10. Reflection

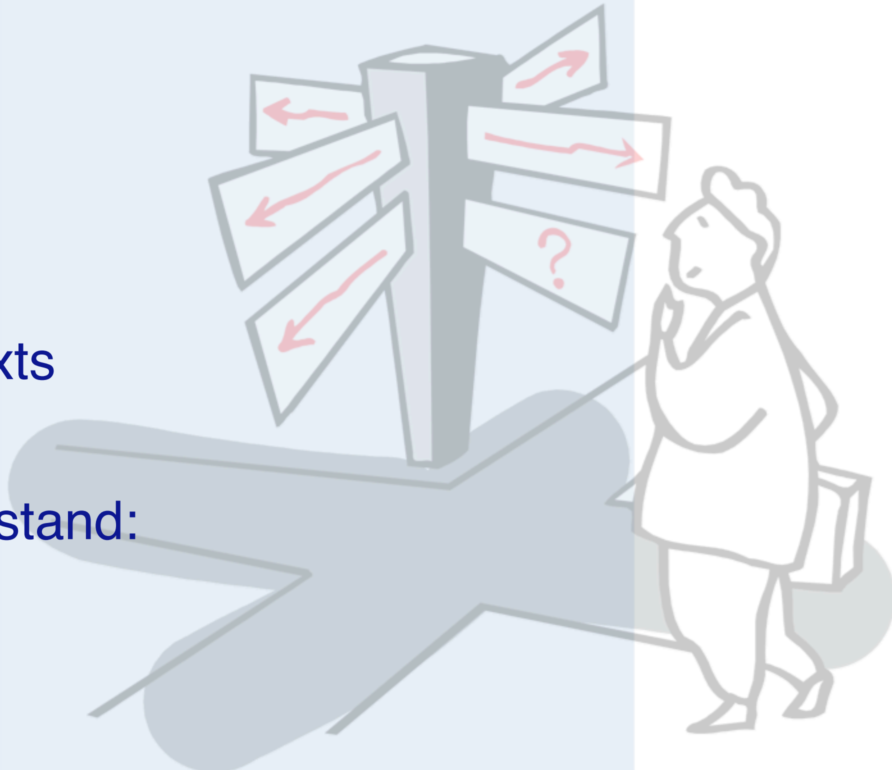
Birds-eye view


Reflection allows you to both *examine* and *alter* the meta-objects of a system.

Using reflection to modify a running system requires some care.

Roadmap

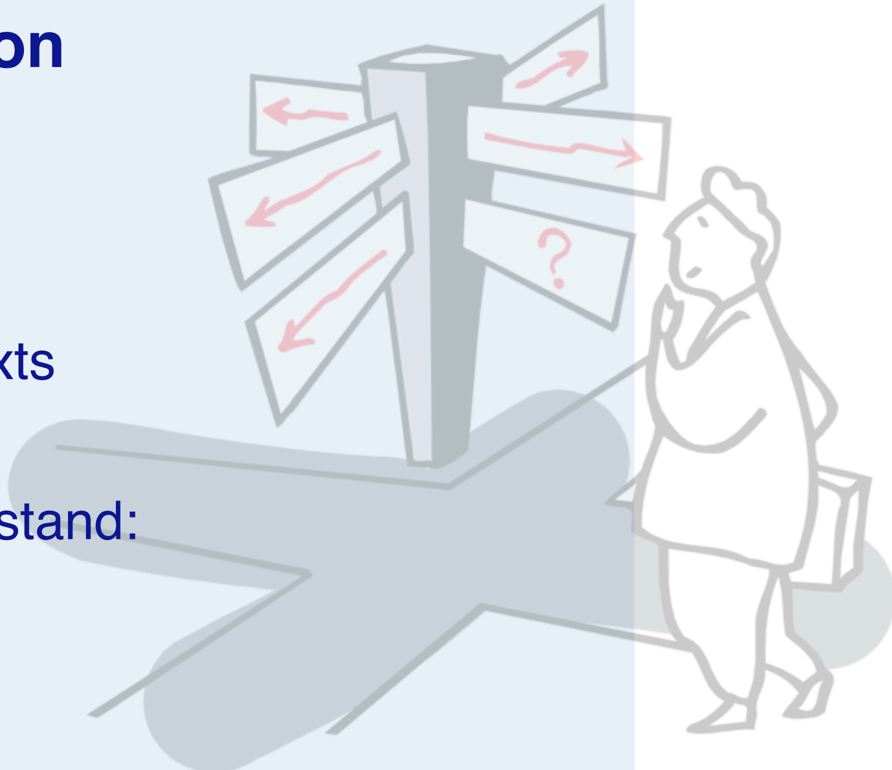
- Reification and reflection
- Introspection
  - Inspecting objects
  - Querying code
  - Accessing run-time contexts
- Intercession
  - Overriding doesNotUnderstand:
  - Anonymous classes
  - Method wrappers



Selected material by Marcus Denker and Stéphane Ducasse

Roadmap

- **Reification and reflection**
- Introspection
  - Inspecting objects
  - Querying code
  - Accessing run-time contexts
- Intercession
  - Overriding doesNotUnderstand:
  - Anonymous classes
  - Method wrappers



Selected material by Marcus Denker and Stéphane Ducasse

Why we need reflection

As a programming language becomes *higher and higher level*, its implementation in terms of underlying machine involves *more and more tradeoffs*, on the part of the implementor, about what cases to optimize at the expense of what other cases. … the *ability to cleanly integrate* something outside of the language's scope *becomes more and more limited*

Kiczales, in Paepcke 1993

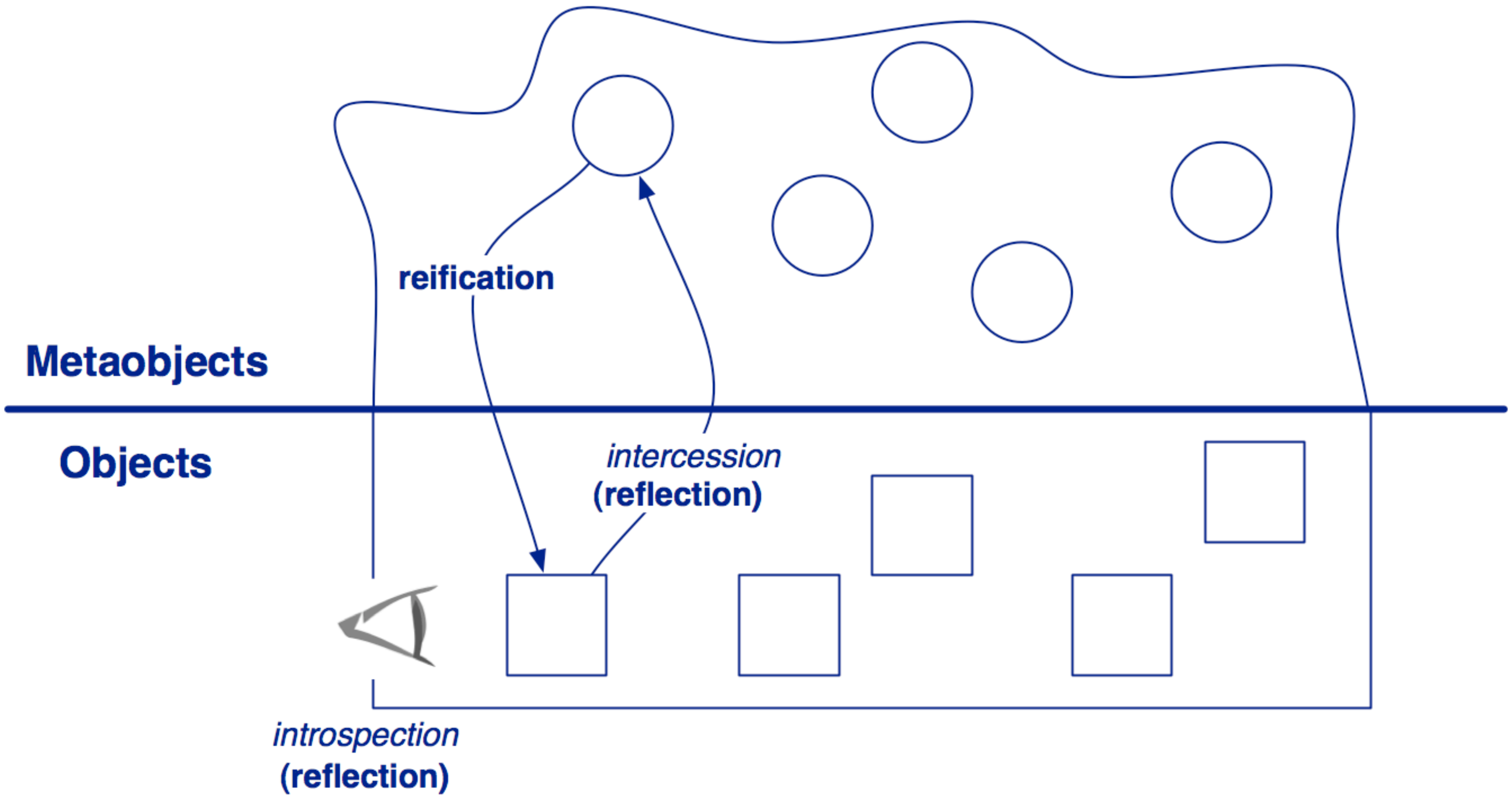
What is are Reflection and Reification?

> Reflection is the ability of a program to *manipulate as data* something representing the *state of the program* during its own execution.
  - Introspection is the ability for a program to *observe* and therefore *reason* about its own state.
  - Intercession is the ability for a program to *modify* its own execution state or *alter its own interpretation* or meaning.

> Reification is the mechanism for encoding execution state as data

— Bobrow, Gabriel & White, 1993

Reflection and Reification



Consequences

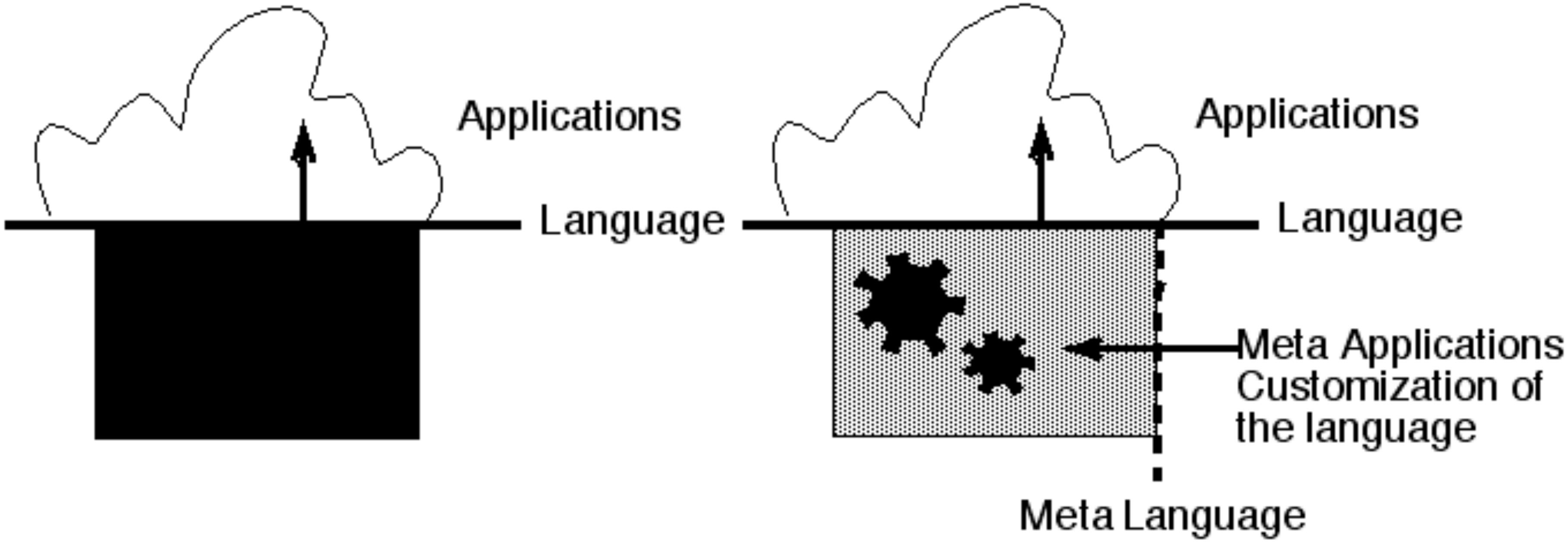
> "A system having itself as application domain and that is *causally connected* with this domain can be qualified as a reflective system"

— Maes, OOPSLA 1987

- A reflective system has an *internal representation of itself*.
- A reflective system is able to *act on itself* with the ensurance that its representation will be causally connected (up to date).
- A reflective system has some static capacity of *self-representation* and dynamic *self-modification* in constant synchronization

Metaprogramming in Programming Languages

- > The meta-language and the language can be different:
  - — Scheme and an OO language
- > The meta-language and the language can be same:
  - — Smalltalk, CLOS
  - — In such a case this is a *metacircular architecture*



Structural and behavioral reflection

> Structural reflection is concerned with the ability of the language to provide a complete *reification* of both
> — the *program* currently executed
> — as well as its *abstract data types*.

> Behavioral reflection is concerned with the ability of the language to provide a complete reification of
> — its own *semantics* and *implementation* (processor)
> — as well as the data and implementation of the *run-time system*.

Malenfant et al., *A Tutorial on Behavioral Reflection and its Implementation*, 1996

Roadmap

> Reification and reflection

> **Introspection**

— **Inspecting objects**

— Querying code

— Accessing run-time contexts

> Intercession

— Overriding doesNotUnderstand:

— Anonymous classes

— Method wrappers

Selected material by Marcus Denker and Stéphane Ducasse

The Essence of a Class

1. A format
   — I.e., a number of instance variables and types
2. A superclass
3. A method dictionary

Behavior class>> new

> In Pharo:

```
Behavior class>>new
    | classInstance |
    classInstance := self basicNew.
    classInstance methodDictionary:
        classInstance emptyMethodDictionary.
    classInstance superclass: Object.
    classInstance setFormat: Object format.
    ^ classInstance
```

NB: not to be confused with `Behavior>>new`!

The Essence of an Object

1. Class pointer
2. Values

> Can be special:
  — `SmallInteger`
  — Indexed rather than pointer values
  — Compact classes (`CompiledMethod`, `Array` ...)

Metaobjects vs metaclasses

- Need distinction between metaclass and metaobject!
  - A metaclass is a class whose instances are classes
  - A metaobject is an object that describes or manipulates other objects
    - *Different metaobjects can control different aspects of objects*

Some MetaObjects

- ***Structure:***
  - Behavior, ClassDescription, Class, Metaclass, ClassBuilder
- ***Semantics:***
  - Compiler, Decompiler, IRBuilder
- ***Behavior:***
  - CompiledMethod, BlockContext, Message, Exception
- ***ControlState:***
  - BlockContext, Process, ProcessorScheduler
- ***Resources:***
  - WeakArray
- ***Naming:***
  - SystemDictionary
- ***Libraries:***
  - MethodDictionary, ClassOrganizer

Meta-Operations

“Meta-operations are operations that provide information about an object as opposed to information directly contained by the object ...They permit things to be done that are not normally possible”

Inside Smalltalk

Accessing state

- *Object*>>instVarAt: aNumber
- *Object*>>instVarNamed: aString
- *Object*>>instVarAt: aNumber put: anObject

```
pt := 10@3.
pt instVarNamed: 'x'.
pt instVarNamed: 'x' put: 33.
pt
```

```
10

33@3
```

Accessing meta-information

- *Object>>*`class`
- *Object>>*`identityHash`

`'hello' class`	`ByteString`
`(10@3) class`	`Point`
`Smalltalk class`	`SystemDictionary`
`Class class`	`Class class`
`Class class class`	`Metaclass`
`Class class class class`	`Metaclass class`
`'hello' identityHash`	`2664`
`Object identityHash`	`2274`
`5 identityHash`	`5`

Changes

- `Object>>primitiveChangeClassTo: anObject`
  - both classes should have the same format, *i.e.*, the same physical structure of their instances
    - *"Not for casual use"*

- `Object>>become: anotherObject`
  - Swap the object pointers of the receiver and the argument.
  - All variables in the entire system that used to point to the receiver now point to the argument, and vice-versa.
  - Fails if either object is a SmallInteger

- `Object>>becomeForward: anotherObject`
  - Like `become:` but only in one direction.

Implementing Instance Specific Methods

```
ReflectionTest>>testPrimitiveChangeClassTo
    | behavior browser |

    behavior := Behavior new.
    behavior superclass: Browser.
    behavior setFormat: Browser format.
    browser := Browser new.

    browser primitiveChangeClassTo: behavior new.
    behavior compile: 'thisIsATest ^ 2'.

    self assert: browser thisIsATest = 2.
    self should: [Browser new thisIsATest]
        raise: MessageNotUnderstood.
```

become:

> Swap all the pointers from one object to the other and back (symmetric)

```
ReflectionTest>>testBecome
    | pt1 pt2 pt3 |

    pt1 := 0@0.
    pt2 := pt1.
    pt3 := 100@100.
    pt1 become: pt3.

    self assert: pt1 = (100@100).
    self assert: pt1 == pt2.
    self assert: pt3 = (0@0).
```

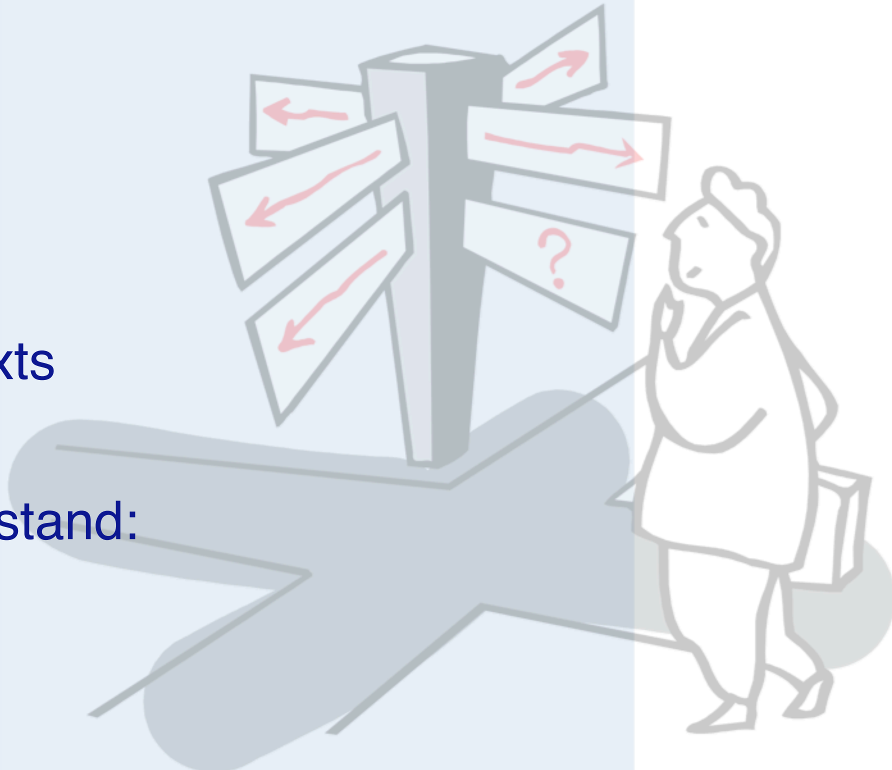
becomeForward:

> Swap all the pointers from one object to the other (asymmetric)

```
ReflectionTest>>testBecomeForward
   | pt1 pt2 pt3 |

   pt1 := 0@0.
   pt2 := pt1.
   pt3 := 100@100.
   pt1 becomeForward: pt3.

   self assert: pt1 = (100@100).
   self assert: pt1 == pt2.
   self assert: pt2 == pt3.
```

Roadmap

> Reification and reflection
> **Introspection**
>   — Inspecting objects
>   — **Querying code**
>   — Accessing run-time contexts
> Intercession
>   — Overriding doesNotUnderstand:
>   — Anonymous classes
>   — Method wrappers



Selected material by Marcus Denker and Stéphane Ducasse

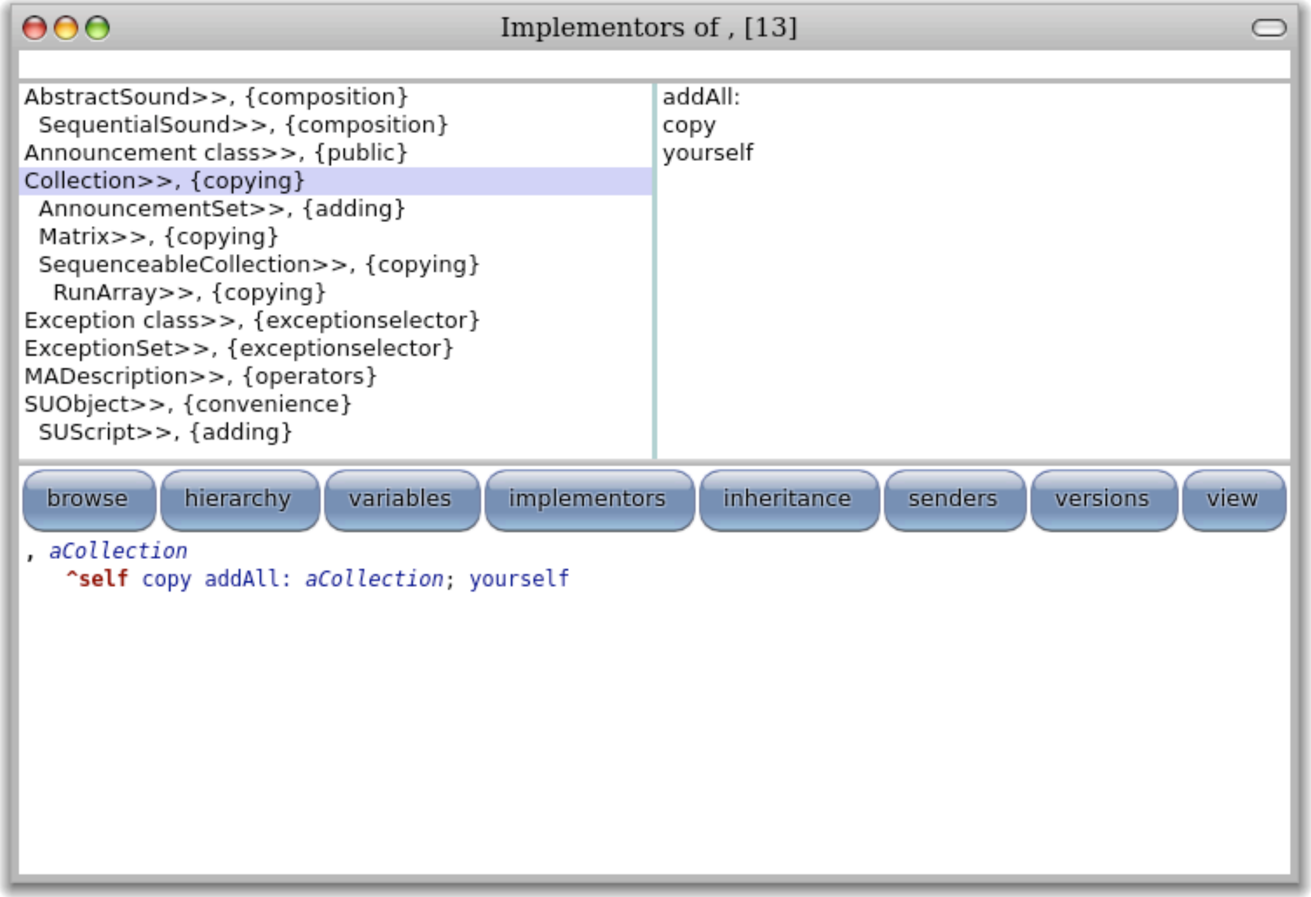
Code metrics

```
Collection allSuperclasses size.
Collection allSelectors size.
Collection allInstVarNames size.
Collection selectors size.
Collection instVarNames size.
Collection subclasses size.
Collection allSubclasses size.
Collection linesOfCode.
```

```
2
610
0
163
0
9
101
864
```

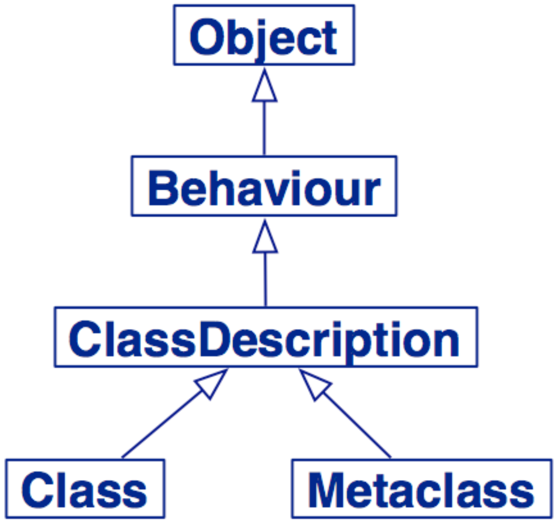
SystemNavigation

```
SystemNavigation default browseAllImplementorsOf: #,
```

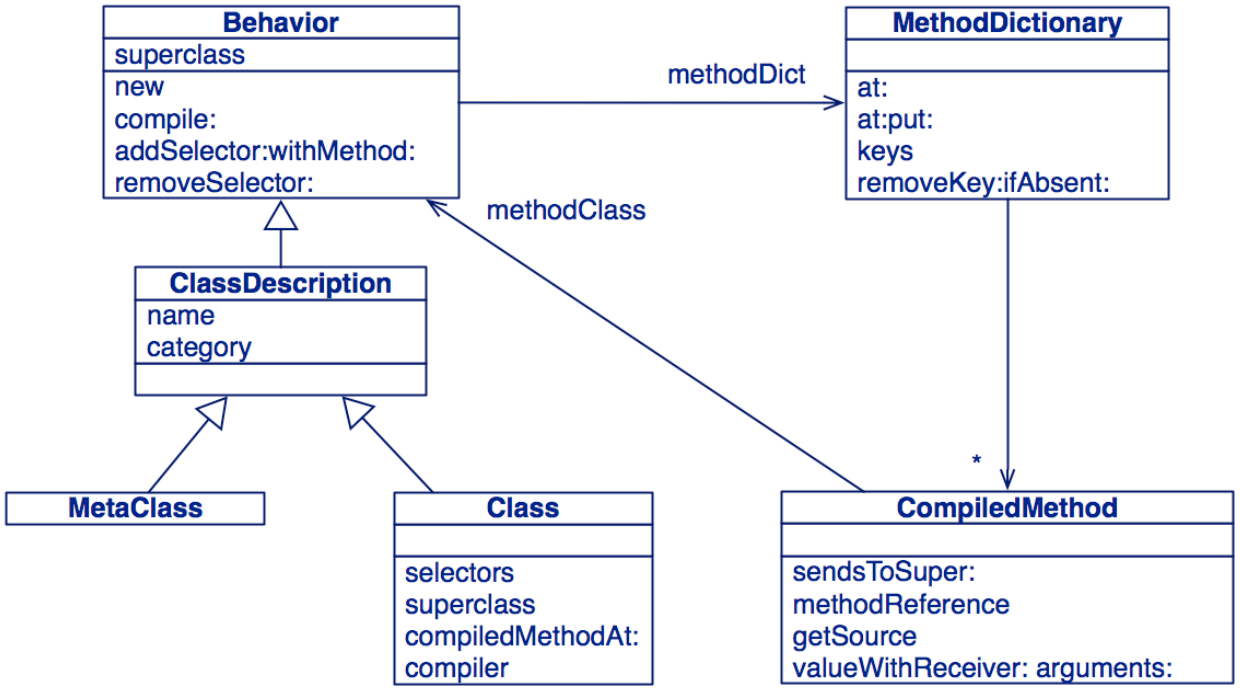


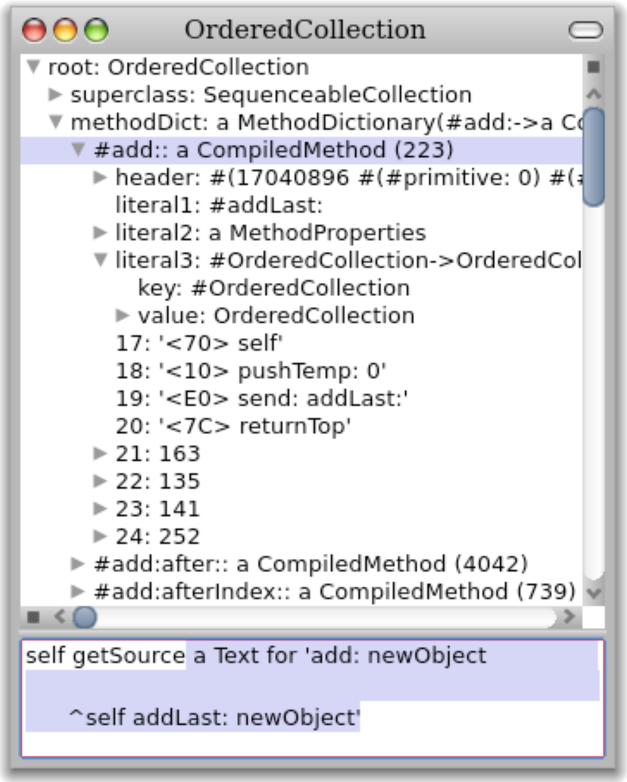
Recap: Classes are objects too

> Object
  - Root of inheritance
  - Default Behavior
  - Minimal Behavior
> Behavior
  - Essence of a class
  - Anonymous class
  - Format, methodDict, superclass
> ClassDescription
  - Human representation and organization
> Metaclass
  - Sole instance



Classes are Holders of CompiledMethods





Invoking a message by its name

```
Object>>perform: aSymbol
Object>>perform: aSymbol with: arg
```

> Asks an object to execute a message
  — Normal method lookup is performed

```
5 factorial
5 perform: #factorial
```

```
120
120
```

Executing a compiled method

```
CompiledMethod>>valueWithReceiver:arguments:
```

- No lookup is performed!

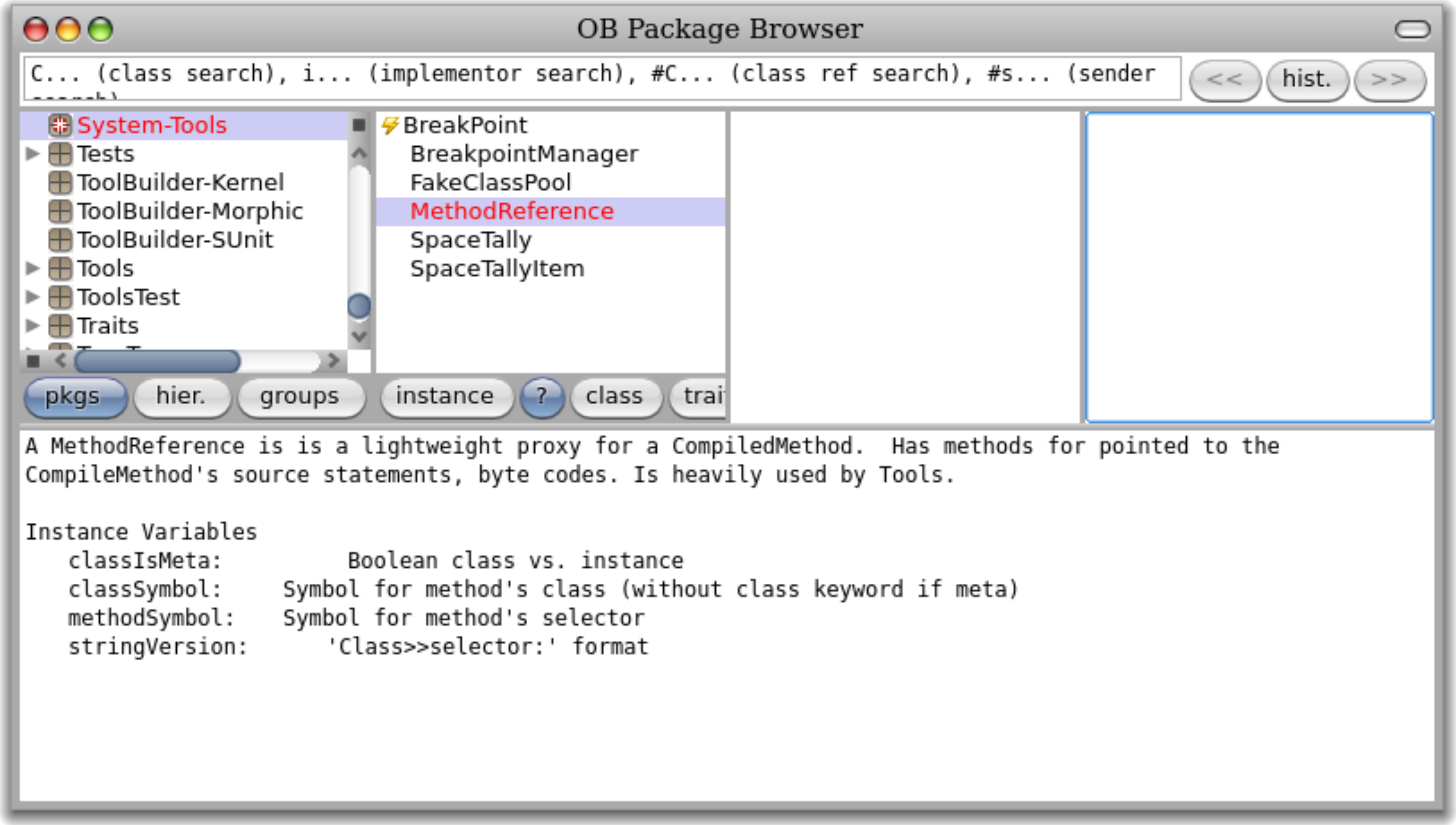
```
(SmallInteger>>#factorial)
   valueWithReceiver: 5
   arguments: #()
```

```
Error: key not found
```

```
(Integer>>#factorial)
   valueWithReceiver: 5
   arguments: #()
```

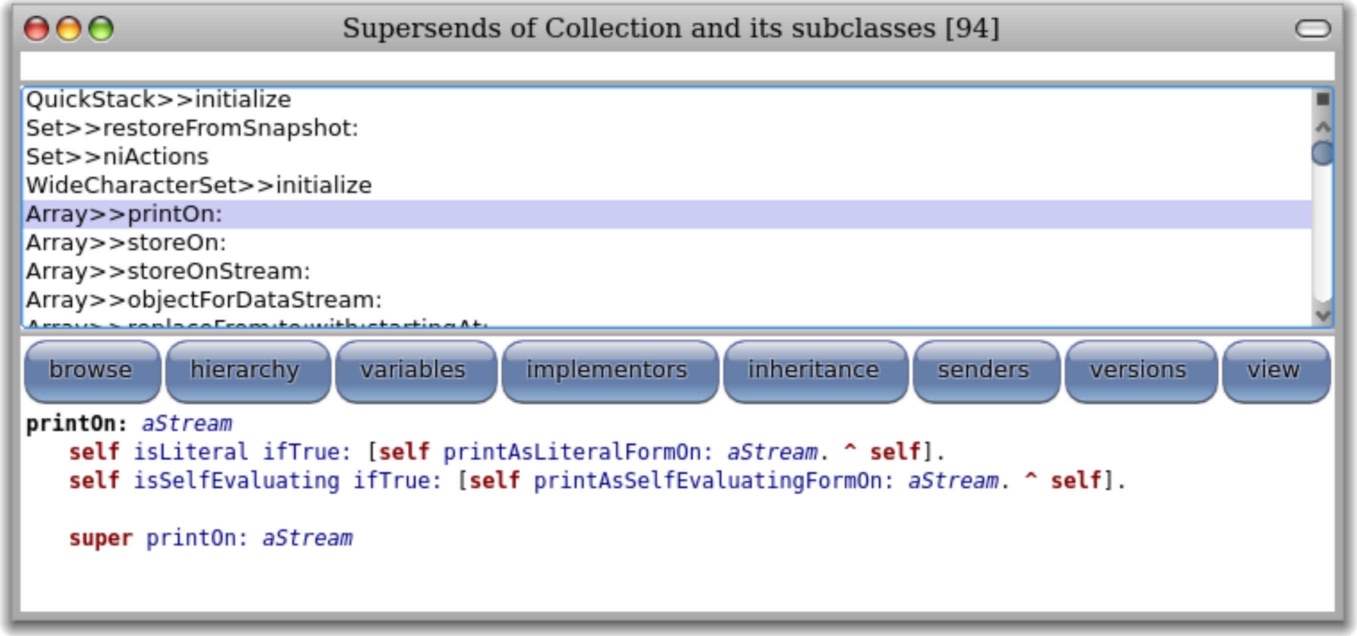
```
120
```

MethodReference



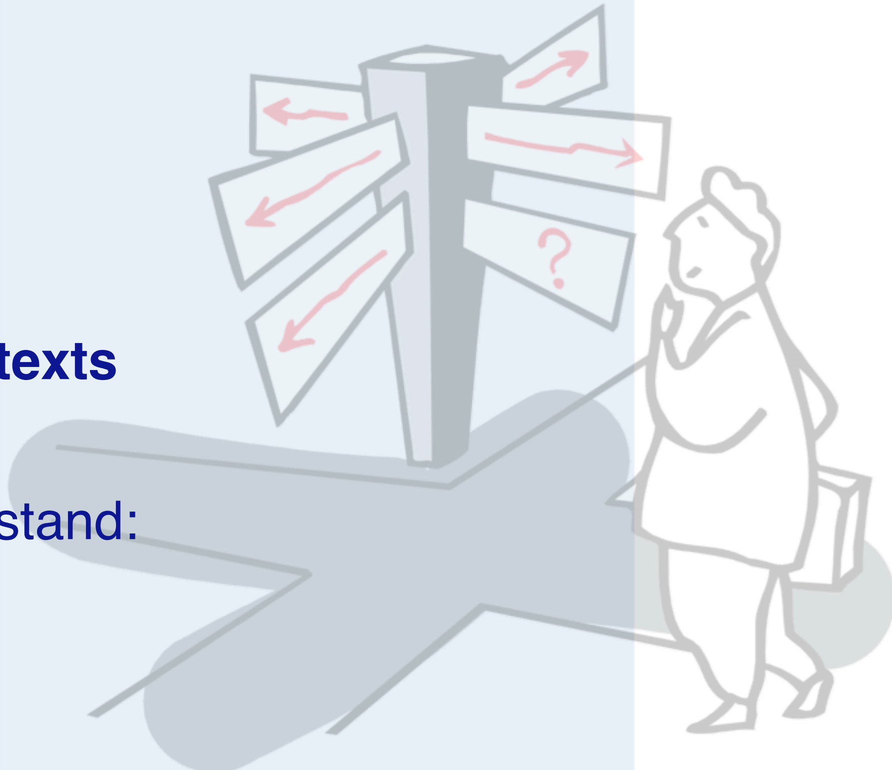
Finding super-sends within a hierarchy

```
class := Collection.
SystemNavigation default
  browseMessageList: (class withAllSubclasses gather: [:each |
    each methodDict associations
      select: [:assoc | assoc value sendsToSuper]
      thenCollect: [:assoc | MethodReference class: each selector: assoc key]])
  name: 'Supersends of ' , class name , ' and its subclasses'
```



Roadmap

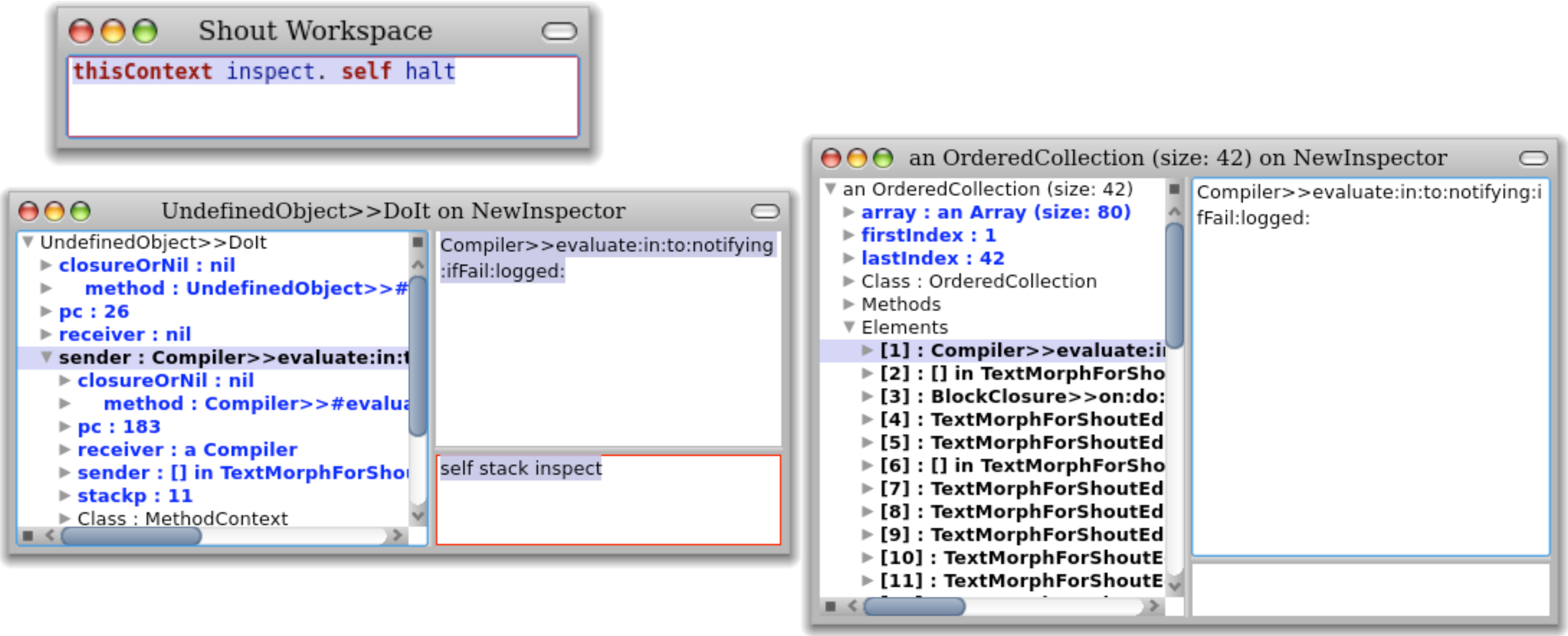
> Reification and reflection
> **Introspection**
  — Inspecting objects
  — Querying code
  — **Accessing run-time contexts**
> Intercession
  — Overriding doesNotUnderstand:
  — Anonymous classes
  — Method wrappers



Selected material by Marcus Denker and Stéphane Ducasse

Accessing the run-time stack

> The execution stack can be *reified* and *manipulated* on demand
  — `thisContext` is a pseudo-variable which gives access to the stack



What happens when a method is executed?

> We need space for:
  - The temporary variables
  - Remembering where to return to

> Everything is an Object!
  - So: we model this space with objects
  - Class `MethodContext`

```
ContextPart variableSubclass: #MethodContext
   instanceVariableNames: 'method closureOrNil receiver'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Methods'
```

MethodContext

- MethodContext holds all state associated with the execution of a CompiledMethod
  - Program Counter (pc, from ContextPart)
  - the Method itself (method)
  - Receiver (receiver) and the Sender (sender)

- The sender is the previous MethodContext
  - (or BlockContext)
  - The *chain of senders* is a stack
  - It grows and shrinks on activation and return

Contextual halting

> You can't put a halt in methods that are called often
  — e.g., `OrderedCollection>>add:`
  — *Idea:* only halt if called from a method with a certain name

```
HaltDemo>>haltIf: aSelector
    | context |
    context := thisContext.
    [context sender isNil]
        whileFalse:
            [context := context sender.
            (context selector = aSelector)
            ifTrue: [ Halt signal ] ].
```

NB: Object>>haltIf: in Pharo is similar

HaltDemo

```
HaltDemo>>foo
    self haltIf: #bar.
    ^ 'foo'

HaltDemo>>bar
    ^ (self foo), 'bar'
```

```
HaltDemo new foo
```

```
'foo'
```

```
HaltDemo new bar
```

Roadmap

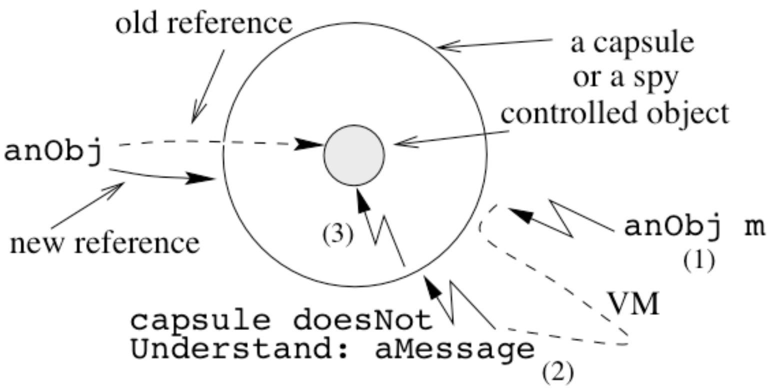
- Reification and reflection
- Introspection
  - Inspecting objects
  - Querying code
  - Accessing run-time contexts
- **Intercession**
  - **Overriding doesNotUnderstand:**
  - Anonymous classes
  - Method wrappers

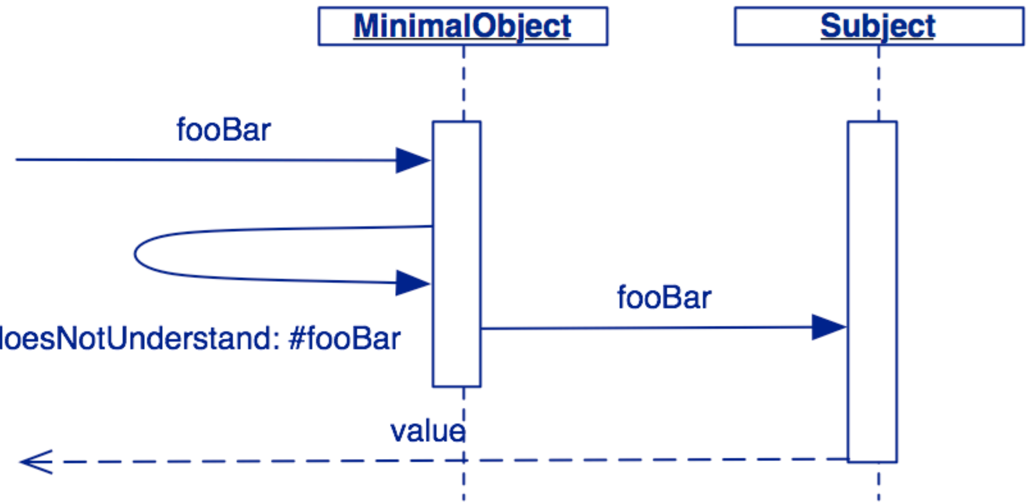
Selected material by Marcus Denker and Stéphane Ducasse

Overriding doesNotUnderstand:

> Introduce a Minimal Object
  - Wraps a normal object
  - Does not understand very much
  - Redefines `doesNotUnderstand:`
  - Superclass is `nil` or `ProtoObject`
  - Uses `becomeForward:` to substitute the object to control

Minimal Object at Work





Logging message sends with a minimal object

```
ProtoObject subclass: #LoggingProxy
   instanceVariableNames: 'subject invocationCount'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'PBE-Reflection'
```

```
LoggingProxy>>initialize
   invocationCount := 0.
   subject := self.
```

```
LoggingProxy>>doesNotUnderstand: aMessage
   Transcript show: 'performing ', aMessage printString; cr.
   invocationCount := invocationCount + 1.
   ^ aMessage sendTo: subject
```

```
Message>>sendTo: receiver
   ^ receiver perform: selector withArguments: args
```

Using become: to install a proxy

```
testDelegation
   | point |
   point := 1@2.
   LoggingProxy new become: point.
   self assert: point invocationCount = 0.
   self assert: point + (3@4) = (4@6).
   self assert: point invocationCount = 1.
```

NB: `become:` will swap the `subject` variable of the proxy

Limitations

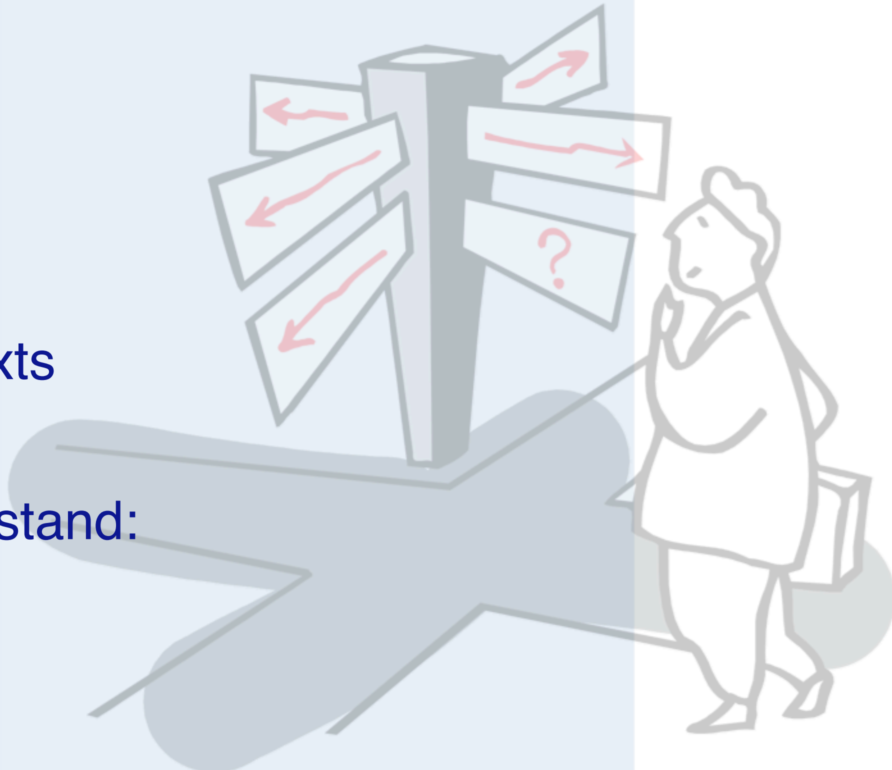
- self problem
  - Messages sent by the object to itself are not trapped!
- Class control is impossible
  - Can’t swap classes
- Interpretation of minimal protocol
  - What to do with messages that are understood by both the MinimalObject and its subject?

Using minimal objects to dynamically generate code

```
DynamicAccessors>>doesNotUnderstand: aMessage
   | messageName |
   messageName := aMessage selector asString.
   (self class instVarNames includes: messageName)
      ifTrue: [self class compile:
                  messageName , String cr , ' ^ ' , messageName.
         ^ aMessage sendTo: self].
   super doesNotUnderstand: aMessage
```

A minimal object can be used to dynamically generate or lazily load code that does not yet exist.

Roadmap



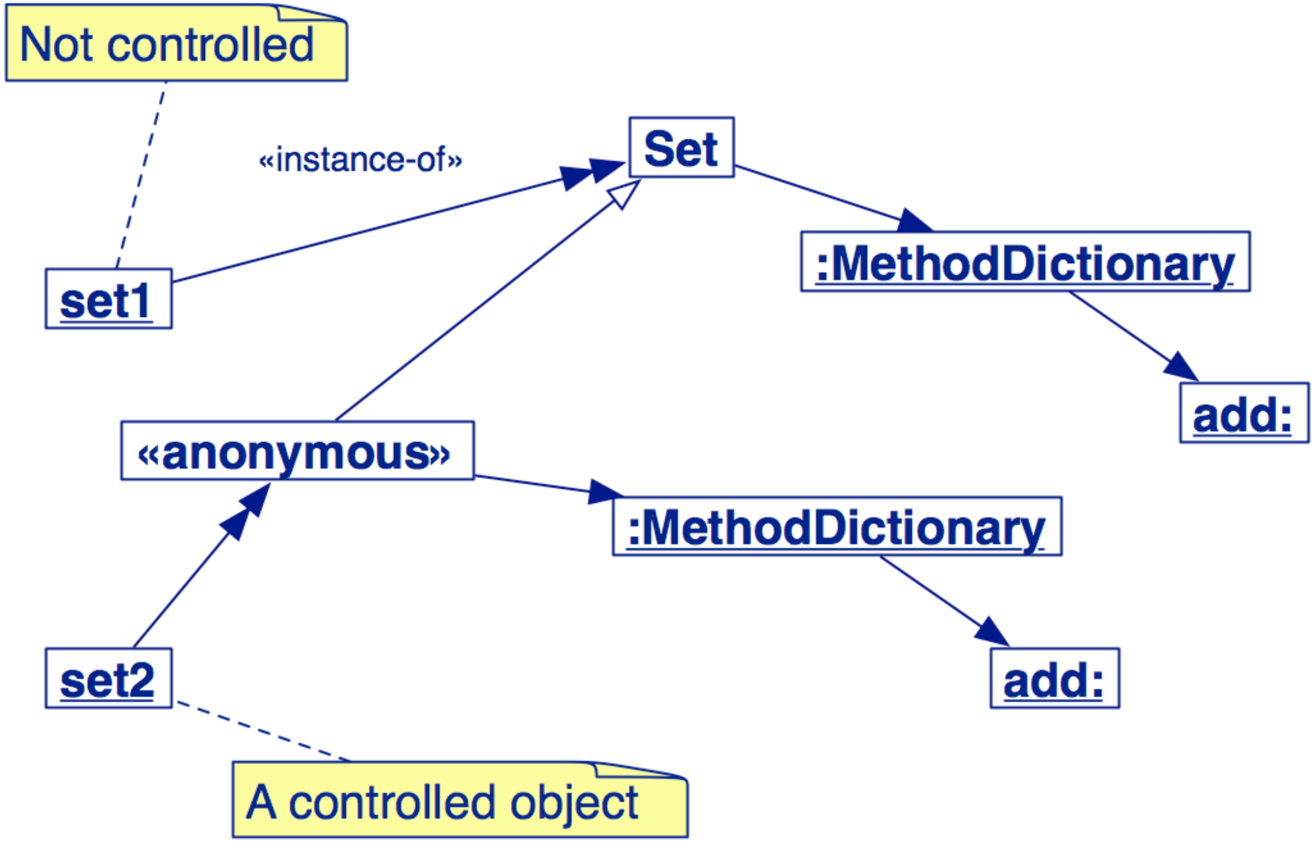
- Reification and reflection
- Introspection
  - Inspecting objects
  - Querying code
  - Accessing run-time contexts
- **Intercession**
  - Overriding doesNotUnderstand:
  - **Anonymous classes**
  - Method wrappers

Selected material by Marcus Denker and Stéphane Ducasse

Message control with anonymous classes

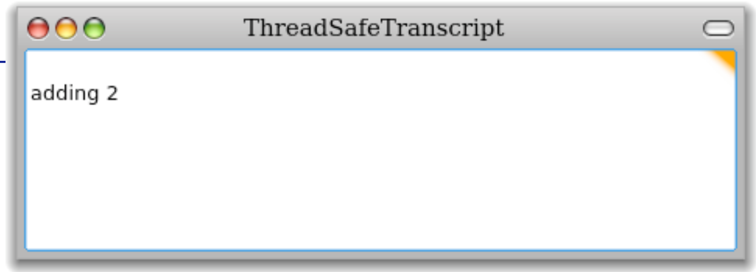
- Create an *anonymous class*
  - Instance of Behavior
  - Define controlling methods
  - Interpose it between the instance and its class

Selective control



Anonymous class in Pharo

```
| anonClass set |
anonClass := Behavior new.
anonClass superclass: Set;
   setFormat: Set format.

anonClass compile:
   'add: anObject
      Transcript show: ''adding '', anObject printString; cr.
      ^ super add: anObject'.

set := Set new.
set add: 1.

set primitiveChangeClassTo: anonClass basicNew.
set add: 2.
```

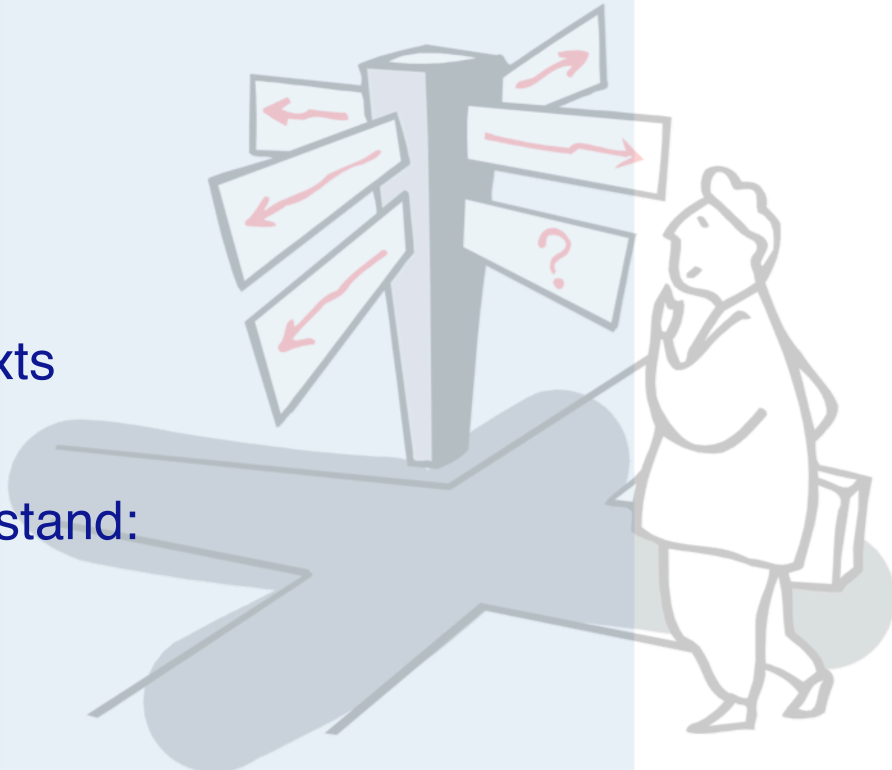


Evaluation

- Either instance-based or group-based
- Selective control
- No self-send problem
- Good performance
- Transparent to the user
- Requires a bit of compilation
  - (could be avoided using clone as in Method Wrapper)

Roadmap

> Reification and reflection
> Introspection
  — Inspecting objects
  — Querying code
  — Accessing run-time contexts
> **Intercession**
  — Overriding doesNotUnderstand:
  — Anonymous classes
  — **Method wrappers**



Selected material by Marcus Denker and Stéphane Ducasse

Method Substitution

First approach:

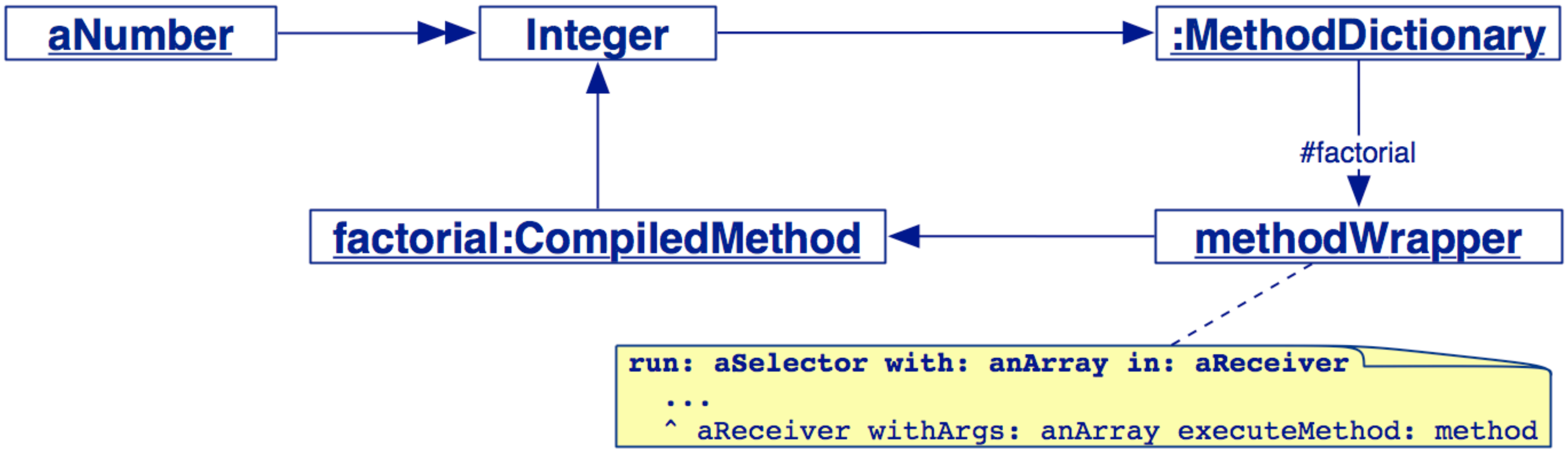
> Add methods with *mangled names*
> — but the user can see them

Second approach:

> Wrap the methods without polluting the interface
> — replace the method by an object that implements `run:with:in:`

MethodWrapper before and after methods

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



A LoggingMethodWrapper

```
LoggingMethodWrapper>>initializeOn: aCompiledMethod
   method := aCompiledMethod.
   reference := aCompiledMethod methodReference.
   invocationCount := 0
```

```
LoggingMethodWrapper>>install
   reference actualClass methodDictionary
      at: reference methodSymbol
      put: self
```

`uninstall` is similar …

```
LoggingMethodWrapper>>run: aSelector with: anArray in: aReceiver
   invocationCount := invocationCount + 1.
   ^ aReceiver withArgs: anArray executeMethod: method
```

NB: Duck-typing also requires (empty) `flushCache`, `methodClass:`, and `selector:` methods

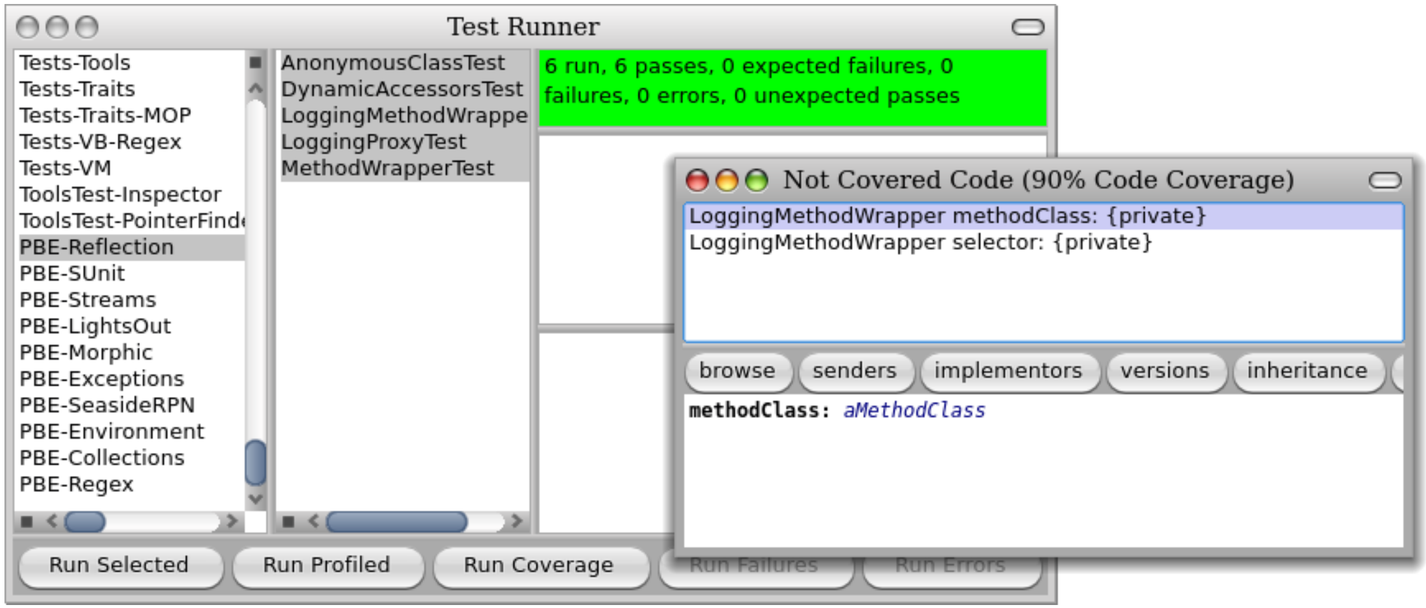
Installing a LoggingMethodWrapper

```
logger := LoggingMethodWrapper on:
Integer>>#factorial.

logger invocationCount.    0
5 factorial.
logger invocationCount.    0

logger install.
[ 5 factorial ] ensure: [logger uninstall].
logger invocationCount.    6

10 factorial.
logger invocationCount.    6
```

Checking Test Coverage

```
TestCoverage>>run: aSelector with: anArray in: aReceiver
   self mark; uninstall.
   ^ aReceiver withArgs: anArray executeMethod: method
```

```
TestCoverage>>mark
   hasRun := true
```



Evaluation

- Class based:
  - all instances are controlled
- Only known messages intercepted
- A single method can be controlled
- Does not require compilation for installation/removal

What you should know!

- ✎ *What is the difference between introspection and intercession?*
- ✎ *What is the difference between structural and Behavioral reflection?*
- ✎ *What is an object? What is a class?*
- ✎ *What is the difference between performing a message send and simply evaluating a method looked up in a MethodDictionary?*
- ✎ *In what way does thisContext represent the run-time stack?*
- ✎ *What different techniques can you use to intercept and control message sends?*

Can you answer these questions?

- ✎ *What form of "reflection" is supported by Java?*
- ✎ *What can you do with a metacircular architecture?*
- ✎ *Why are Behavior and Class different classes?*
- ✎ *What is the class ProtoObject good for?*
- ✎ *Why is it not possible to become: a SmallInteger?*
- ✎ *What happens to the stack returned by thisContext if you proceed from the* `self halt`*?*
- ✎ *What is the metaclass of an anonymous class?*

License