Software Design and Evolution

10. Dynamic Analysis

Nevena Milojković

Based on the slides of Jorge Ressia

Roadmap

- Motivation
- Sources of Runtime Information
- Dynamic Analysis Techniques
- Dynamic Analysis in a Reverse Engineering Context
- The Purpose of Dynamic Analysis
- Conclusion

Roadmap

- > **Motivation**
- > Sources of Runtime Information
- > Dynamic Analysis Techniques
- > Dynamic analysis in a Reverse Engineering Context
- > The Purpose of Dynamic Analysis
- > Conclusion

What does this class do?

```
package org.jhotdraw.standard;public class CreationTool
extends AbstractTool {private List fAddedFigures;private
Figure fCreatedFigure; private Figure myAddedFigure; private
Figure myPrototypeFigure; public CreationTool(DrawingEditor
newDrawingEditor, Figure prototype) {super(newDrawingEditor);
setPrototypeFigure(prototype);} protected CreationTool
(DrawingEditor newDrawingEditor) {this(newDrawingEditor,
null);} public void activate() {super.activate(); if
(isUsable()) {getActiveView().setCursor(new AWTCursor
(java.awt.Cursor.CROSSHAIR_CURSOR)); setAddedFigures
(CollectionsFactory.current().createList());}public void
deactivate() {setCreatedFigure(null); setAddedFigure(null);
setAddedFigures(null); super.deactivate();} public void
mouseDown(MouseEvent e, int x, int y) {super.mouseDown(e, x,
y); setCreatedFigure(createFigure()); setAddedFigure
(getActiveView().add(getCreatedFigure()));
getAddedFigure().displayBox(new Point(getAnchorX(),
getAnchorY()), new Point(getAnchorX(), getAnchorY()));}
protected Figure createFigure() {if (getPrototypeFigure() ==
null) {throw new JHotDrawRuntimeException("No protoype
defined");}return (Figure)getPrototypeFigure().clone();}}
```

CreationTool class from JHotDraw system

What does this class do?

UNIVERSITÄT BERN

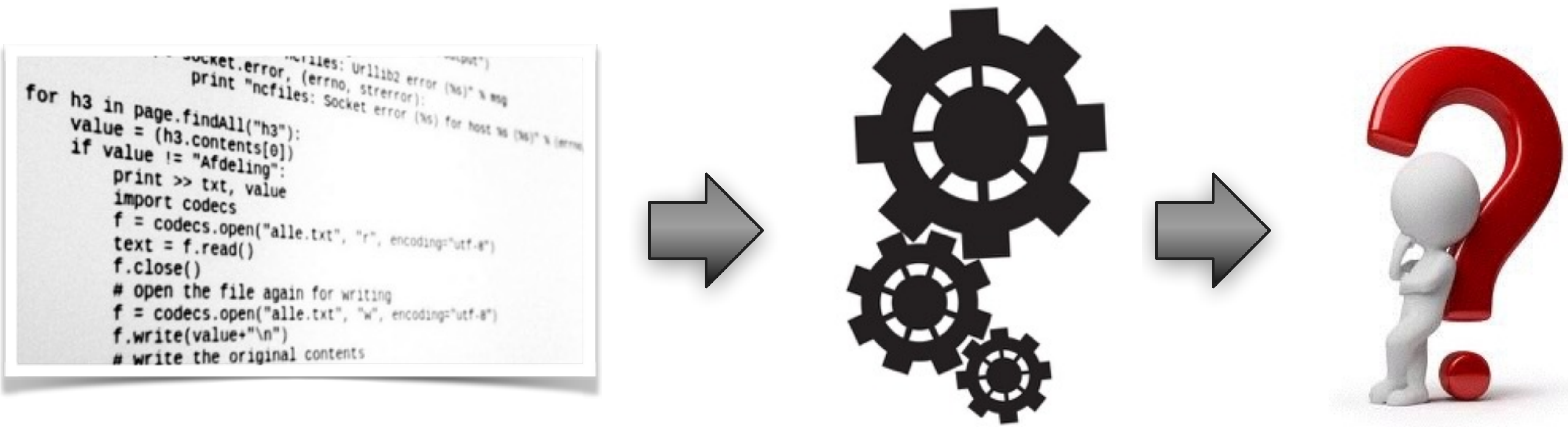
```
public abstract class AbstractFigure implements Figure {


    private transient FigureChangeListener fListener;
    private List myDependendFigures;
    private static final long serialVersionUID = -10857585979273442L;
    private int abstractFigureSerializedDataVersion = 1;
    private int _nZ;

    protected AbstractFigure() {
        myDependendFigures = CollectionsFactory.current().createList();
    }

    public void moveBy(int dx, int dy) {
        willChange();
        basicMoveBy(dx, dy);
        changed();
    }

    protected abstract void basicMoveBy(int dx, int dy);

    public void displayBox(Point origin, Point corner) {
        willChange();
        basicDisplayBox(origin, corner);
        changed();
    }

    public abstract void basicDisplayBox(Point origin, Point corner);

    public abstract Rectangle displayBox();

    public abstract HandleEnumeration handles();

    public FigureEnumeration figures() {
        return FigureEnumerator.getEmptyEnumeration();
    } …}
```

Finding Features

Software Feature:
A distinguishing characteristic of a software item.

IEEE 829

Finding Features



What is Dynamic Analysis?

UNIVERSITÄT BERN

Dynamic analysis is the investigation of the properties of a software system during run-time.

Static analysis examines the program code alone.

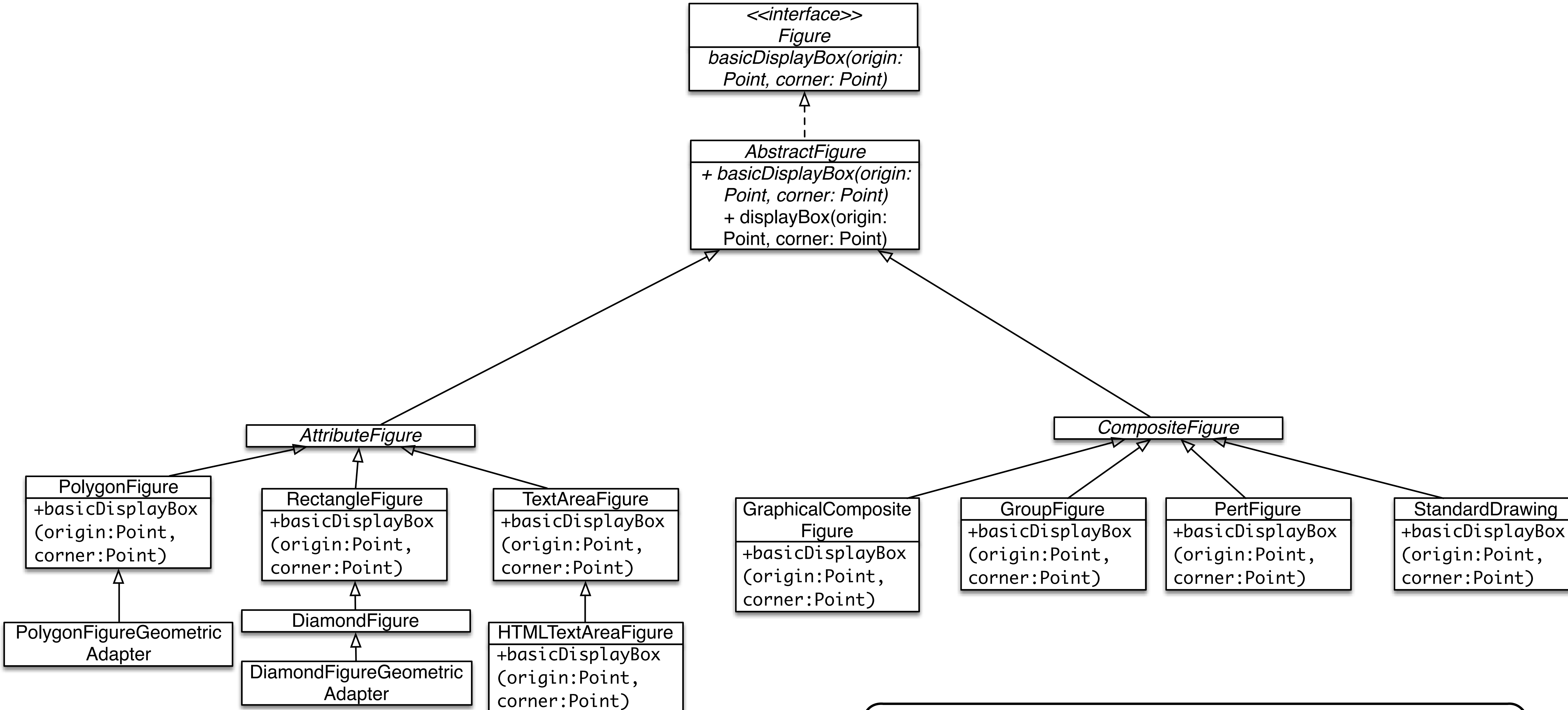
Properties of a software system are represented by the system behaviour

System behaviour is established by methods

UNIVERSITÄT BERN

Why is static analysis not enough?

UML example from JHotDraw




```
new TextAreaFigure();
new GroupFigure();
…
Figure f = fe.nextFigure();
f.basicDisplayBox(partOrigin, corner);
```

Different static analysis techniques

CHA

RTA

CTA

MTA

FTA

XTA

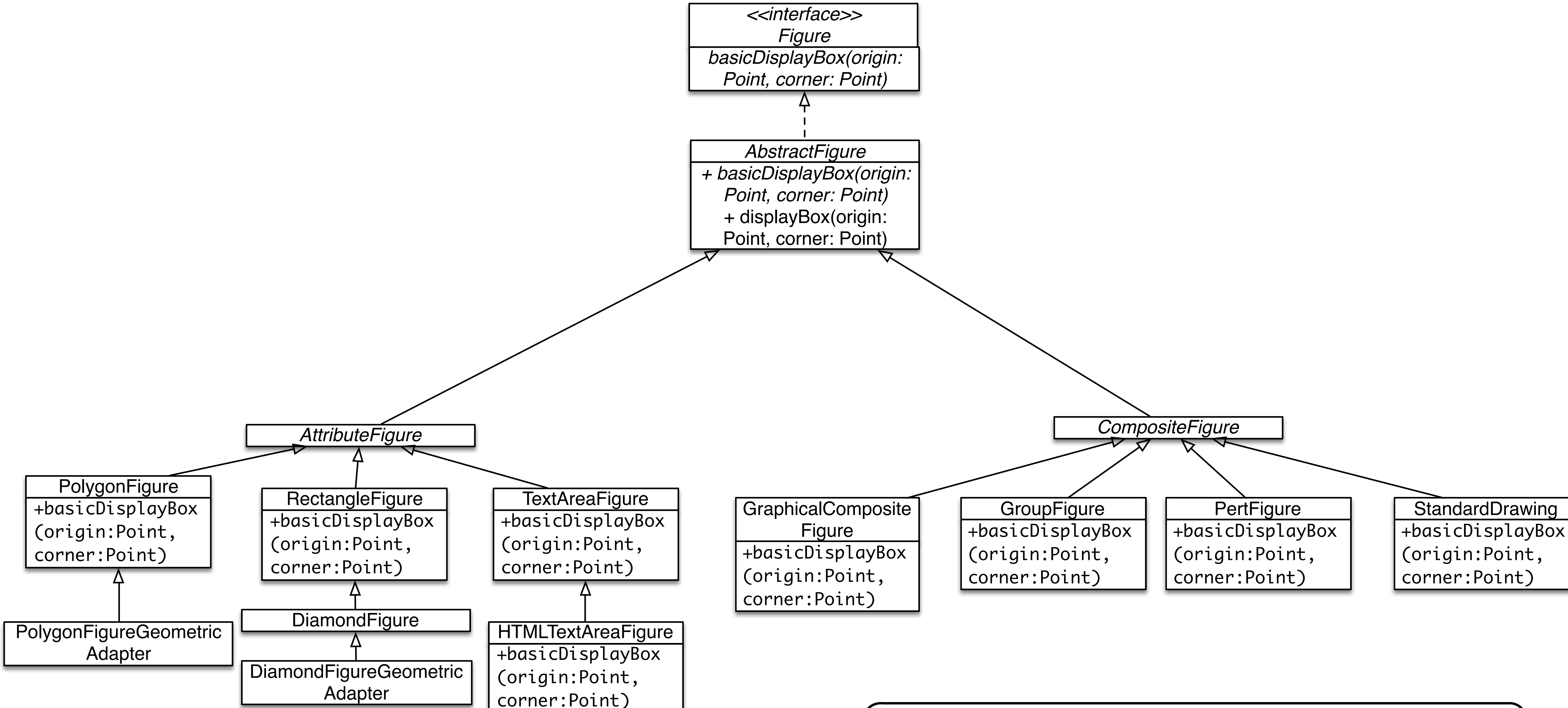
k-CFA

```
new TextAreaFigure();
new GroupFigure();
…
Figure f = fe.nextFigure();
f.basicDisplayBox(partOrigin, corner);
```

Symbolic execution

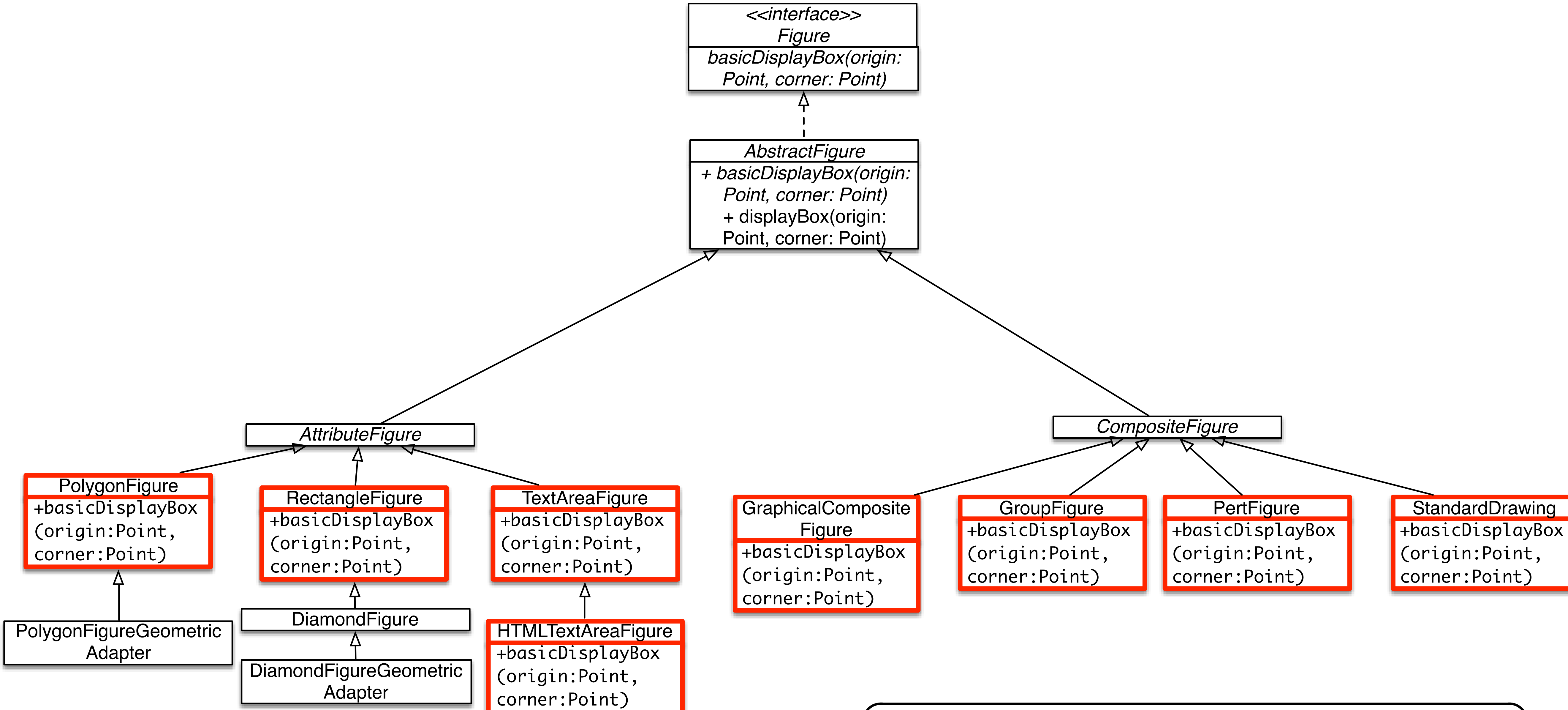
- Assigning the symbolic values to the variables, rather than concrete
- Executing the program with symbolic values, and figuring out which value causes which program path to execute

CHA algorithm



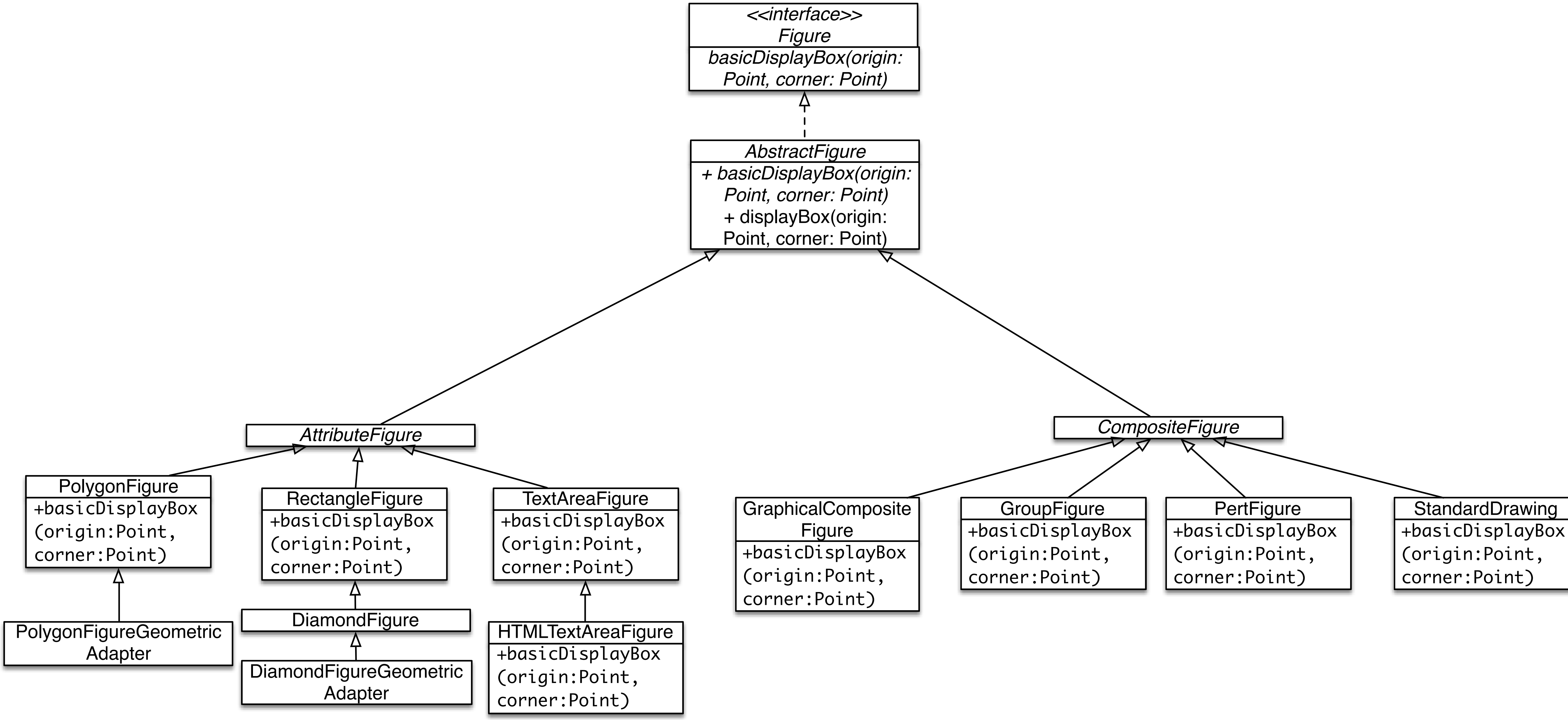

```
new TextAreaFigure();
new GroupFigure();

Figure f = fe.nextFigure();
f.basicDisplayBox(partOrigin, corner);
```

CHA algorithm



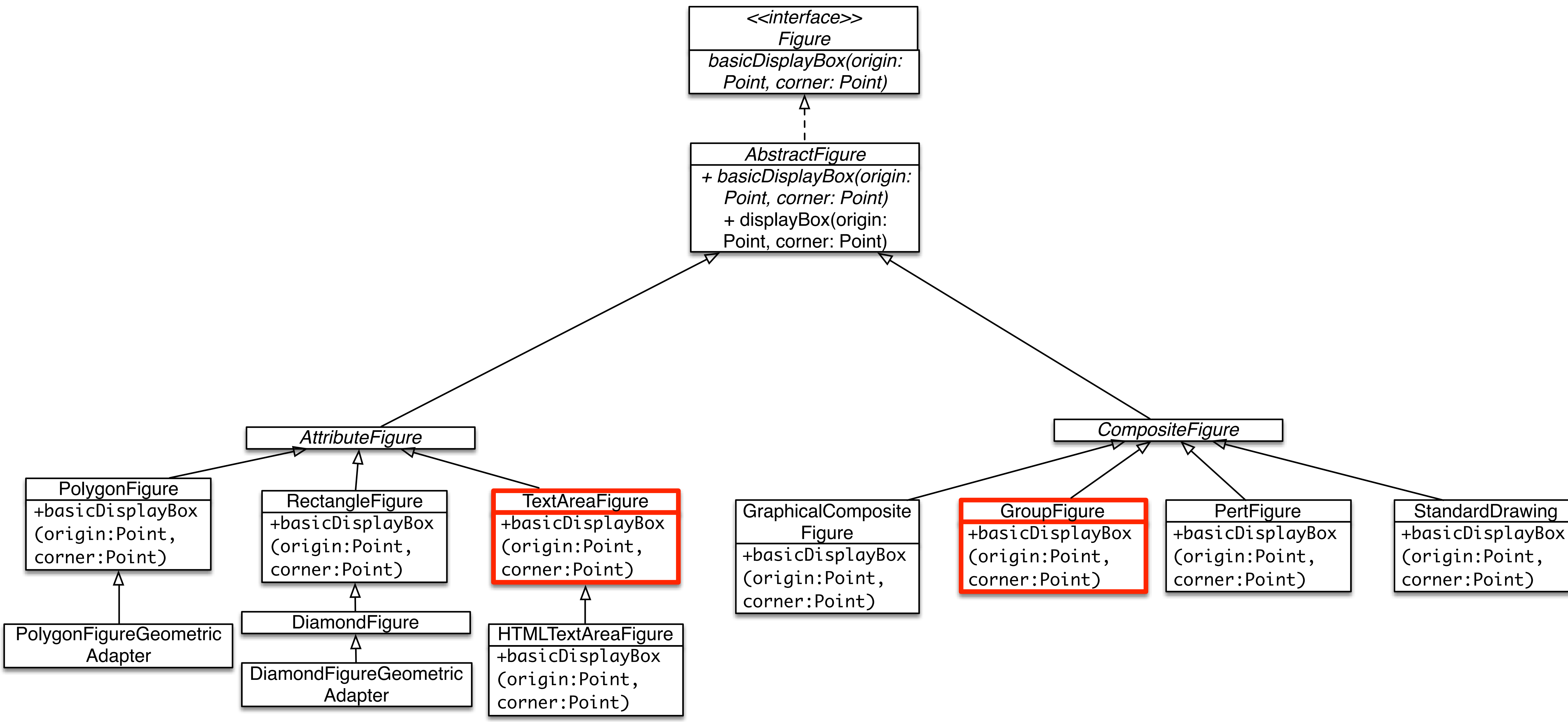

```
new TextAreaFigure();
new GroupFigure();

Figure f = fe.nextFigure();
f.basicDisplayBox(partOrigin, corner);
```

RTA algorithm




```
new TextAreaFigure();
new GroupFigure();
…
Figure f = fe.nextFigure();
f.basicDisplayBox(partOrigin, corner);
```

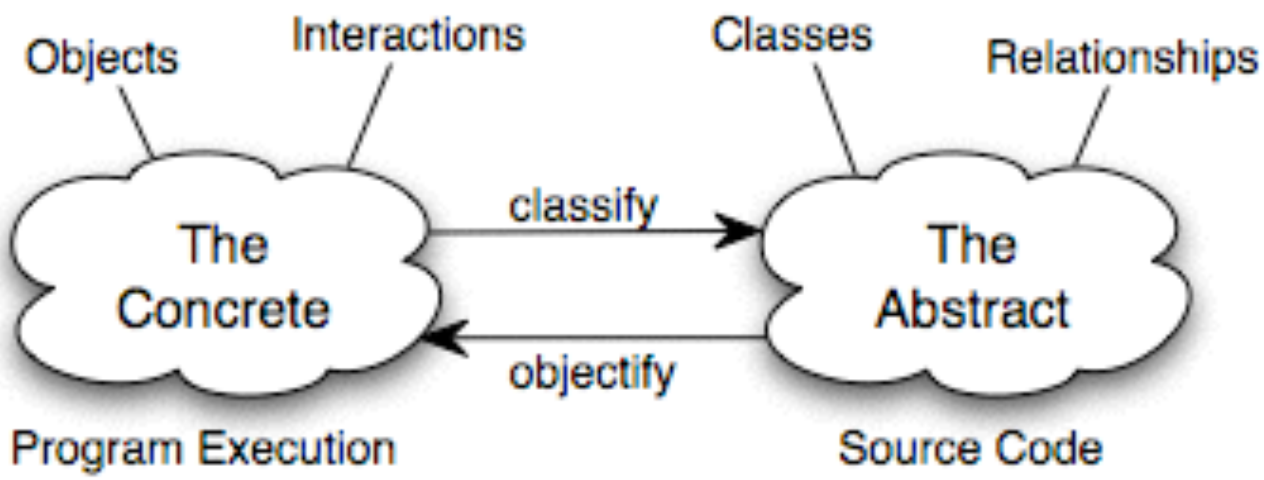
RTA algorithm




```
new TextAreaFigure();
new GroupFigure();
...
Figure f = fe.nextFigure();
f.basicDisplayBox(partOrigin, corner);
```

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

Application

- Software understanding
- Software testing

Roadmap

> Motivation
> **Sources of Runtime Information**
> Dynamic Analysis Techniques
> Dynamic analysis in a Reverse Engineering Context
> The Purpose of Dynamic Analysis
> Conclusion

Runtime Information Sources

> tracing method execution
> tracing values of variables
> tracing memory usage

Two ways of getting the information

- External
  - execute program and collect the information from outside
- Internal
  - instrument program, and get the information from inside

External View

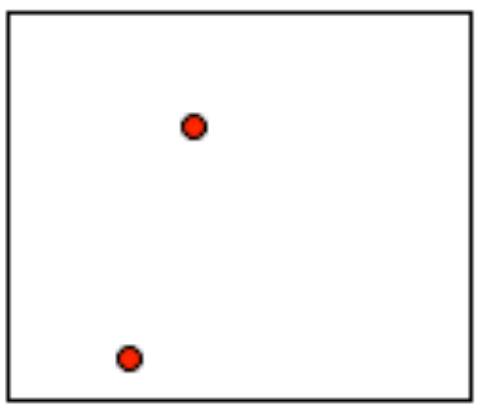
> System.out.println

> Examine logs

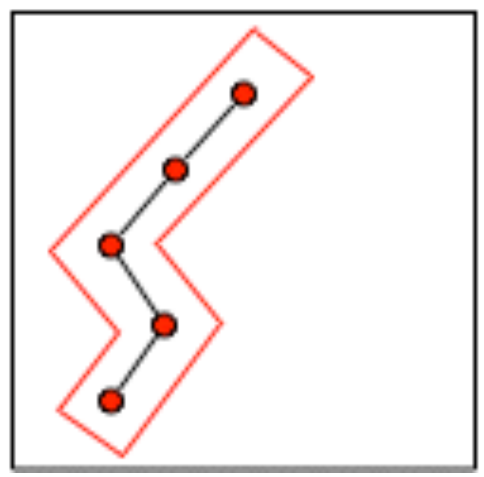
> Analyse used resources
>> CPU and memory usage
>> Open files

Internal View

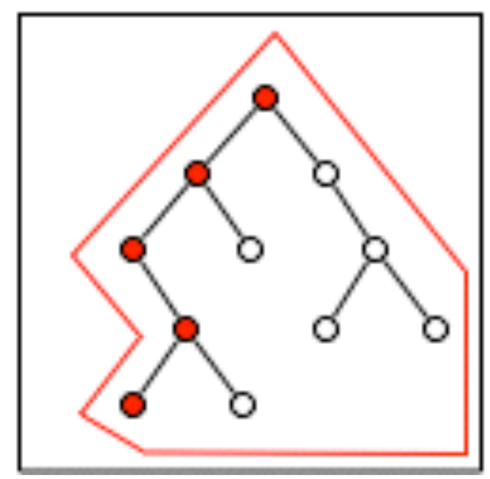
Log statements in code



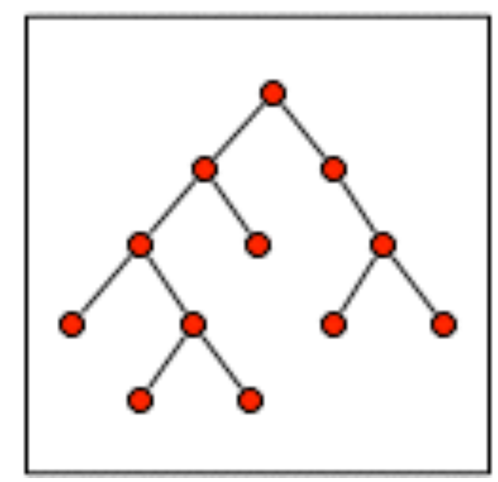
Stack trace



Debugger



Execution trace



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

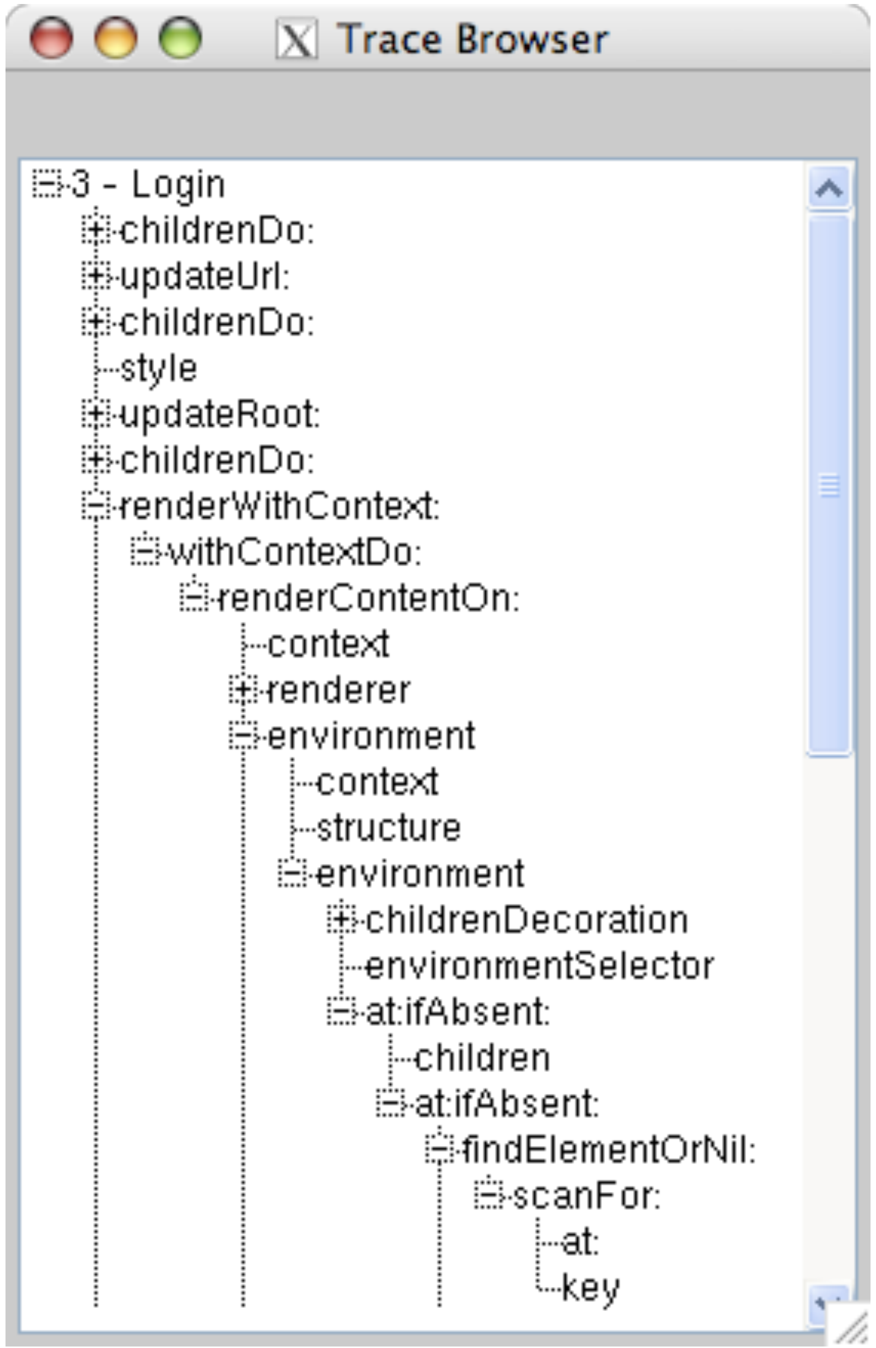
Execution Tracing

How can we capture full program execution?

Trace entry and exit of methods

Additional information:

- receiver and arguments
- return values
- fields assigning
- class instantiations



Tracing Techniques

- Instrumentation approaches
  - Source code modification
  - Byte code modification
  - Wrapping methods (Smalltalk)

- Simulate execution (using debugger infrastructure)

Technical Challenges

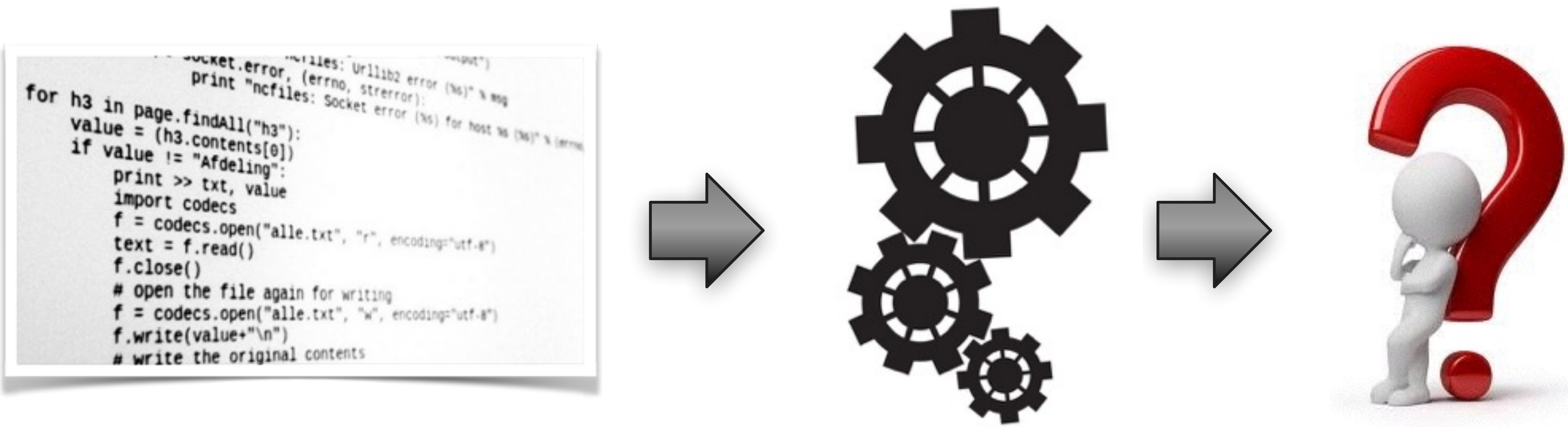
UNIVERSITÄT BERN

- Instrumentation influences the behaviour of the
    - execution
- Overhead: increased execution time
- Large amount of data

Roadmap

- Motivation
- Sources of Runtime Information
- **Dynamic Analysis Techniques**
- Dynamic analysis in a Reverse Engineering Context
- The Purpose of Dynamic Analysis
- Conclusion

Feature Analysis



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 anAlien = new Clingon();
      System.out.println("in main");
      anAlien.spendLife();
      }
}
```

Smalltalk Mechanisms

> become: function
> Method Wrappers
> Anonymous Classes

become: function

- primitive function
- swaps the object pointers of the receiver and the argument and all variables in the system that used to point to the receiver now point to the argument, and vice-versa.

Method Wrappers

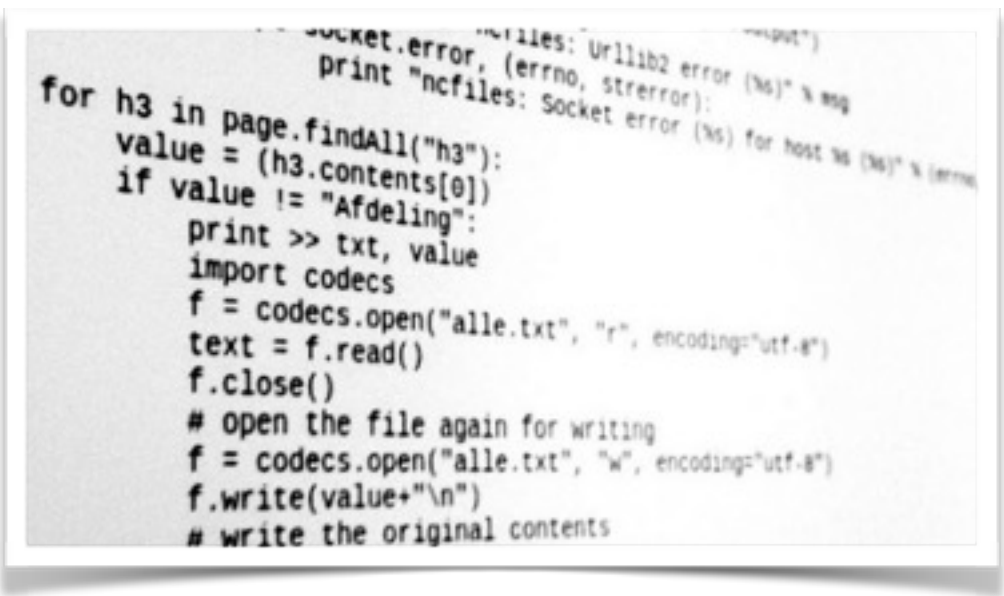
A MethodWrapper replaces an original CompiledMethod in the method dictionary of a class and wraps it by performing some before and after actions.

Anonymous Classes

UNIVERSITÄT BERN

To intercept the behaviour of the object, we need to create an anonymous subclass of its class and override a method whose behaviour we want to inspect.

Sub-method Feature Analysis



Bytecode Instrumentation

Smalltalk

Example: Number>>asInteger

> 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 literal 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

Reflectivity

UNIVERSITÄT BERN

> Reflectivity is a tool to annotate AST nodes with metalinks.

> A metalink is a message sent to an arbitrary object.

> A metalink can be executed before a node, instead a node, after a node.

http://smalltalkhub.com/#!/~RMoD/Reflectivity

ByteSurgeon

- Enables runtime bytecode transformations for Smalltalk
- Provides high-level API
- Complements the reflective ability of Smalltalk with the possibility to instrument method

- Runtime transformation needed for
  - Adaptation of running systems
  - Tracing / debugging

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'*

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

Java

Bytecode Manipulation

- Java
  - Javassist
    - *high-level API*
  - ASM
    - *working on low-level*

Javassist

UNIVERSITÄT BERN

```
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;
    }
}
```

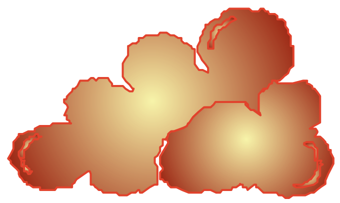
Javassist - Edit Body

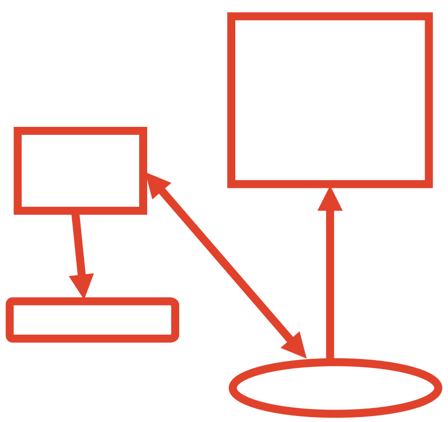
```
CtMethod cm = ... ;
cm.instrument(
    new ExprEditor() {
        public void edit(MethodCall m)
                      throws CannotCompileException
        {
            if (m.getClassName().equals("Point")
                          && m.getMethodName().equals("move"))
                m.replace("{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

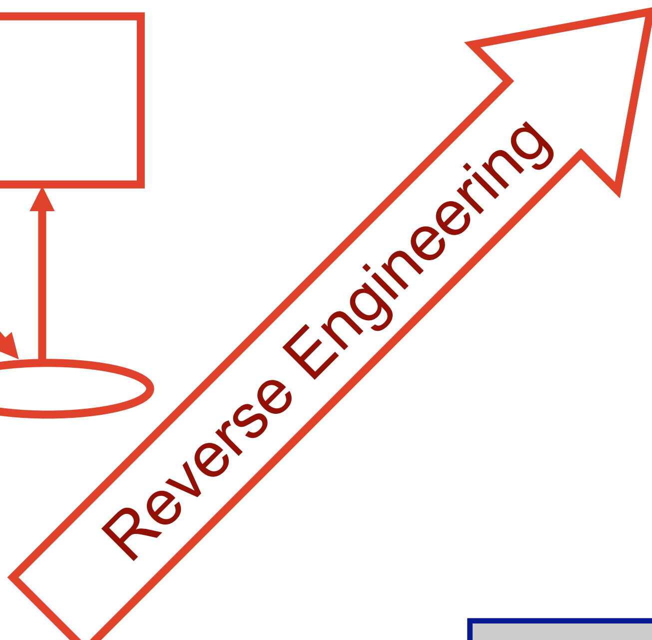
Roadmap

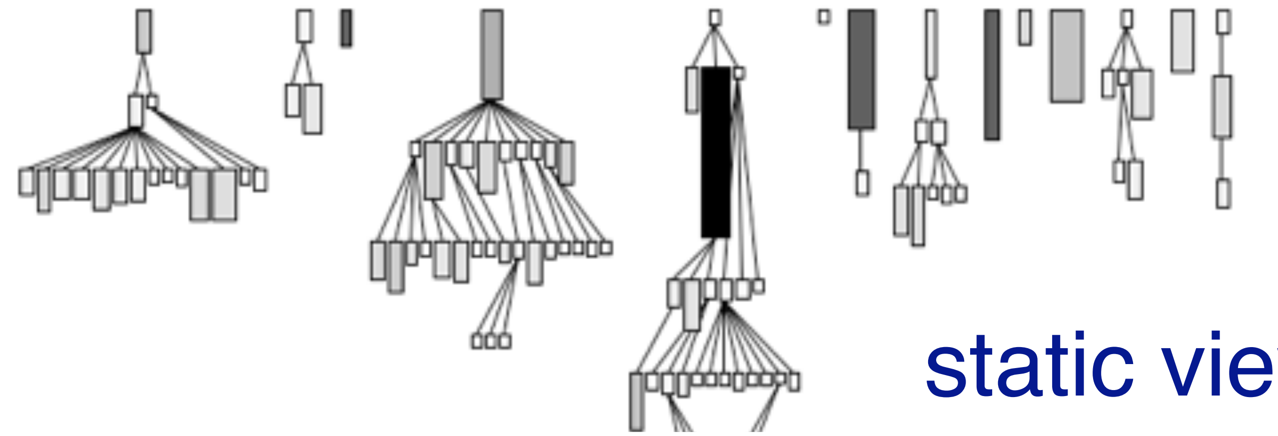
- > Motivation
- > Sources of Runtime Information
- > Dynamic Analysis Techniques
- > **Dynamic analysis in a Reverse Engineering Context**
- > The Purpose of Dynamic Analysis
- > Conclusion

Reverse Engineering + Dynamic Analysis

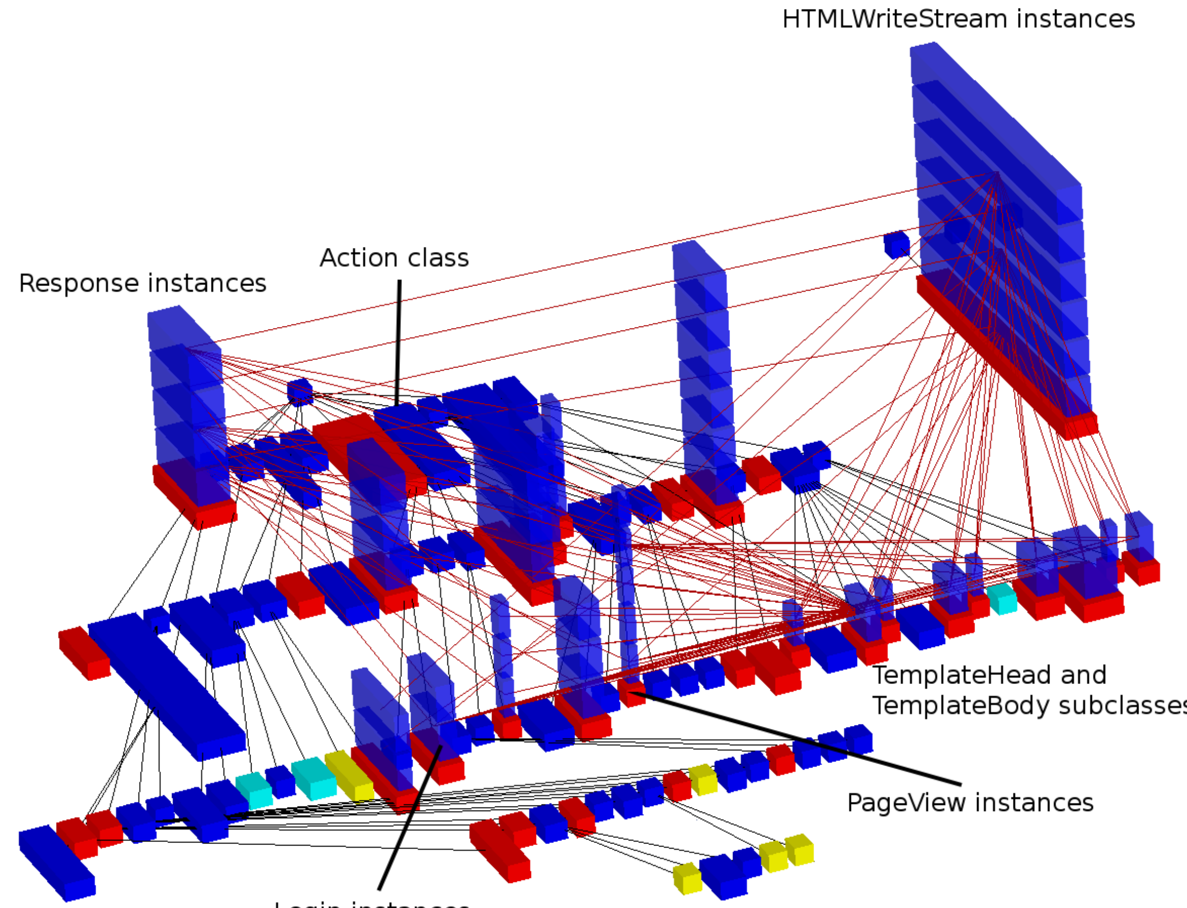








static view



+

execution traces

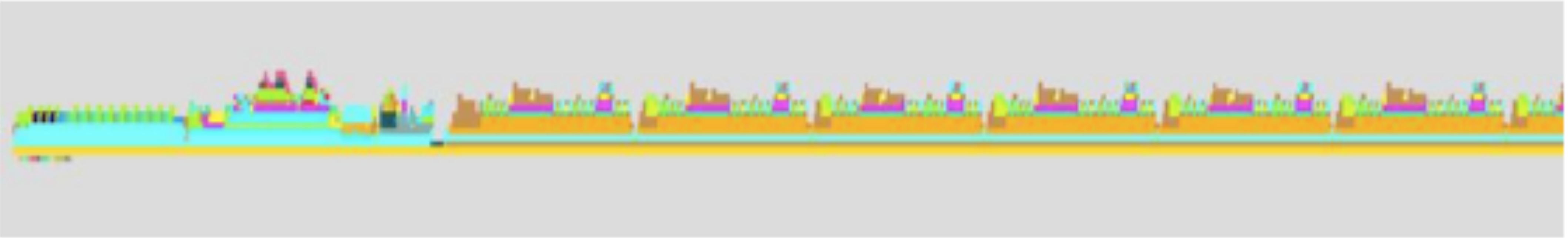
dynamic view

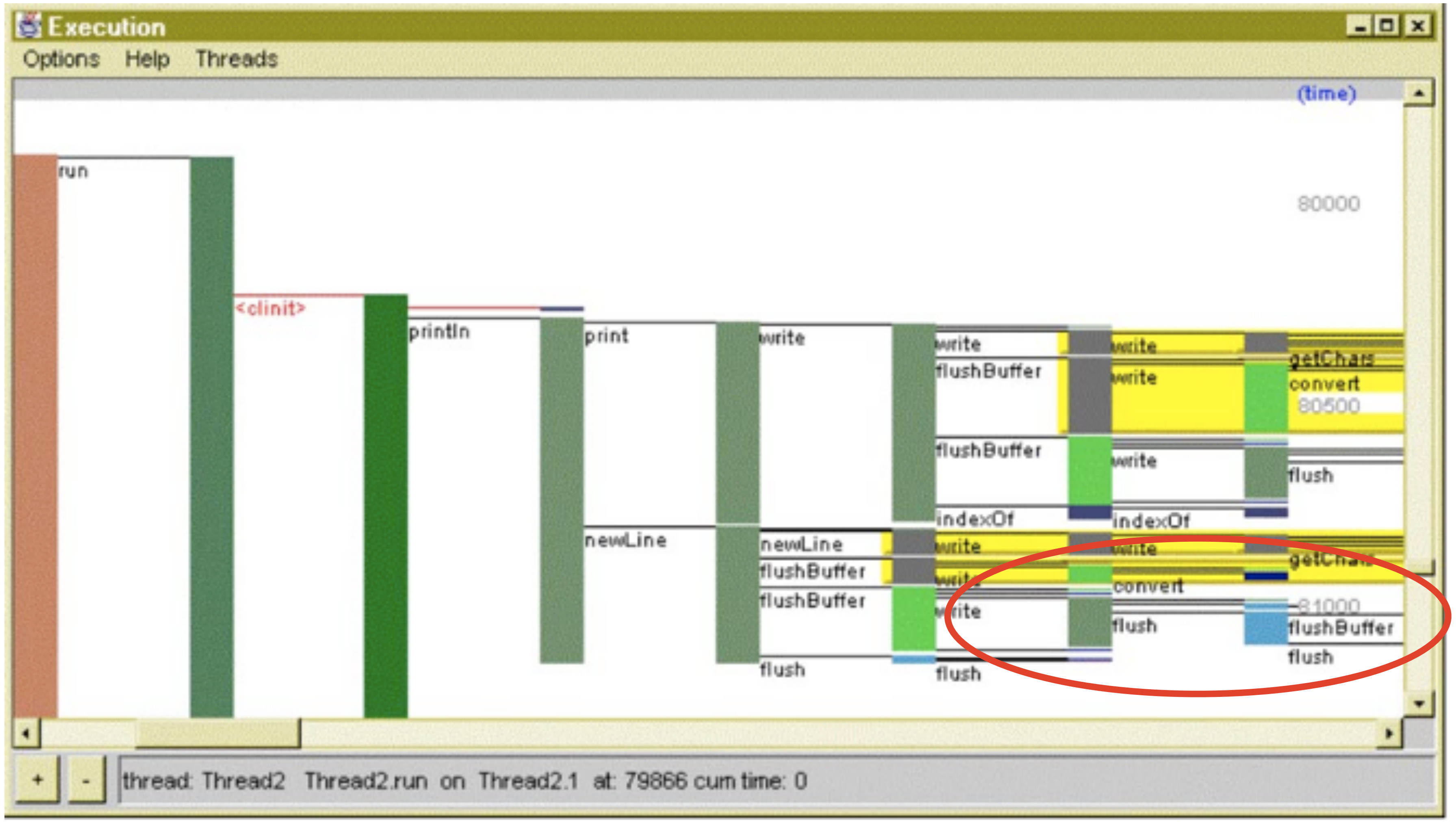
Dynamic Analysis for Program Comprehension

UNIVERSITÄT BERN

- Frequency Analysis [Ball, Zaidman]
- Runtime Coupling Metrics based on Web mining techniques to detect key classes in a trace [Zaidman 2005]
- Recovering high-level views from runtime data [Richner and Ducasse 1999]

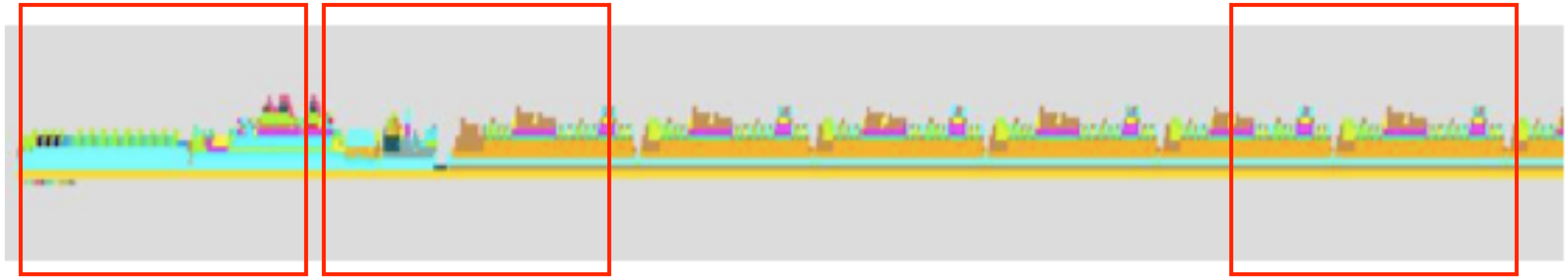
Visualization of Runtime Behaviour





[JinSight, De Pauw 1993]

Dividing a trace into features

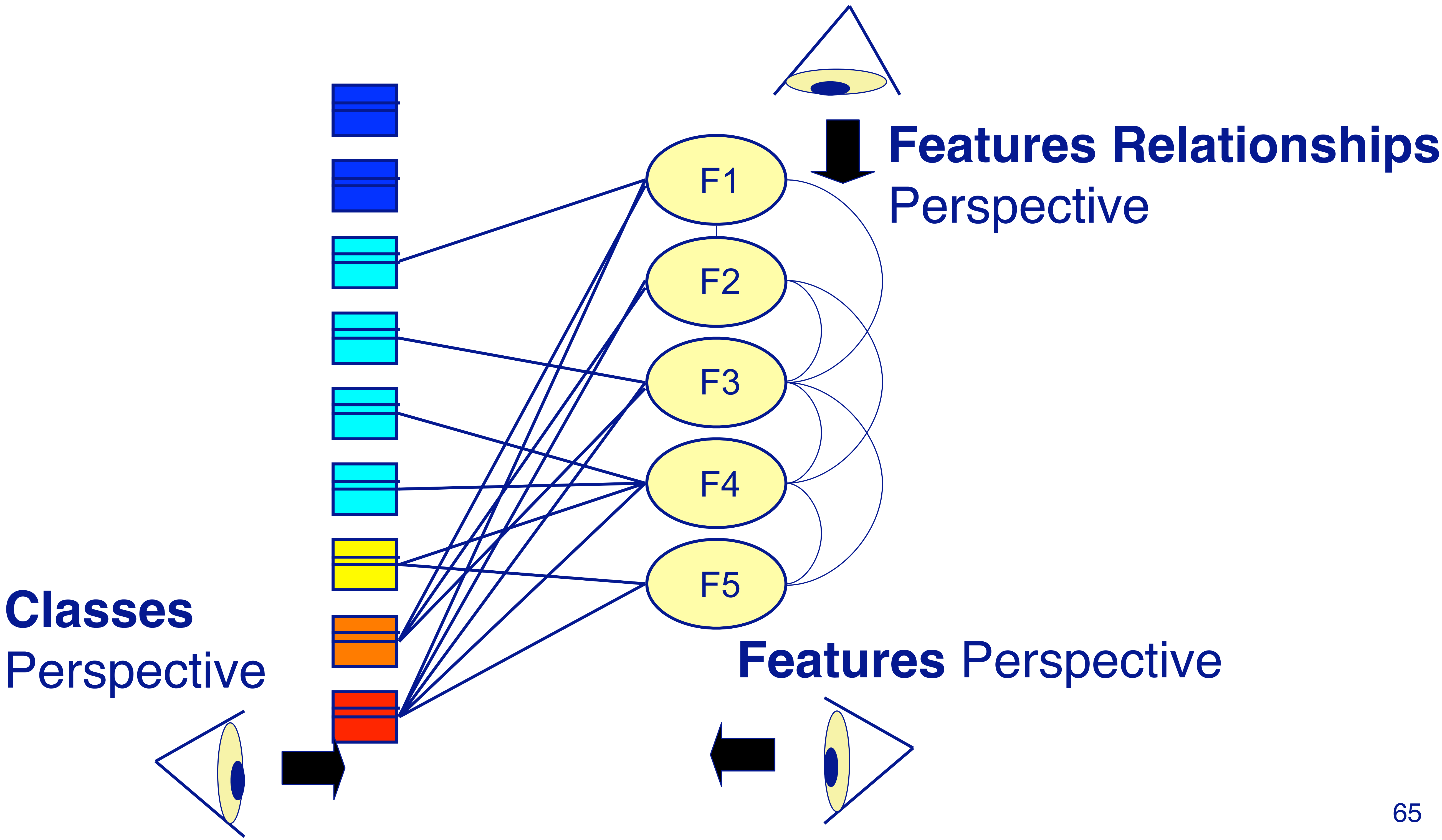


Feature 1 Feature 2 Feature n

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

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

Feature-Centric Analysis: 3 Complementary Perspectives



Object Flow Analysis

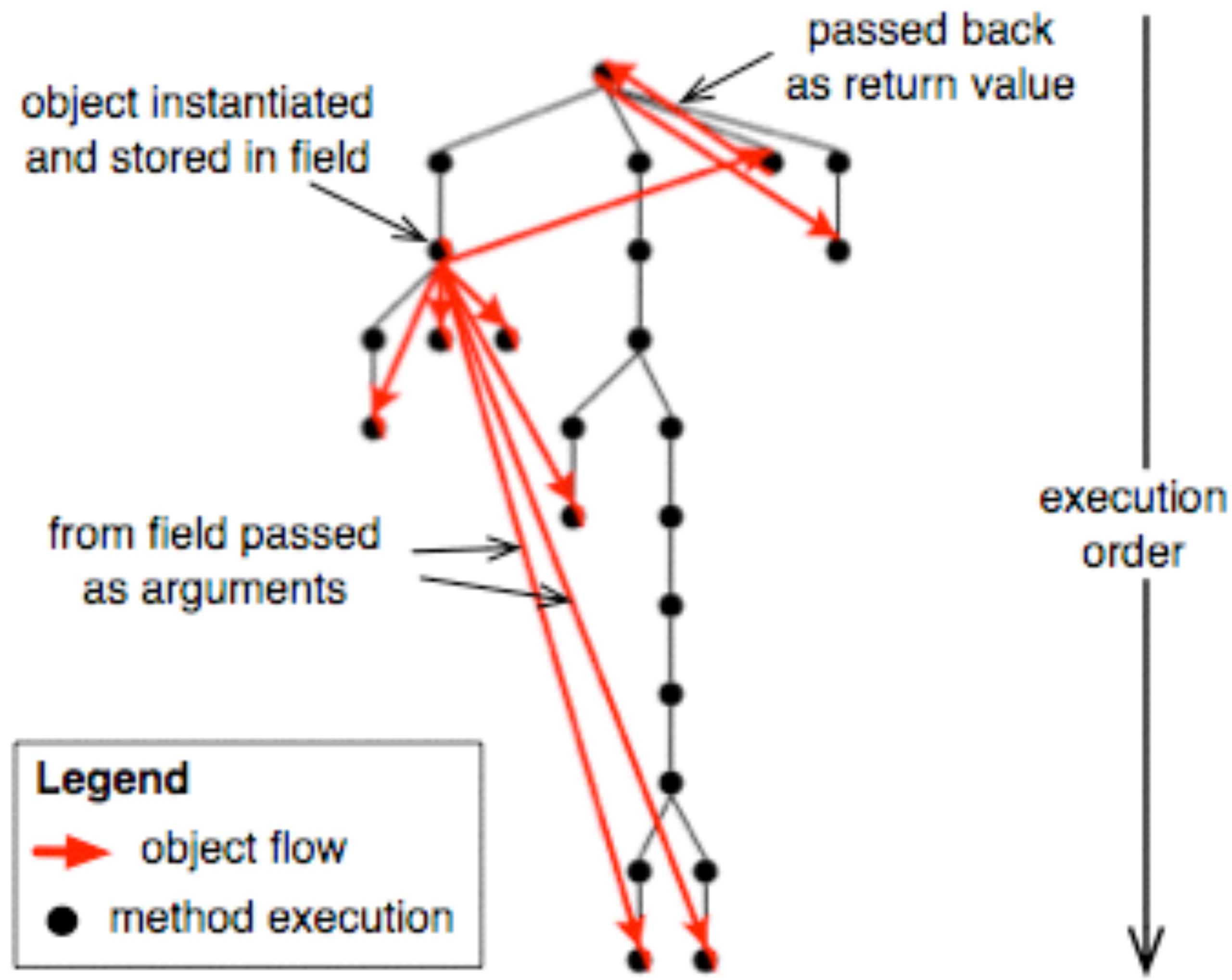
Method execution traces do not reveal how

… objects refer to each other

… object references evolve

Trace and analyse object flow

- Detect object dependencies between features



Roadmap

- Motivation
- Sources of Runtime Information
- Dynamic Analysis Techniques
- Dynamic analysis in a Reverse Engineering Context
- **The Purpose of Dynamic Analysis**
- Conclusion

- Augment the notion of developers with run-time information
- Traditional IDEs lack information about sometimes purely dynamic relationship between source code artefacts
- The lack of the dynamic type of the receiver is one of the biggest obstacles in program comprehension

Hermion

> integrates dynamic information directly in the source code
> augments the static source code with type information for variables
> shows which methods get invoked at particular call sites in source code
> aggregates its dynamic information over different runs

http://scg.unibe.ch/archive/reports/Roet08dHermion.pdf

Hermion

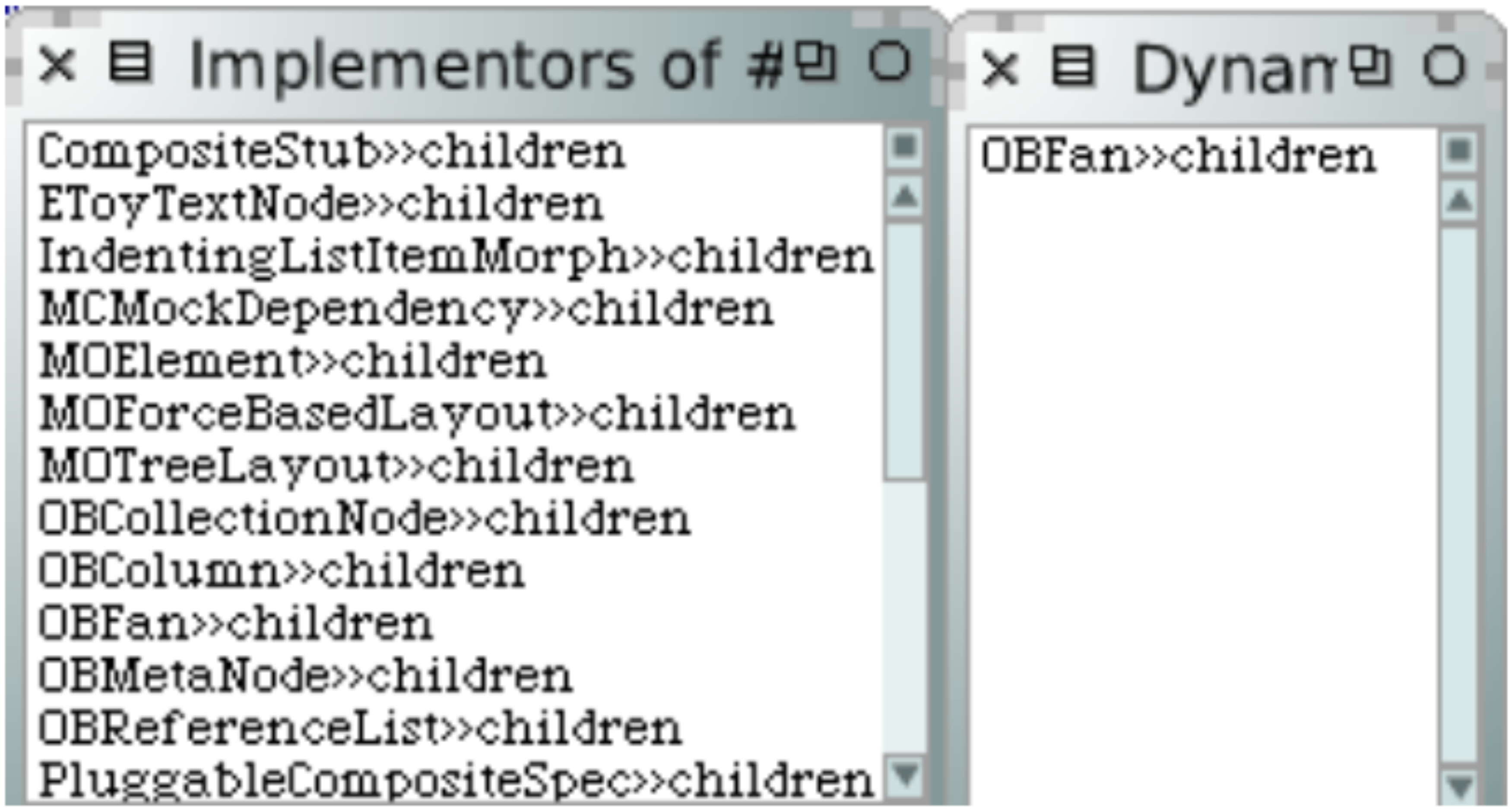


Figure 2. Static search (1) vs. precise dynamic search (2) for implementors of *children* **in** Hermion

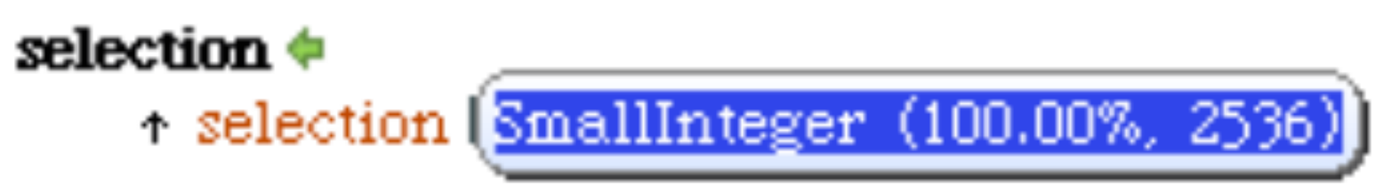
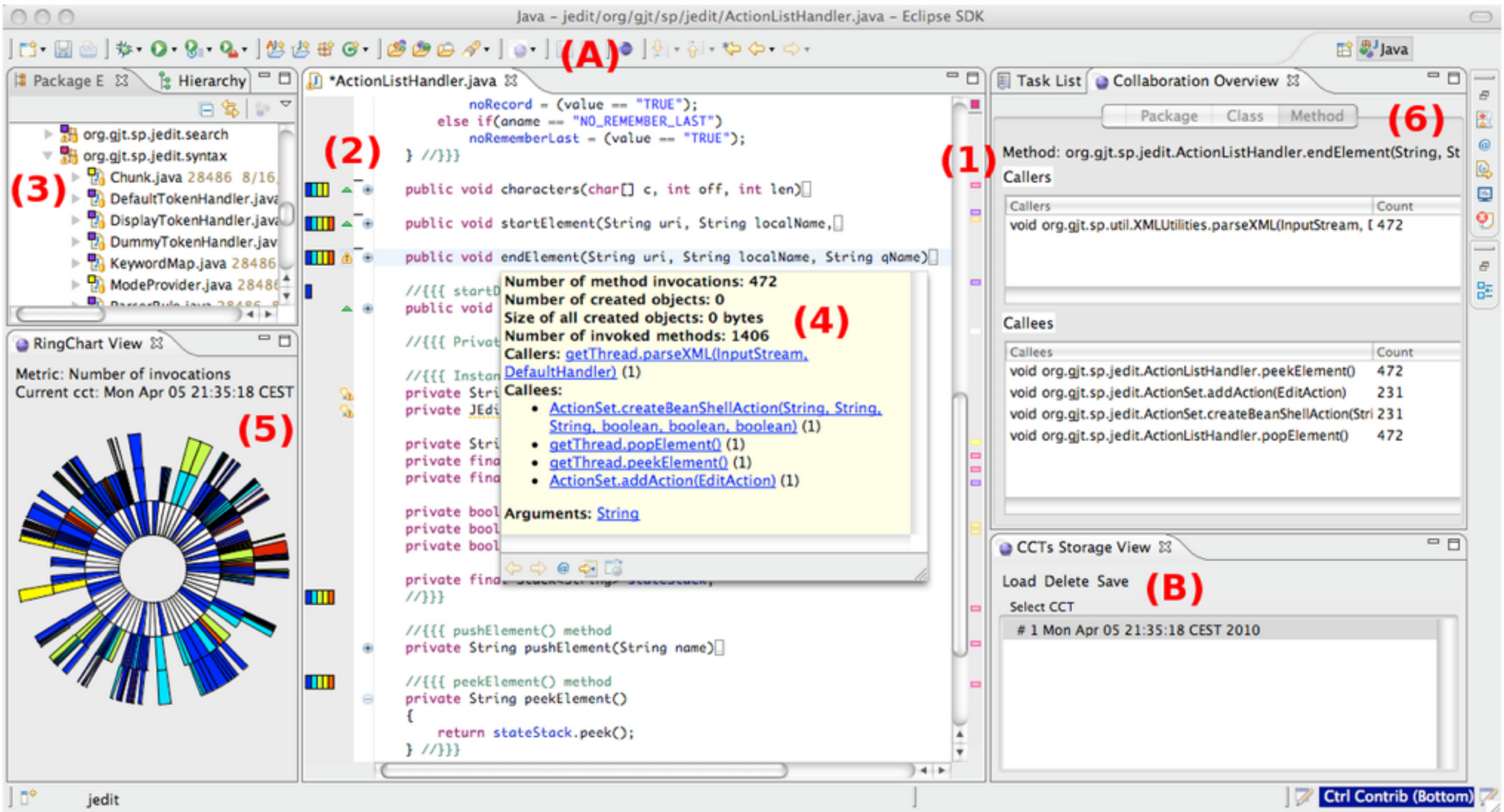


Figure 4. List of types of instance variable *selection* **extracted from dynamic information in** Hermion

Senseo



http://scg.unibe.ch/archive/papers/Roet09cSenseo.pdf

Roadmap

> Motivation
> Sources of Runtime Information
> Dynamic Analysis Techniques
> Dynamic analysis in a Reverse Engineering Context
> The Purpose of Dynamic Analysis
> **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

UNIVERSITÄT BERN

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 behaviour of OO systems is difficult

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