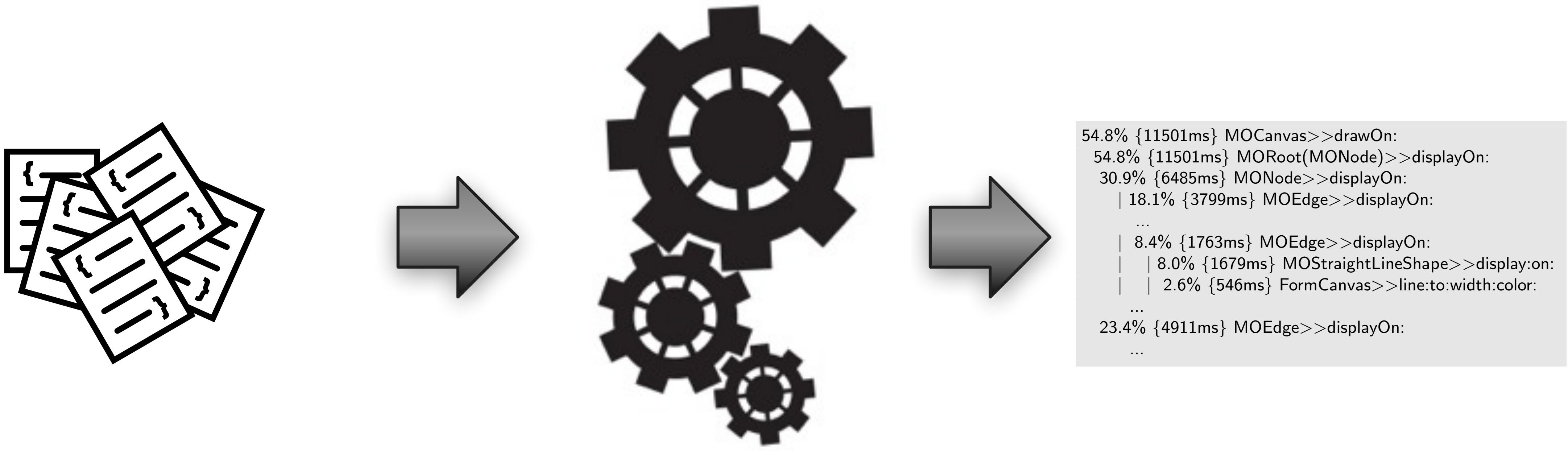
```
54.8% {11501ms} MOCanvas>>drawOn:
  54.8% {11501ms} MORoot(MONode)>>displayOn:
   30.9% {6485ms} MONode>>displayOn:
      | 18.1% {3799ms} MOEdge>>displayOn:
         ...
      |  8.4% {1763ms} MOEdge>>displayOn:
      |    | 8.0% {1679ms} MOStraightLineShape>>display:on:
      |    |  2.6% {546ms} FormCanvas>>line:to:width:color:
        ...
   23.4% {4911ms} MOEdge>>displayOn:
        ...
```

http://scg.unibe.ch/research/bifrost/metaspy

Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-Specific Profiling. In Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11), LNCS 6705 p. 68—82, Springer-Verlag, June 2011.

Domain-specific Profiling

Profile



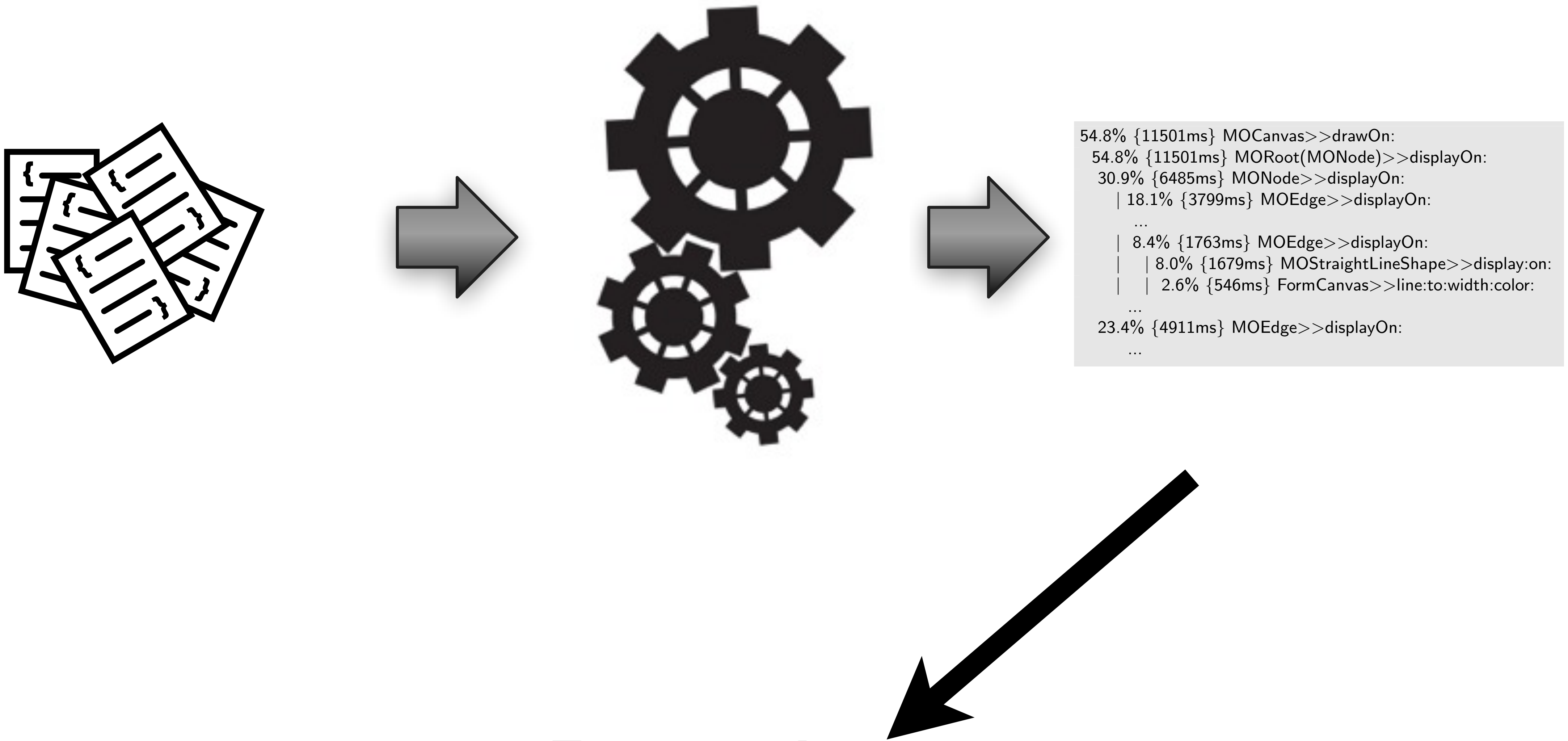
```
54.8% {11501ms} MOCanvas>>drawOn:
  54.8% {11501ms} MORoot(MONode)>>displayOn:
   30.9% {6485ms} MONode>>displayOn:
      | 18.1% {3799ms} MOEdge>>displayOn:
           ...
      |  8.4% {1763ms} MOEdge>>displayOn:
      |    | 8.0% {1679ms} MOStraightLineShape>>display:on:
      |    |  2.6% {546ms} FormCanvas>>line:to:width:color:
          ...
   23.4% {4911ms} MOEdge>>displayOn:
          ...
```

Domain

http://scg.unibe.ch/research/bifrost/metaspy

Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-Specific Profiling. In Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11), LNCS 6705 p. 68—82, Springer-Verlag, June 2011.

Domain-specific Profiling



Profile

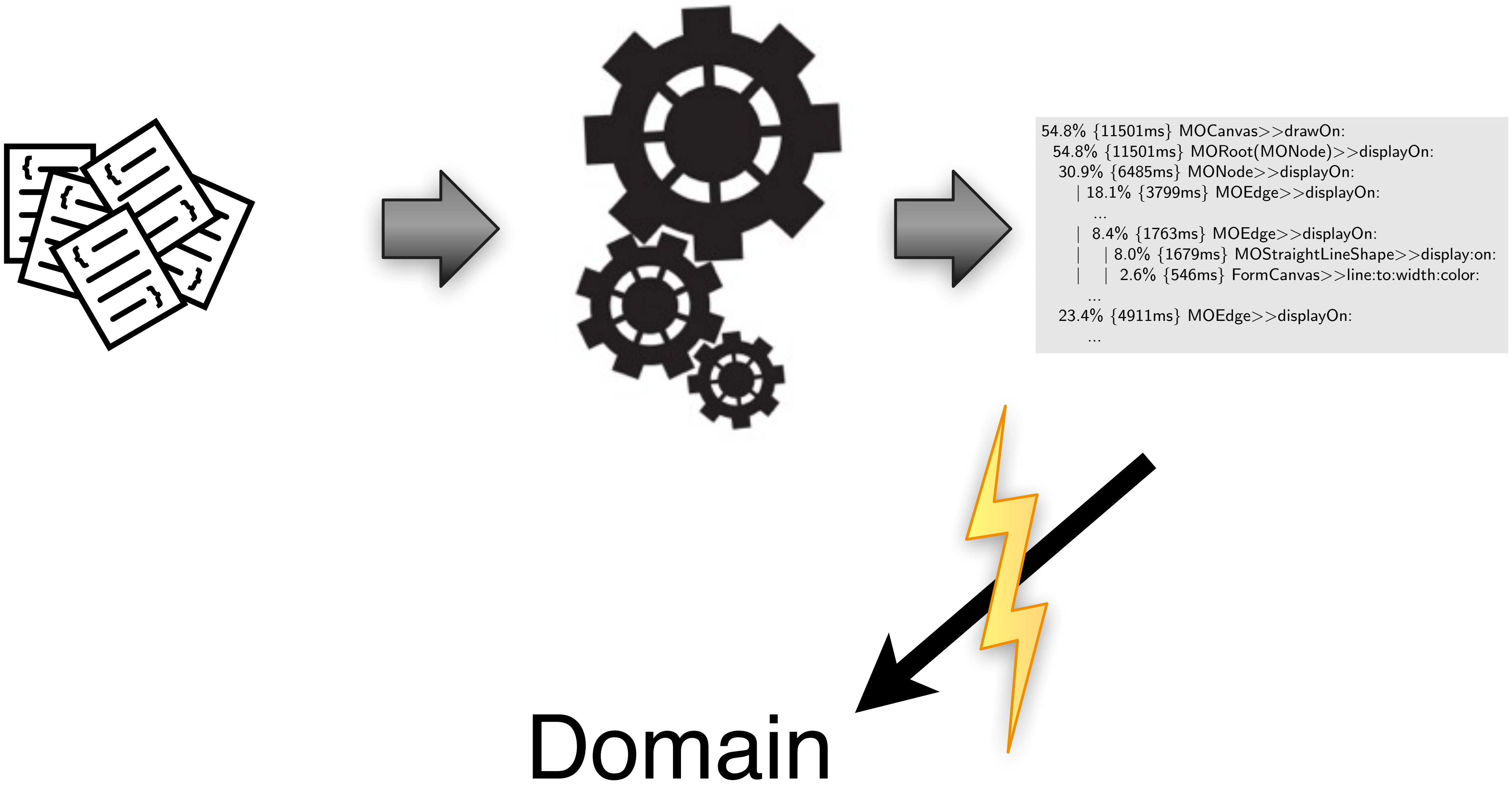
```
54.8% {11501ms} MOCanvas>>drawOn:
  54.8% {11501ms} MORoot(MONode)>>displayOn:
   30.9% {6485ms} MONode>>displayOn:
      | 18.1% {3799ms} MOEdge>>displayOn:
         ...
      |  8.4% {1763ms} MOEdge>>displayOn:
      |    | 8.0% {1679ms} MOStraightLineShape>>display:on:
      |    |  2.6% {546ms} FormCanvas>>line:to:width:color:
        ...
   23.4% {4911ms} MOEdge>>displayOn:
        ...
```

Domain

http://scg.unibe.ch/research/bifrost/metaspy

Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-Specific Profiling. In Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11), LNCS 6705 p. 68—82, Springer-Verlag, June 2011.

Domain-specific Profiling



Profile

```
54.8% {11501ms} MOCanvas>>drawOn:
  54.8% {11501ms} MORoot(MONode)>>displayOn:
   30.9% {6485ms} MONode>>displayOn:
      | 18.1% {3799ms} MOEdge>>displayOn:
         ...
      |  8.4% {1763ms} MOEdge>>displayOn:
      |    | 8.0% {1679ms} MOStraightLineShape>>display:on:
      |    |  2.6% {546ms} FormCanvas>>line:to:width:color:
        ...
   23.4% {4911ms} MOEdge>>displayOn:
        ...
```

Domain

http://scg.unibe.ch/research/bifrost/metaspy

Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-Specific Profiling. In Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11), LNCS 6705 p. 68—82, Springer-Verlag, June 2011.

Domain-specific Profiling

Profile

http://scg.unibe.ch/research/bifrost/metaspy

Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-Specific Profiling. In Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11), LNCS 6705 p. 68—82, Springer-Verlag, June 2011.

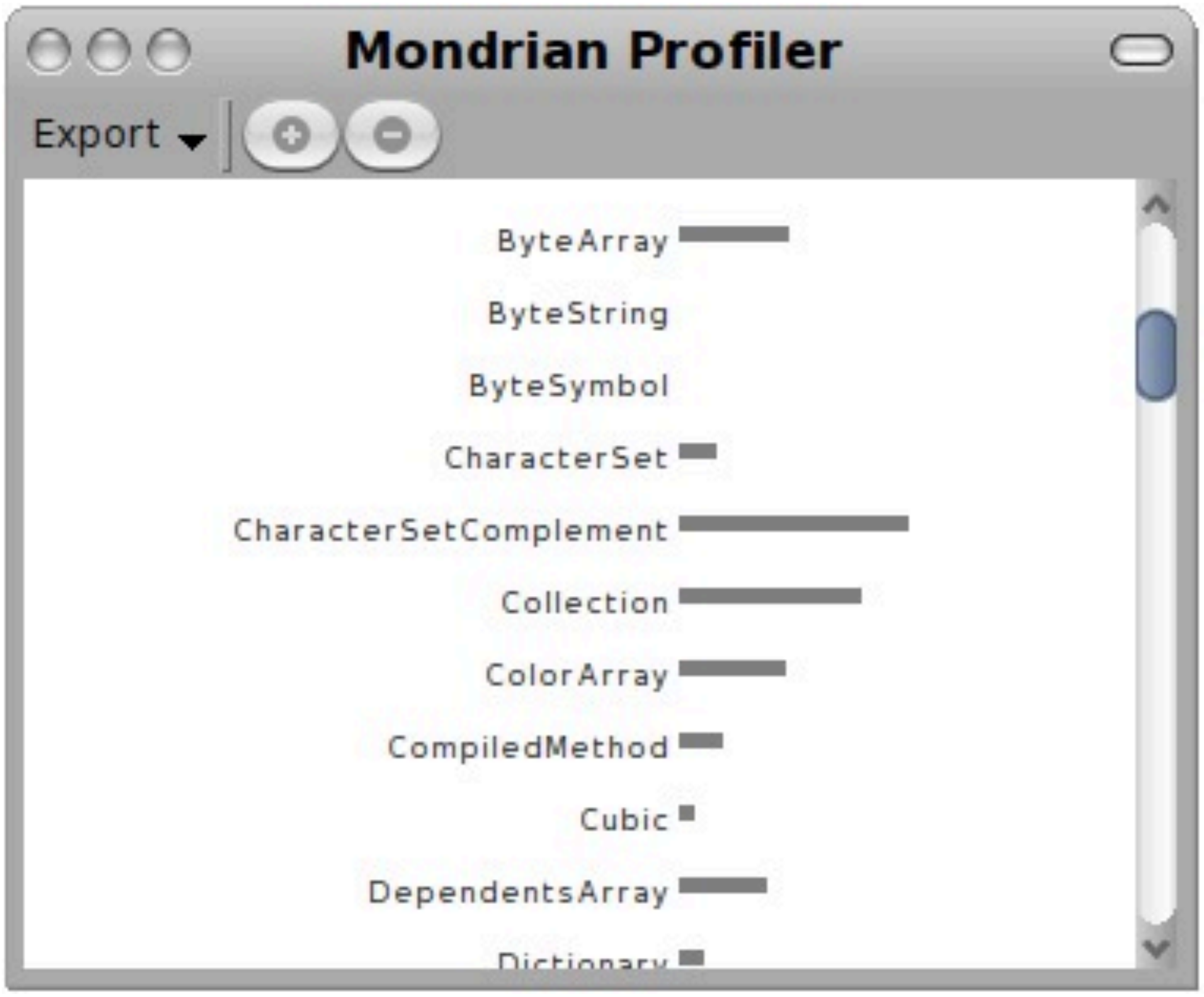
Domain-specific Profiling

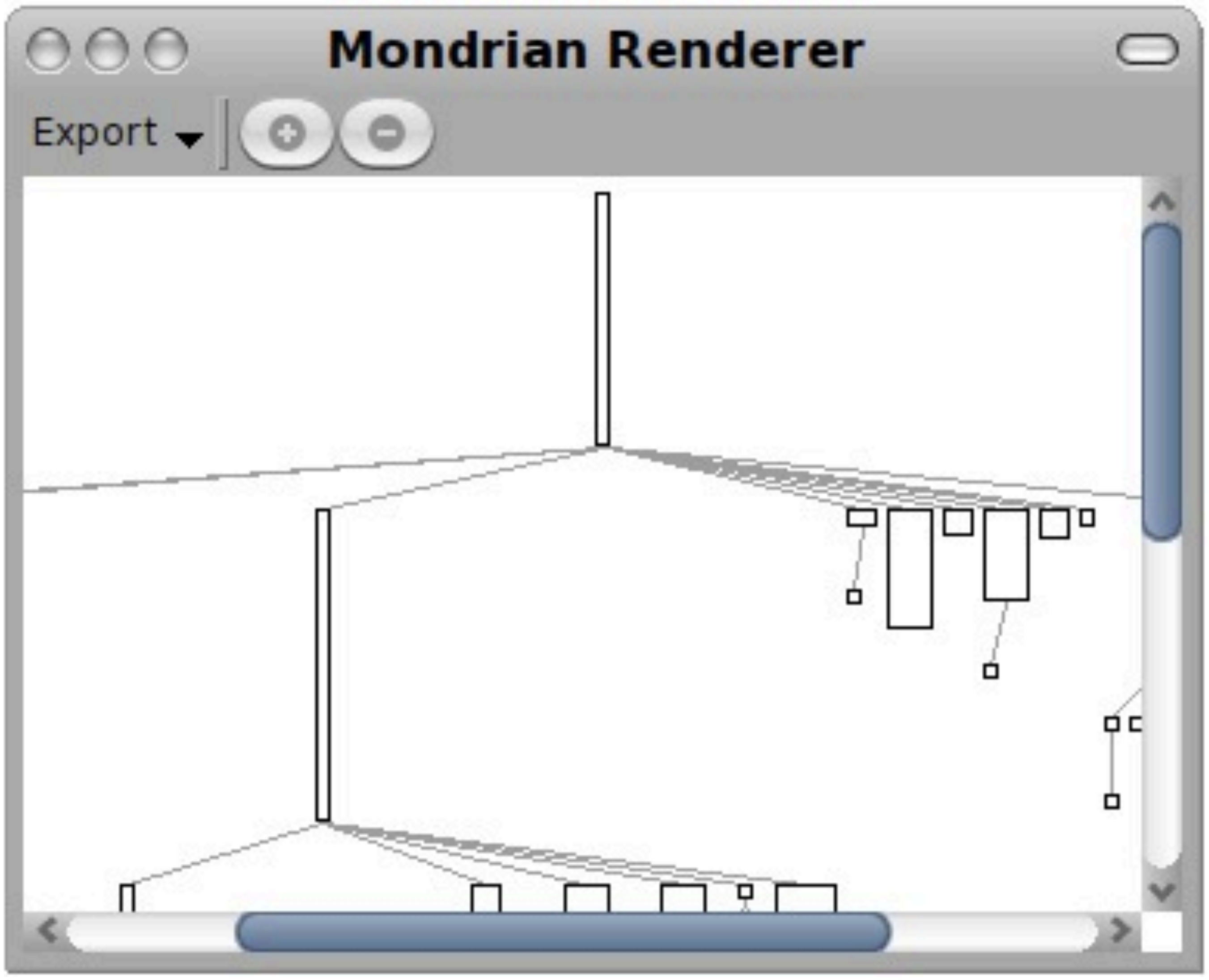
Profile

http://scg.unibe.ch/research/bifrost/metaspy

Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-Specific Profiling. In Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11), LNCS 6705 p. 68—82, Springer-Verlag, June 2011.

Domain-specific Profiling





Paradox

We claim to be doing dynamic analysis but we keep on going back to the static abstractions.

For dynamic languages the Dilemma is even worst. We are happy to have a dynamic environment like Smalltalk but, in certain way, we are trapped using the static abstractions when we should use the dynamic ones.

Roadmap

- > Motivation
- > Sources of Runtime Information
- > Dynamic Analysis Techniques
- > Advanced Dynamic Analysis Techniques
- > Dynamic analysis in a Reverse Engineering Context
- > What can we achieve with all this?
- > **Conclusion**

Dynamic vs. Static Analysis

Static analyses extract properties that hold for *all possible* program runs

Dynamic analysis provides more precise information

…but only for the execution under consideration

Dynamic analysis cannot show that a program satisfies a particular property, but can detect *violations* of the property

Conclusions: Pros and Cons

Dependent on input

- —Advantage: Input or features can be directly related to execution
- —Disadvantage: May fail to exercise certain important paths and poor choice of input may be unrepresentative

Broad scope: dynamic analyses follow long paths and may discover semantic dependencies between program entities widely separated in space and time

However, understanding dynamic behavior of OO systems is difficult

- Large number of executed methods
- Execution paths crosscut abstraction layers
- Side effects