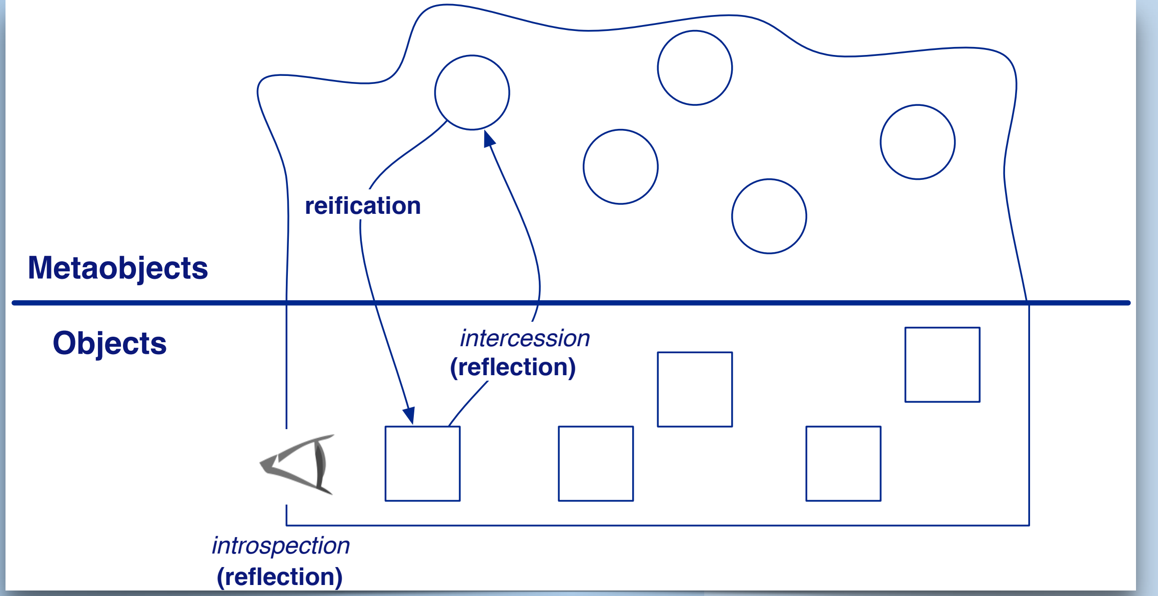
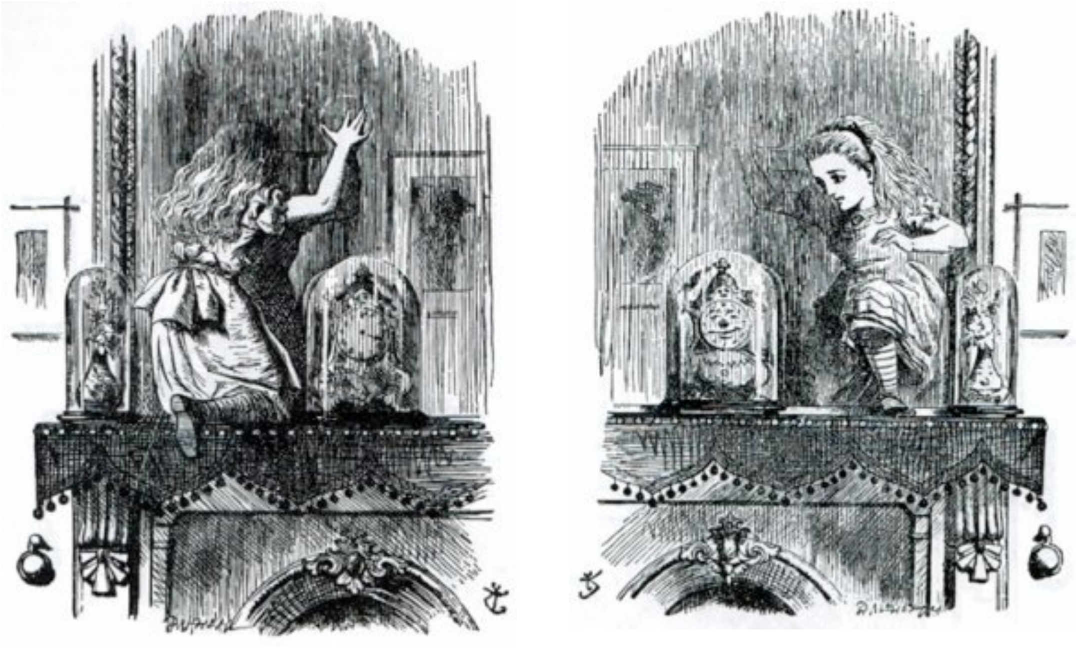
Reflection

Oscar Nierstrasz



Selected material by Marcus Denker and Stéphane Ducasse

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
- Reflection in Programming Languages
- Introspection
  - Inspecting objects
  - Querying code
  - Accessing run-time contexts
- Intercession
  - Overriding doesNotUnderstand:
  - Anonymous classes
  - Method wrappers

Roadmap

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

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, 1993

Adding new features to a high-level language is only feasible if the language is “opened up” with the help of reflection. A “meta-object protocol” is a kind of reflection API.

Gregor Kiczales, et al., “Metaobject protocols: Why we want them and what else they can do,” in Object-Oriented Programming: the CLOS Perspective, pp. 101-118, MIT Press, 1993.

http://scgresources.unibe.ch/Literature/SMA/Kicz93b-MOPs.pdf

What are Reflection and Reification? (review)

- *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

In order to “reflect” on one’s own behaviour, one must have a model of it, that is, one must *make it concrete*, or “reify” it.

Most programming languages provide some reflective features to allow you to query a running system. This is known as “introspection”. Few languages allow you to *change the running system* through reflection; this is *intercession*.

Daniel G. Bobrow, Richard P. Gabriel, and J.L. White. *CLOS in Context — The Shape of the Design*. In A. Paepcke (Ed.), Object-Oriented Programming: the CLOS perspective, p. 29—61, MIT Press,

http://scgresources.unibe.ch/Literature/SMA/Bobr93a-CLOS.pdf

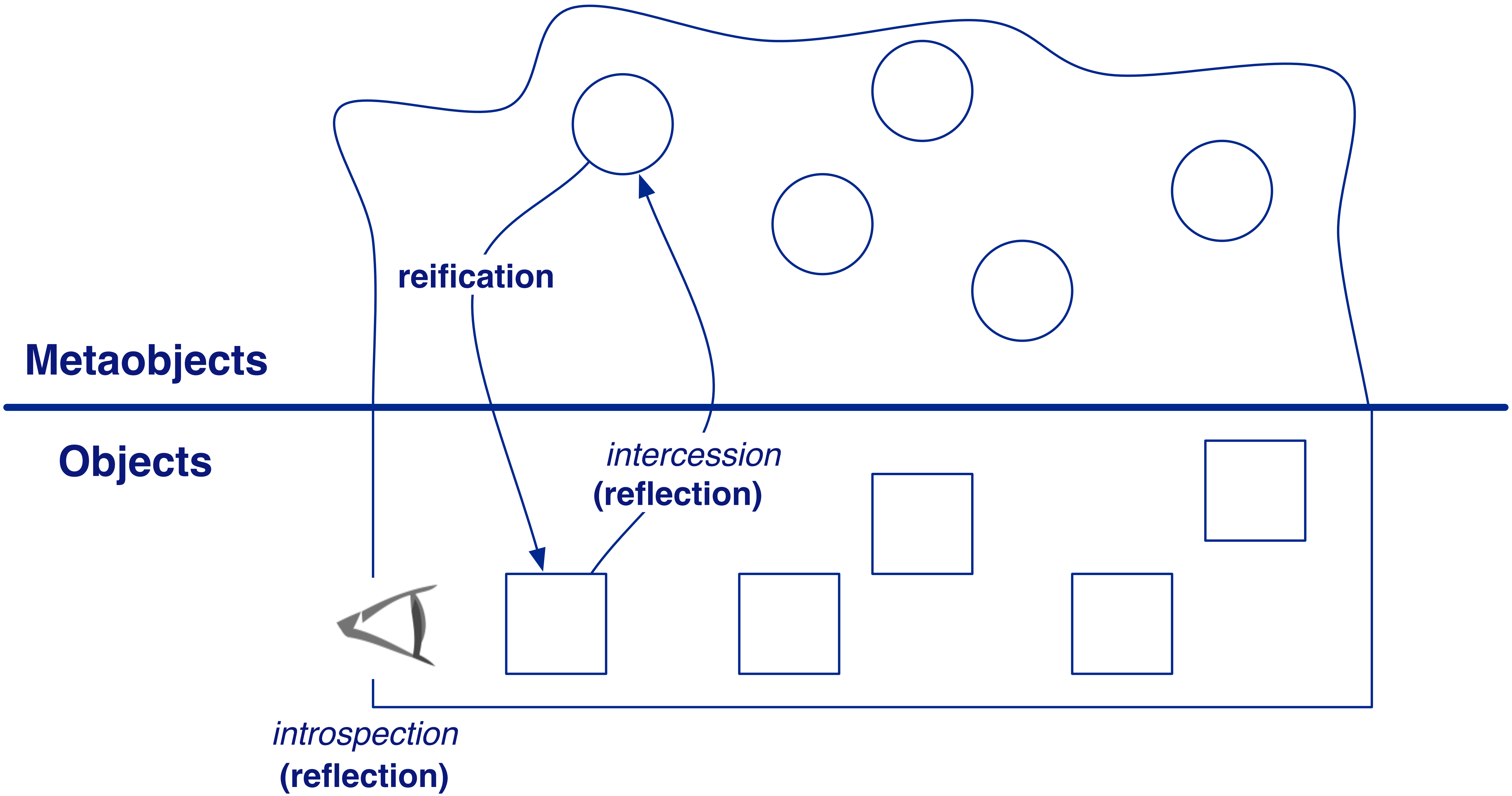
Structural and behavioral reflection

> Structural reflection lets you reify and reflect on
  - the *program* currently executed
  - its *abstract data types*.

> Behavioral reflection lets you reify and reflect on
  - the language *semantics* and *implementation* (processor)
  - the data and implementation of the *run-time system*.

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

Reflection and Reification (review)



reification

Metaobjects

Objects

intercession
(reflection)

introspection
(reflection)

To reflect on the structure or behaviour of a system we must *reify* concepts from the metamodel (i.e., from the implementation) to make them available to the run time system as ordinary “objects”. We can then examine or “introspect” these objects.

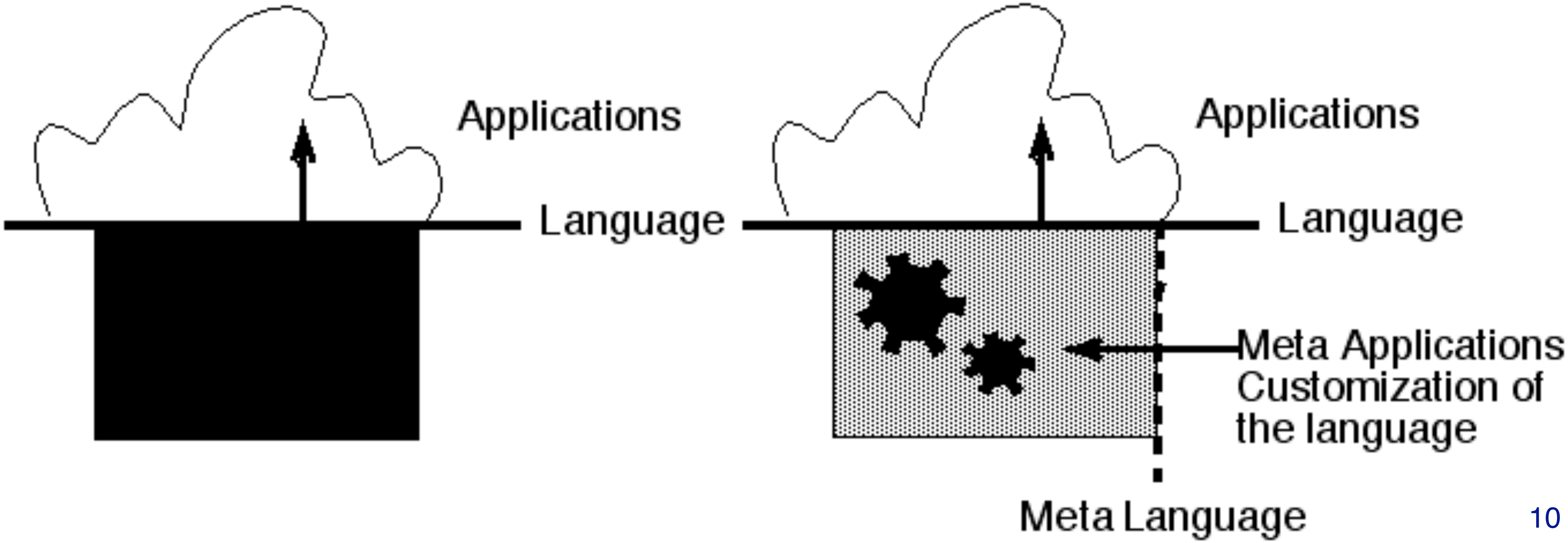
If we can *change* these objects and *reflect these changes* back to the meta level, then we are performing *intercession*.

Roadmap

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

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*



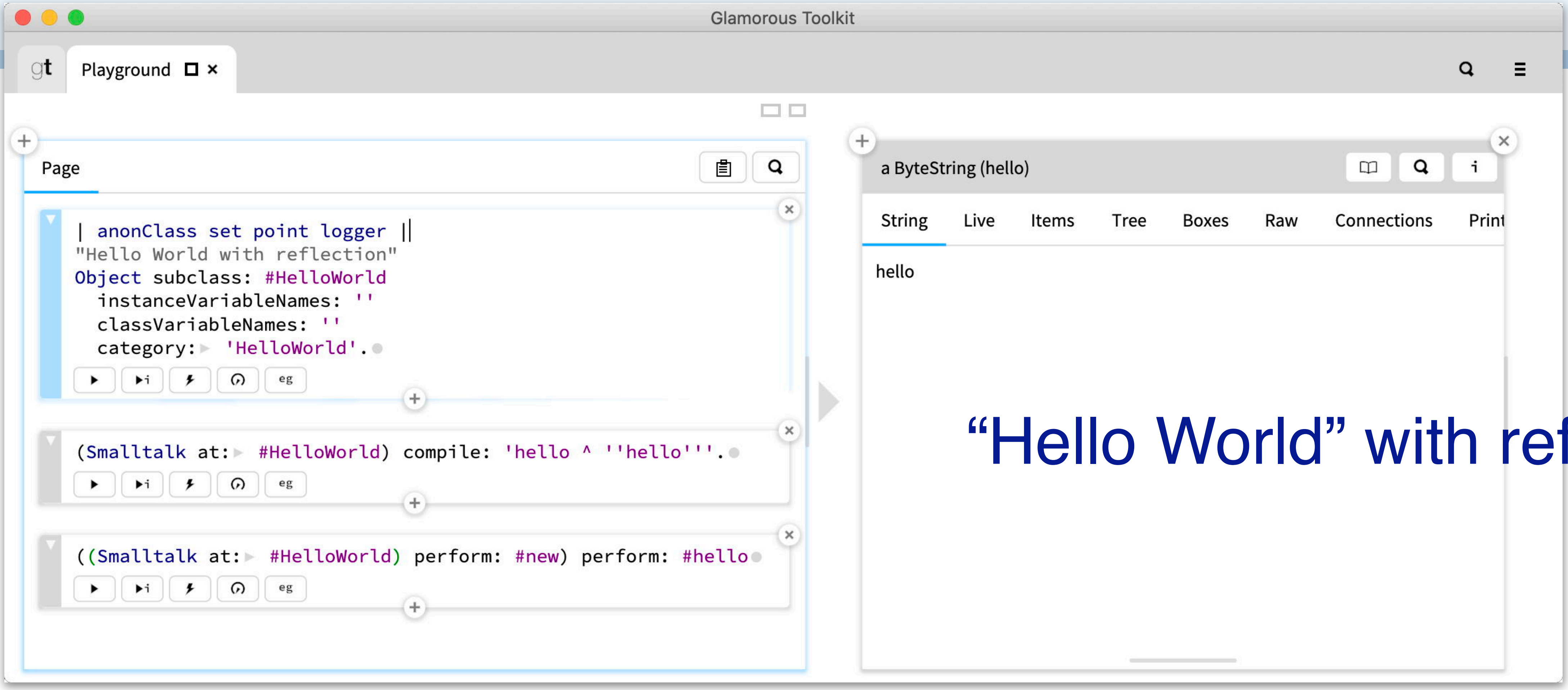
Introspection in Java

```java
// Without introspection
World world = new World();
world.hello();
```

```java
// With introspection
Class cls = Class.forName("World");
Method method = cls.getMethod("hello", null);
method.invoke(cls.newInstance(), null);
```

In Java we can reify classes, inspect them, and invoke certain services to create instances or call methods, but we cannot compile new classes or methods. (To do so requires class loader magic.)

Reflection in Smalltalk



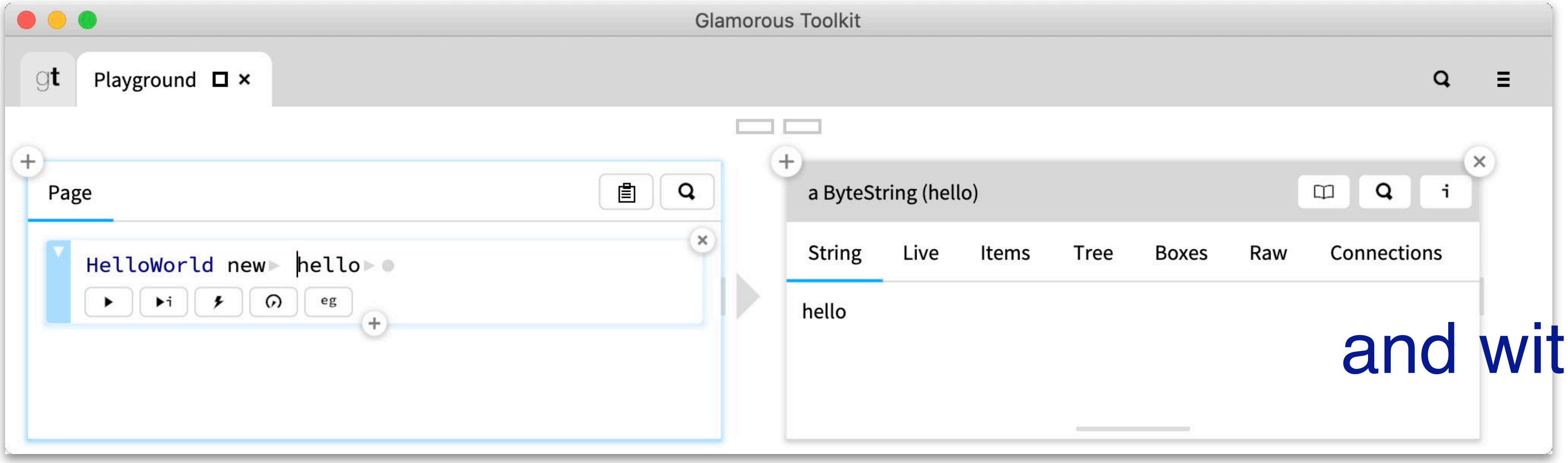
```
| anonClass set point logger |
"Hello World with reflection"
Object subclass: #HelloWorld
  instanceVariableNames: ''
  classVariableNames: ''
  category: 'HelloWorld'.
```

```
(Smalltalk at: #HelloWorld) compile: 'hello ^ ''hello'''.
```

```
((Smalltalk at: #HelloWorld) perform: #new) perform: #hello
```

```
hello
```

“Hello World” with reflection



```
HelloWorld new hello
```

```
hello
```

and without

In Smalltalk we can create classes and compile methods at run time simply by interacting with reified classes. (In fact, we *must*, since there is no other way to compile new code.)

Three approaches

1. Tower of meta-circular interpreters
2. Reflective languages
3. Open implementation

1. Tower of meta-circular interpreters

- Each level interprets and controls the next
  - 3-Lisp, Scheme
- “Turtles all the way down” [up]
  - In practice, levels are reified on-demand

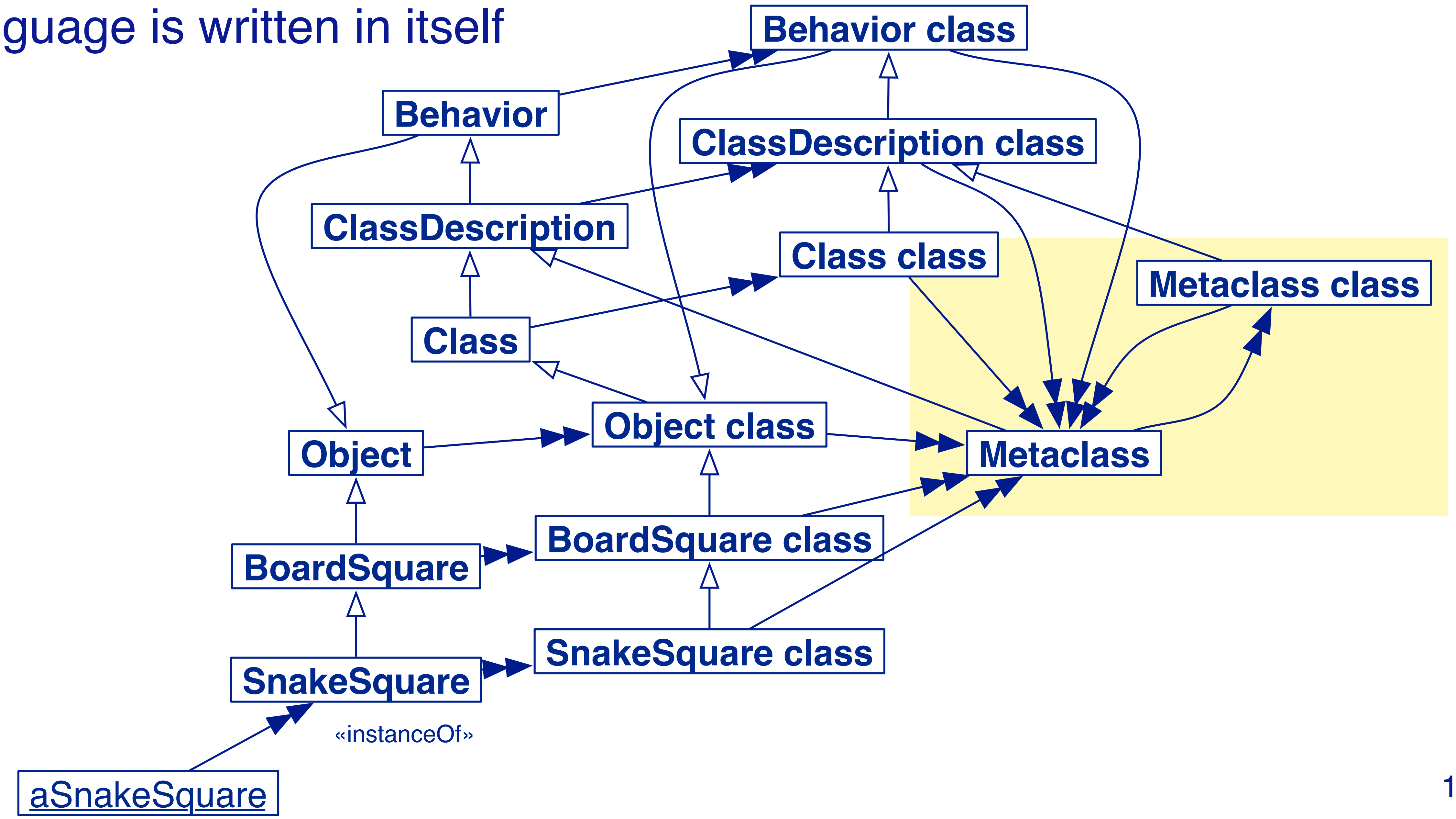
In this approach there is an infinite tower of interpreters, each interpreting the next layer below. In practice, of course, this tower does not really exist, but only springs into existence on request — if you need to do something at a given level, then that level will be reified on demand.

2. Reflective languages

> Meta-entities control base entities

— Smalltalk, Self

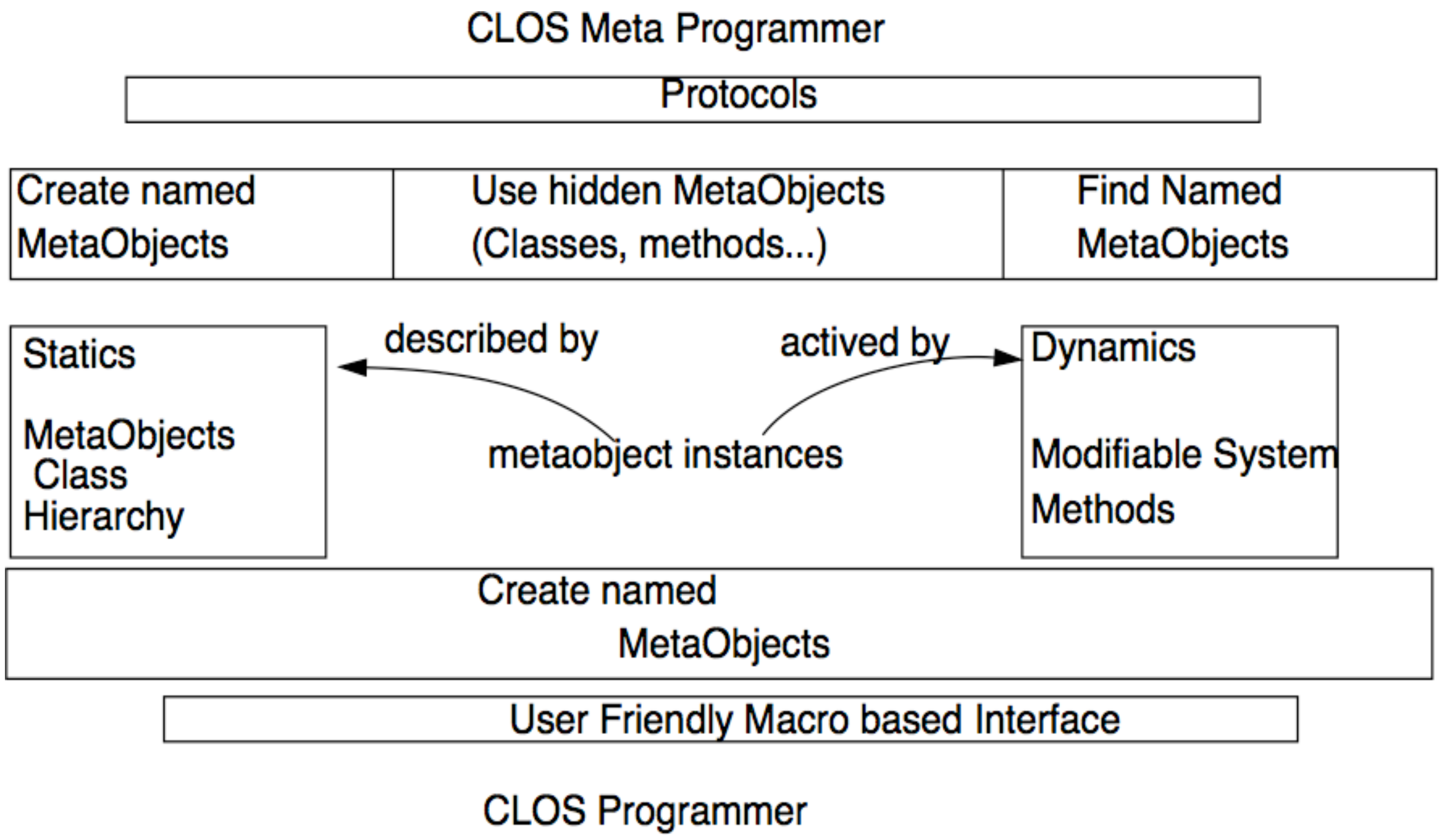
— Language is written in itself



Smalltalk adopts the second approach: the language is reflective, and all meta-entities are reified and can be accessed at the base level. In contrast to the previous approach there is only one level of interpretation.

3. Open implementation

- *Meta-object protocols* provide an interface to access and modify the implementation and semantics of a language
  - CLOS
- *More efficient, less expressive than infinite towers*



CLOS Meta Programmer

Protocols

Create named MetaObjects	Use hidden MetaObjects (Classes, methods...)	Find Named MetaObjects

Statics

MetaObjects
Class
Hierarchy

described by

actived by

metaobject instances

Dynamics

Modifiable System
Methods

Create named
MetaObjects

User Friendly Macro based Interface

CLOS Programmer

The Common Lisp Object System (CLOS) instead offers a dedicated API, known as a Meta-Object Protocol (MOP). Meta-objects are responsible for for controlling base entities.

Note that while the metaclass hierarchy of Smalltalk essentially serves as a MOP, in general a MOP does not need to reify metamodel entities.

https://en.wikipedia.org/wiki/Metaobject

https://en.wikipedia.org/wiki/The_Art_of_the_Metaobject_Protocol

Roadmap

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

The Essence of a Class

1. A format (e.g. a set of instance variables)
2. A method dictionary
3. A superclass

Classes serve three purposes:

1. to define the structure of instances (format)
2. to serve as a repository of behavior (method dictionary)
3. to support a class hierarchy (superclass)

Behavior >> initialize

In Pharo:

```
initialize
   "moved here from the class side's #new"
   super initialize.
   self superclass: Object.
   "no longer sending any messages, some of them crash the VM"
   self methodDict: self emptyMethodDictionary.
   self setFormat: Object format.
   self traitComposition: nil.
   self users: IdentitySet new
```

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

Note that this is the default initialization method for all entities with behaviour, in particular classes and metaclasses.

The superclass of a new class is initially set to be `Object`, and then later redefined to its actual superclass.

The initial method dictionary is empty. The “format” is an integer that encodes the object layout.

Aside: This method actually comes from the trait `TBehavior`, but we will not discuss traits for now. (*Traits* are reusable sets of methods that can be shared across classes independently of the inheritance hierarchy.)

The Essence of an Object

1. Objects are *references* (“pointers”)
2. Objects *contain values* (references to other objects)
3. Objects have a *class* (reference to a class)

- Can be special:
  - `SmallInteger`
  - Indexed rather than referenced values
  - Compact classes (`CompiledMethod`, `Array` ...)

Most objects in Smalltalk consist of a set of named instance variables, which are references to other objects. Special cases are `SmallInteger`s, which occupy 31 bits (the last bit is used to distinguish `SmallInteger`s from object references), and indexed objects, which contain indexed rather than named properties.

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*

Wilf LaLonde and John Pugh. Inside Smalltalk: Volume 1, Prentice Hall, 1990. p. 195

http://sdmeta.gforge.inria.fr/FreeBooks/InsideST/InsideSmalltalk.pdf

Accessing state

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

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

```
10

33@3
```

Note how reflective operations violate encapsulation. Even though instance variables are “private” in Smalltalk, we can violate this privacy by explicitly reading and writing named instance variables of arbitrary objects.

Accessing meta-information

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

```
'hello' class
(10@3) class
Smalltalk class
Class class
Class class class
Class class class class

'hello' identityHash
Object identityHash
5 identityHash
```

```
ByteString
Point
SmalltalkImage
Class class
Metaclass
Metaclass class

2664
2274
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 references 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
  | anon anObject |
  anon := Class new.   "NB: an anonymous class"
  anon superclass: Object.
  anon setFormat: Object format.

  anObject := Object new.
  anObject primitiveChangeClassTo: anon new.
  anon compile: 'thisIsATest ^ 2'.

  self assert: anObject thisIsATest equals: 2.
  self should: [ Object new thisIsATest ]
    raise: MessageNotUnderstood
```

Here we create an anonymous class `anon` as an instance of `Class`, and we explicitly set its superclass and format.

We manually set the class of `anObject` to be `anon` (note that `Object>>primitiveChangeClassTo:` takes an *object*, not a class as its argument), and we dynamically compile the method `thisIsATest`.

become:

- Swap all the references 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 equals: (100@100).
  self assert: pt1 == pt2.
  self assert: pt3 equals: (0@0).
```

becomeForward:

- Swap all the references 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 equals: (100@100).
  self assert: pt1 == pt2.
  self assert: pt2 == pt3.
```

Roadmap

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

Basic code metrics

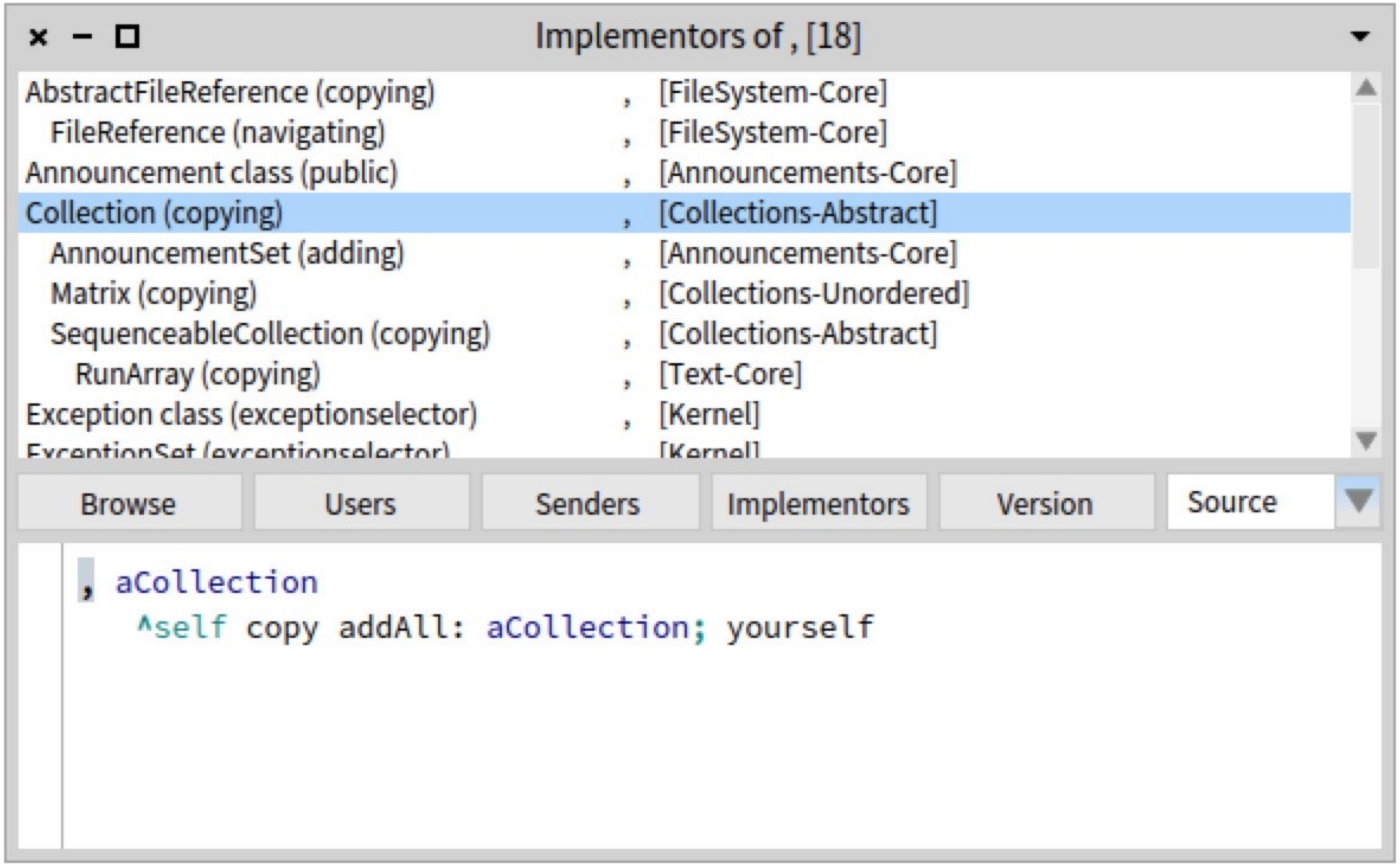
```
Collection allSuperclasses size.      2
Collection allSelectors size.         622
Collection allInstVarNames size.      0
Collection selectors size.            199
Collection instVarNames size.         0
Collection subclasses size.           14
Collection allSubclasses size.        96
Collection linesOfCode.               1077
```

Many code metrics are directly computed by methods of classes.

Most of these methods are defined in `Behavior`.

SystemNavigation (Pharo)

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



Implementors of , [18]

AbstractFileReference (copying) ,	[FileSystem-Core]
FileReference (navigating) ,	[FileSystem-Core]
Announcement class (public) ,	[Announcements-Core]
Collection (copying) ,	[Collections-Abstract]
AnnouncementSet (adding) ,	[Announcements-Core]
Matrix (copying) ,	[Collections-Unordered]
SequenceableCollection (copying) ,	[Collections-Abstract]
RunArray (copying) ,	[Text-Core]
Exception class (exceptionselector) ,	[Kernel]
ExceptionSet (exceptionselector) ,	[Kernel]

```
, aCollection
  ^self copy addAll: aCollection; yourself
```

The class `SystemNavigation` supports a gamut of standard useful queries. Evaluate `SystemNavigation default` to get an instance.

A useful method to search for methods containing a particular source code snippet is:

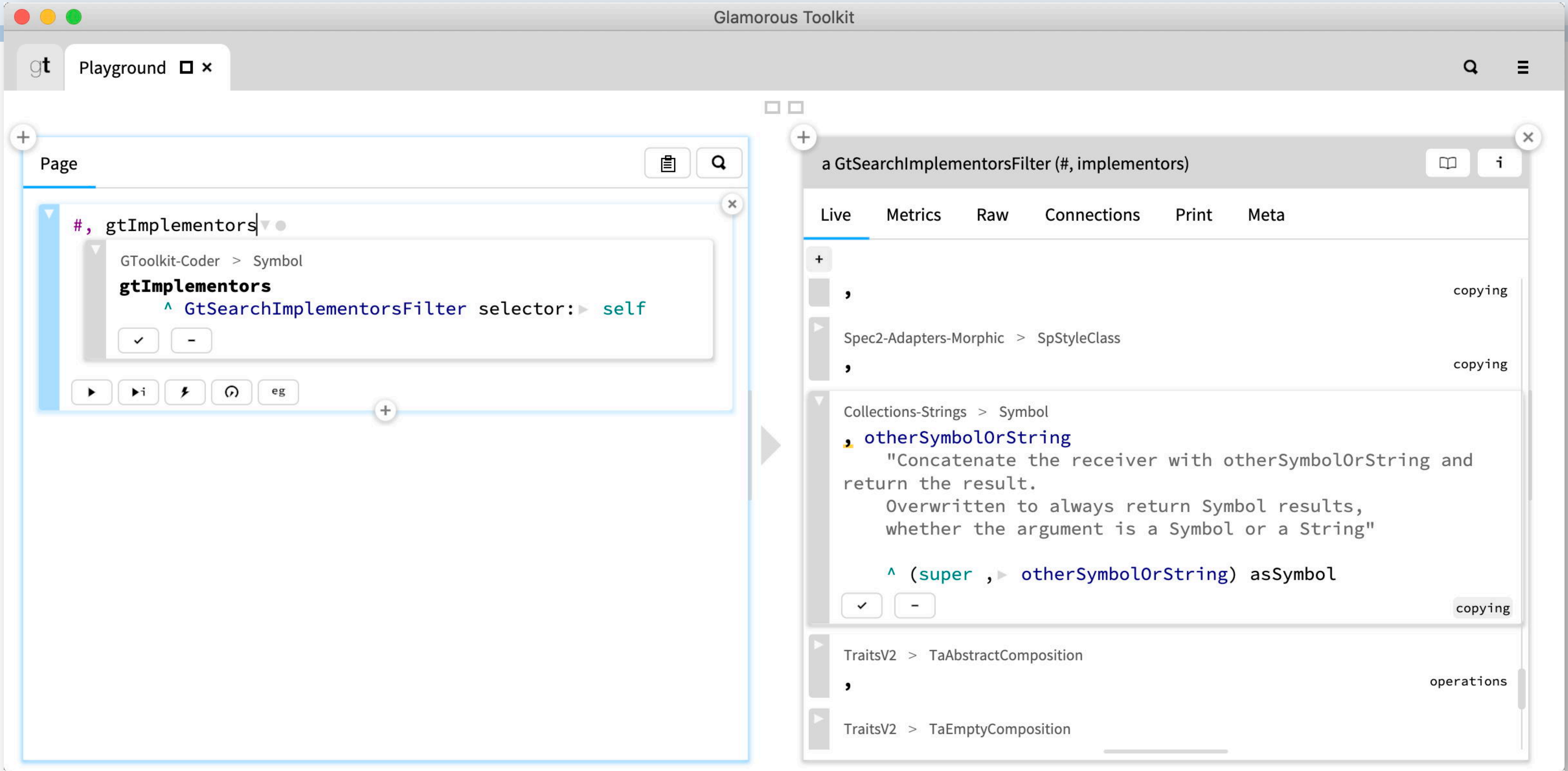
```
SystemNavigation>>allMethodsWithSourceString:matchCase:
```

Browse `SystemNavigation` to find other useful queries.

For this example, there is a convenience method of `CompiledMethod` to do the same thing:

```
#, implementors
```

Gt search filters



```
#, gtImplementors
```

GToolkit-Coder > Symbol

```
gtImplementors
	^ GtSearchImplementorsFilter selector: self
```

```
,                                                   copying
```

Spec2-Adapters-Morphic > SpStyleClass

```
,                                                   copying
```

Collections-Strings > Symbol

```
, otherSymbolOrString
	"Concatenate the receiver with otherSymbolOrString and
return the result.
	Overwritten to always return Symbol results,
	whether the argument is a Symbol or a String"

	^ (super , otherSymbolOrString) asSymbol
                                                    copying
```

TraitsV2 > TaAbstractComposition

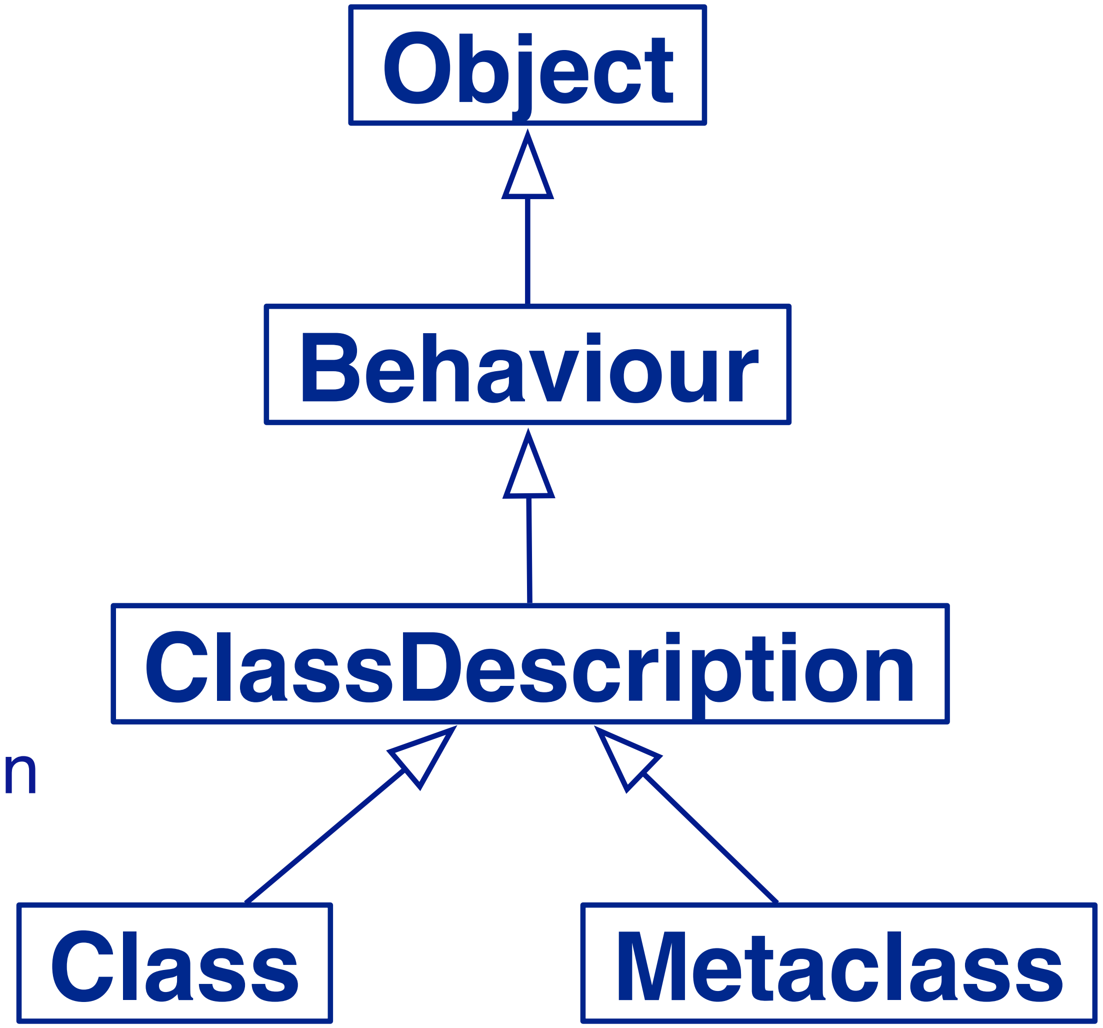
```
,                                                   operations
```

TraitsV2 > TaEmptyComposition

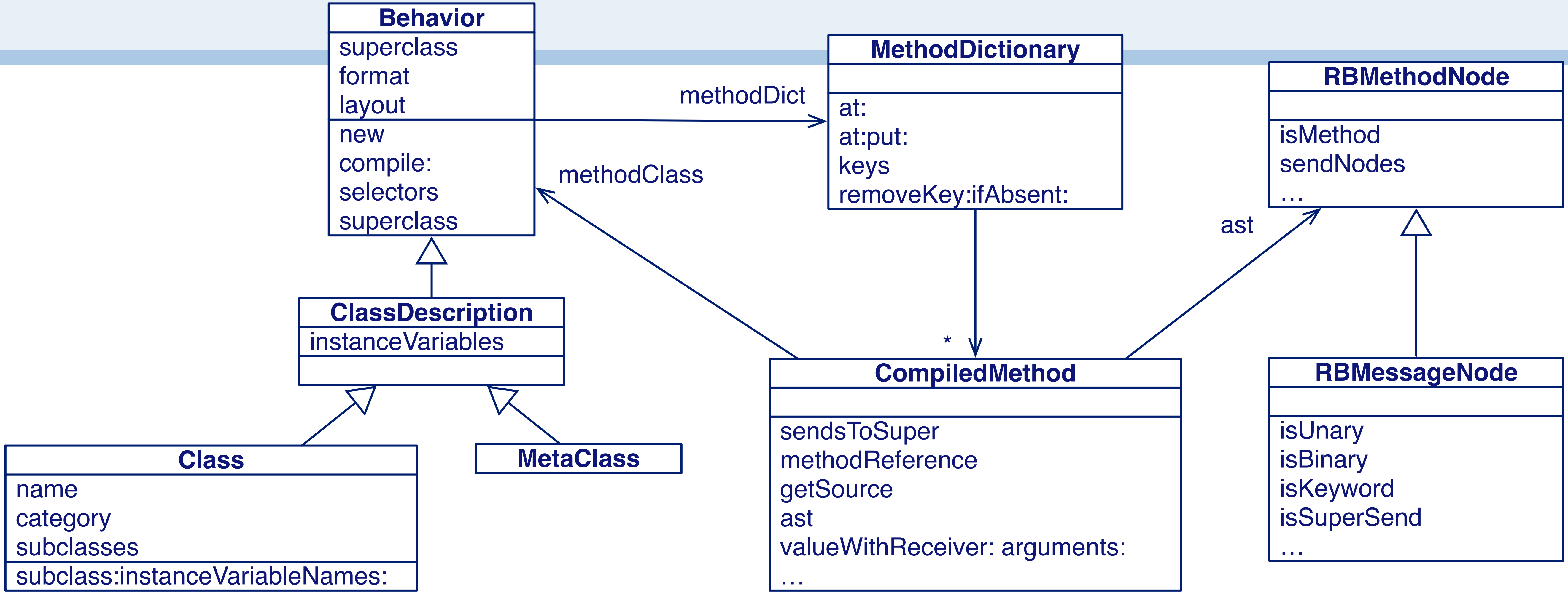
In Gt, queries are expressed with the help of composable filters

Recap: Classes are objects too

- Object
  - Root of inheritance
  - Default Behavior
  - Minimal Behavior
- Behavior
  - Essence of a class
  - Format, methodDict, superclass
- ClassDescription
  - Human representation and organization
- Class
  - Normal and anonymous classes
- Metaclass
  - Sole instance



Classes are Holders of CompiledMethods



Behavior
superclass
format
layout
new
compile:
selectors
superclass

ClassDescription
instanceVariables

Class
name
category
subclasses
subclass:instanceVariableNames:

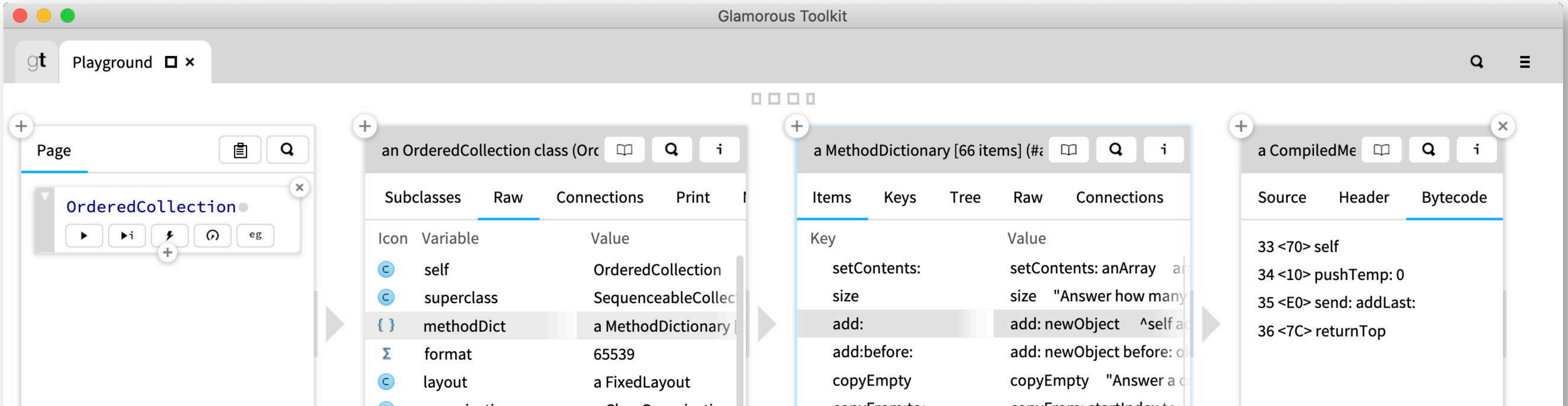
MetaClass

MethodDictionary
at:
at:put:
keys
removeKey:ifAbsent:

CompiledMethod
sendsToSuper
methodReference
getSource
ast
valueWithReceiver: arguments:
…

RBMethodNode
isMethod
sendNodes
…

RBMessageNode
isUnary
isBinary
isKeyword
isSuperSend
…



```
OrderedCollection
```

Icon	Variable	Value
C	self	OrderedCollection
C	superclass	SequenceableCollec
{ }	methodDict	a MethodDictionary
Σ	format	65539
C	layout	a FixedLayout

Key	Value
setContents:	setContents: anArray a
size	size "Answer how many
add:	add: newObject ^self a
add:before:	add: newObject before: o
copyEmpty	copyEmpty "Answer a c

```
33 <70> self
34 <10> pushTemp: 0
35 <E0> send: addLast:
36 <7C> returnTop
```

This simple metamodel allows us to navigate through the system.

If we inspect the class `OrderedCollection`, we can navigate to its method dictionary and to each of its `CompiledMethod` instances. Of course we can also navigate programmatically.

We can also navigate to the AST nodes (`RBNode`…) if we require more detailed information about the source code.

Note that there is a method >> defined in `Behavior` that returns a compiled method, so, for example `OrderedCollection>>#add:` will evaluate to the corresponding `CompiledMethod` object.

Given the metamodel, how is do you think >> is implemented?

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: #()
```

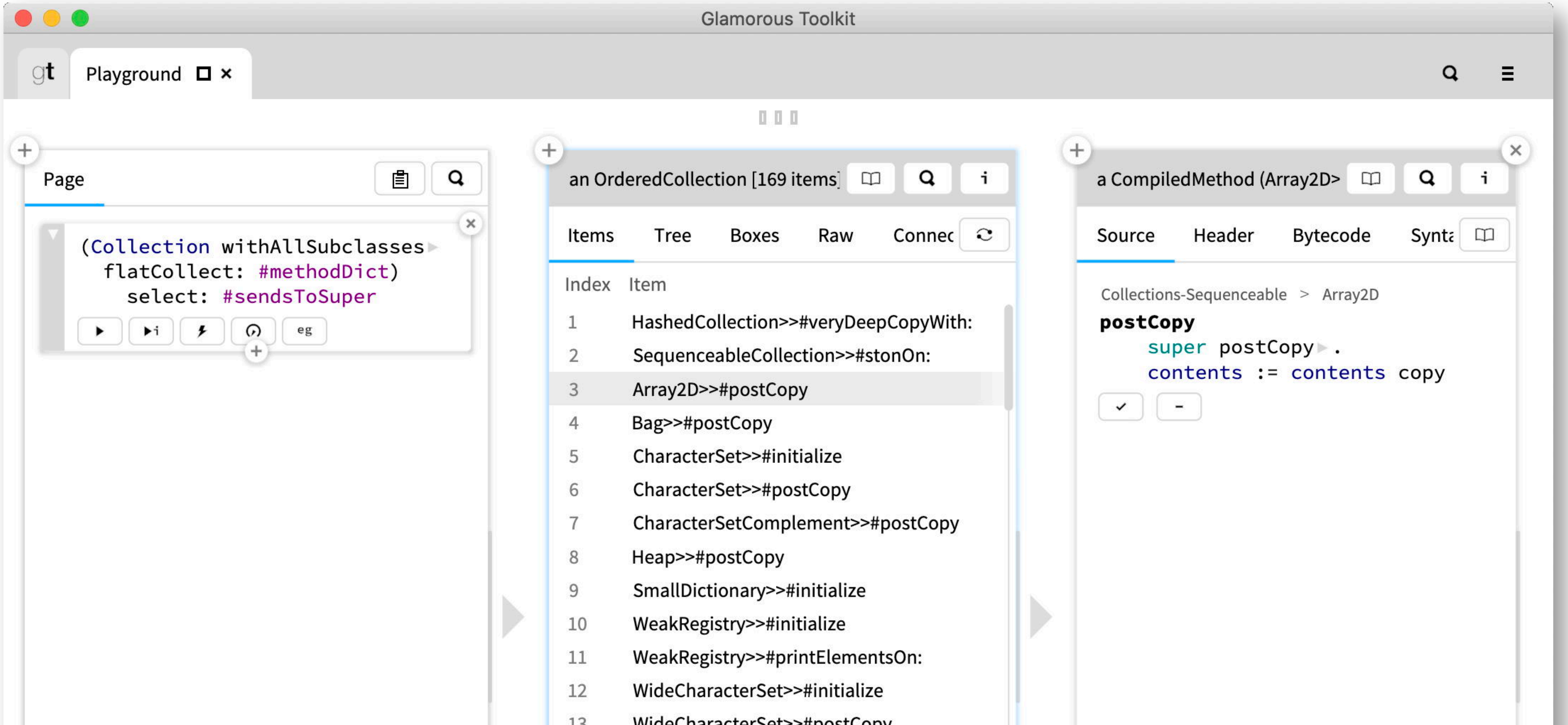
```
Error: key not found
```

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

```
120
```

Example: Finding super-sends within a hierarchy

```
(Collection withAllSubclasses flatCollect: #methodDict)
select: #sendsToSuper
```



```
(Collection withAllSubclasses
  flatCollect: #methodDict)
    select: #sendsToSuper
```

Index	Item
1	HashedCollection>>#veryDeepCopyWith:
2	SequenceableCollection>>#stonOn:
3	Array2D>>#postCopy
4	Bag>>#postCopy
5	CharacterSet>>#initialize
6	CharacterSet>>#postCopy
7	CharacterSetComplement>>#postCopy
8	Heap>>#postCopy
9	SmallDictionary>>#initialize
10	WeakRegistry>>#initialize
11	WeakRegistry>>#printElementsOn:
12	WideCharacterSet>>#initialize
13	WideCharacterSet>>#postCopy

Collections-Sequenceable > Array2D

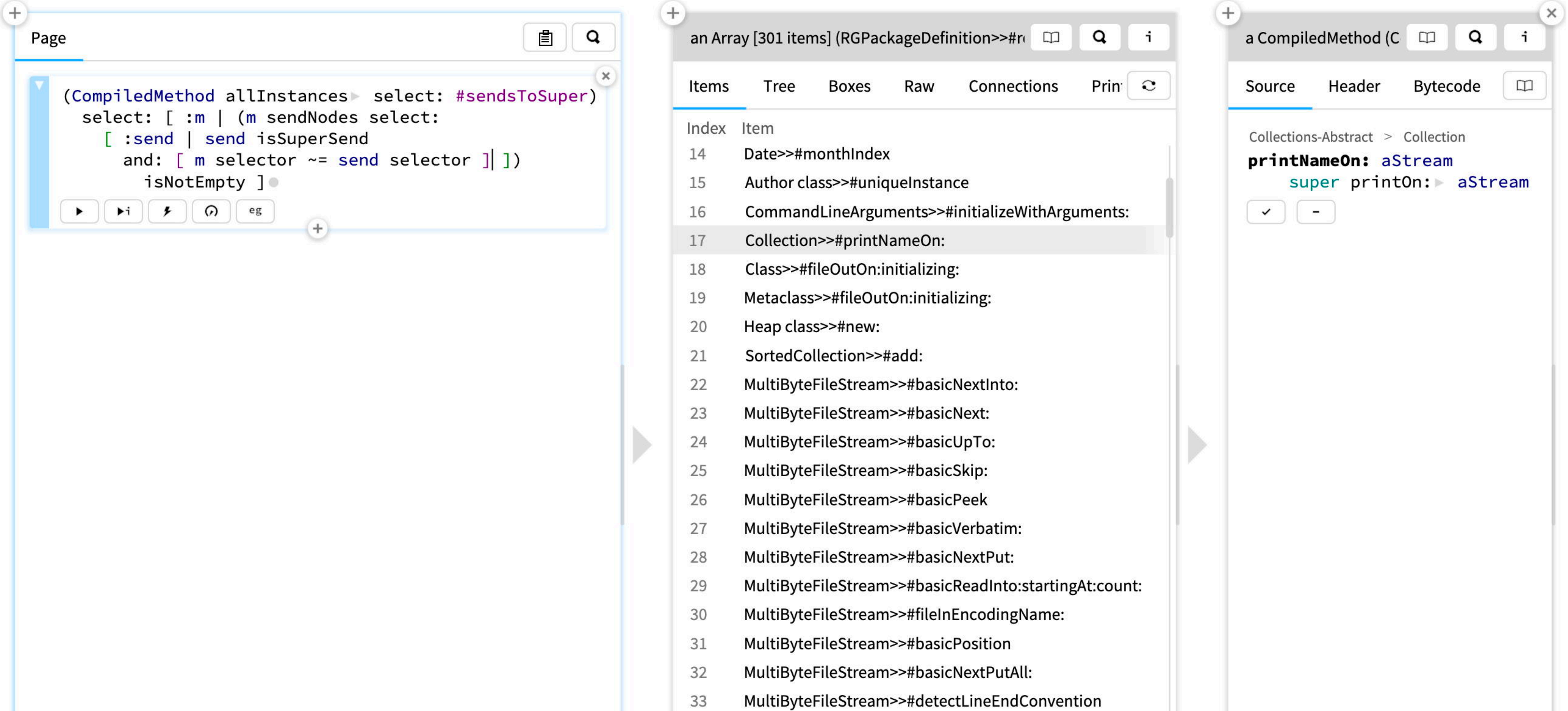
```
postCopy
    super postCopy .
    contents := contents copy
```

`Collection>>#flatCollect:` will collect a list of lists, and then flatten the result one level. Here we collect the method dictionaries of all the subclasses of `Collection` and flatten them, yielding a collection of `CompiledMethod` instances. (The method dictionaries will behave like sets of compiled methods in the flattening.)

Note that `#methodDict` and `#sendsToSuper` are duck-typed, behaving like query blocks.

Example: Finding super-sends to other methods

```
(CompiledMethod allInstances select: #sendsToSuper)
  select: [ :m | (m sendNodes select:
    [ :send | send isSuperSend
      and: [ m selector ~= send selector ] ])
        isNotEmpty ]
```



```
(CompiledMethod allInstances select: #sendsToSuper)
  select: [ :m | (m sendNodes select:
    [ :send | send isSuperSend
      and: [ m selector ~= send selector ] ])
        isNotEmpty ]
```

```
14   Date>>#monthIndex
15   Author class>>#uniqueInstance
16   CommandLineArguments>>#initializeWithArguments:
17   Collection>>#printNameOn:
18   Class>>#fileOutOn:initializing:
19   Metaclass>>#fileOutOn:initializing:
20   Heap class>>#new:
21   SortedCollection>>#add:
22   MultiByteFileStream>>#basicNextInto:
23   MultiByteFileStream>>#basicNext:
24   MultiByteFileStream>>#basicUpTo:
25   MultiByteFileStream>>#basicSkip:
26   MultiByteFileStream>>#basicPeek
27   MultiByteFileStream>>#basicVerbatim:
28   MultiByteFileStream>>#basicNextPut:
29   MultiByteFileStream>>#basicReadInto:startingAt:count:
30   MultiByteFileStream>>#fileInEncodingName:
31   MultiByteFileStream>>#basicPosition
32   MultiByteFileStream>>#basicNextPutAll:
33   MultiByteFileStream>>#detectLineEndConvention
```

```
printNameOn: aStream
    super printOn: aStream
```

First we select all methods that contain super sends using `CompiledMethod>>#sendToSuper`. Then we need more detailed information than the compiled method can provide, so we navigate to the message nodes of the AST using `#sendNodes`. We now select only the super send nodes, and then extract the subset where the message sent to super does not match the selector of the method itself. Finally inspect those methods for which this set is not empty.

Aside: The snippet `CompiledMethod allInstances` will also include any code evaluated in a Playground, but not yet garbage-collected. If you want to be sure that you only query the compiled methods belonging to classes, you can use the prepared query:

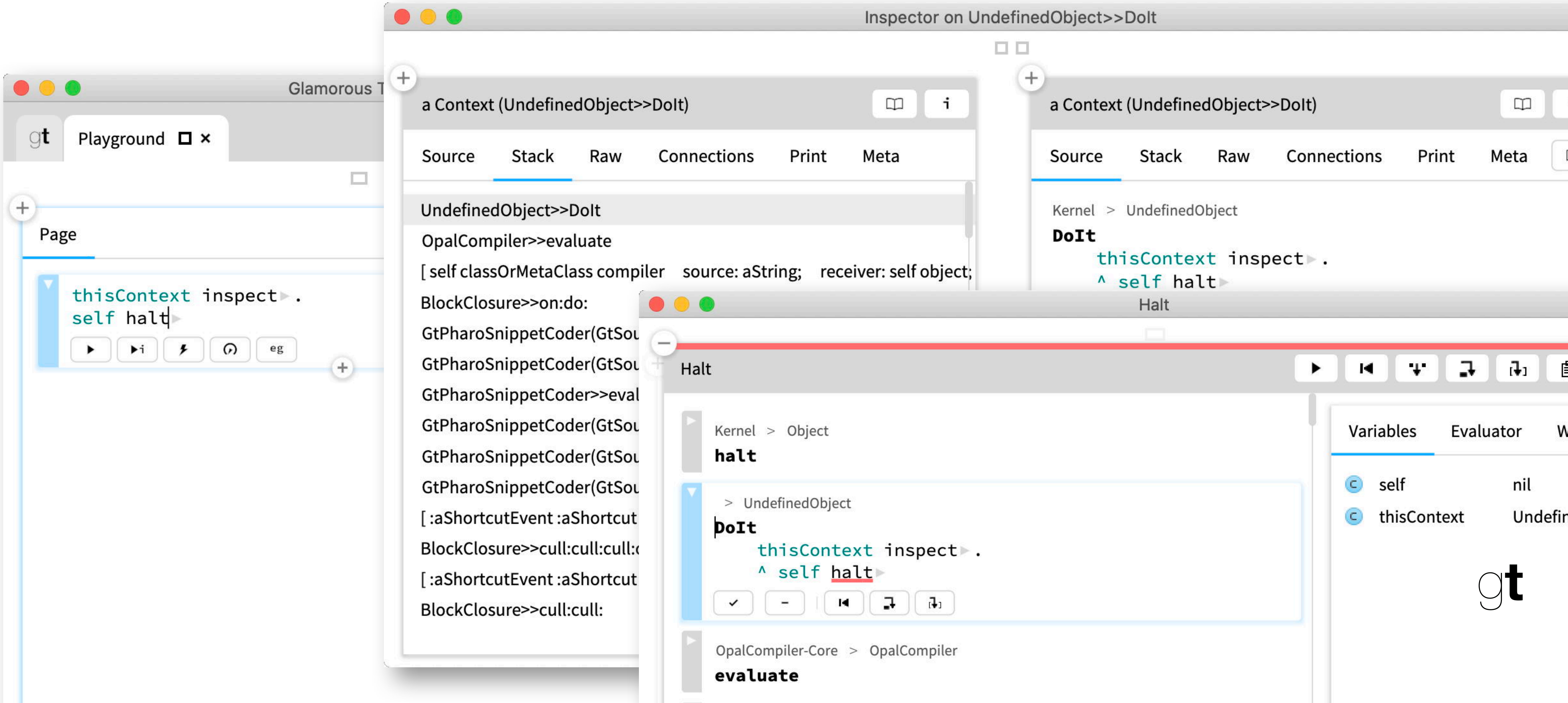
```
SystemNavigation default allMethods
```

Roadmap

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

Accessing the run-time stack

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



```
thisContext inspect .
self halt
```

UndefinedObject>>DoIt
OpalCompiler>>evaluate
[self classOrMetaClass compiler source: aString; receiver: self object;
BlockClosure>>on:do:
GtPharoSnippetCoder(GtSou
GtPharoSnippetCoder(GtSou
GtPharoSnippetCoder>>eval
GtPharoSnippetCoder(GtSou
GtPharoSnippetCoder(GtSou
GtPharoSnippetCoder(GtSou
[:aShortcutEvent :aShortcut
BlockClosure>>cull:cull:cull:
[:aShortcutEvent :aShortcut
BlockClosure>>cull:cull:

Kernel > UndefinedObject

```
DoIt
    thisContext inspect .
    ^ self halt
```

Kernel > Object

```
halt
```

> UndefinedObject

```
DoIt
    thisContext inspect .
    ^ self halt
```

OpalCompiler-Core > OpalCompiler

```
evaluate
```

self	nil
thisContext	Undefin

First start a Playground and evaluate:

```
thisContext inspect. self halt
```

An inspector and a debugger window will open. In the inspector run:

```
self stack inspect
```

This will open a second inspector on the stack, showing a view similar to that of the debugger (select the *Source* tab when you select a `Context` object in the stack inspector).

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 `Context`

```
InstructionStream variableSubclass: #Context
   instanceVariableNames: 'stackp method closureOrNil receiver'
   classVariableNames: 'PrimitiveFailToken QuickStep
SpecialPrimitiveSimulators TryNamedPrimitiveTemplateMethod'
   package: 'Kernel-Methods'
```

NB: In earlier versions of Pharo this class was called `MethodContext`. It inherits variables `pc` and `sender` from its superclass, `InstructionStream`.

Context

- `Context` holds all state associated with the execution of a `CompiledMethod`
  - pc: the program counter (from `InstructionStream`)
  - method: the `CompiledMethod` itself
  - receiver: the receiver object
  - sender: the previous `Context` or `BlockContext` (from `InstructionStream`)
    - *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>>haltIfCalledFrom: aSelector
   | context |
   context := thisContext.
   "walk up the stack looking for a Context with this selector"
   [ context sender isNil ]
      whileFalse: [ context := context sender.
         context selector = aSelector
            ifTrue: [ Halt signal ] ]
```

NB: `Object>>haltIf:` in Pharo is similar

A conditional breakpoint is one that triggers the debugger only if some condition holds. In this case we only want to halt if we are being called from some specific method, possibly indirectly. To determine this we need to search through the call stack for a context object corresponding to the given selector.

NB: Pharo provides conditional breakpoints that essentially work this way.

HaltDemo

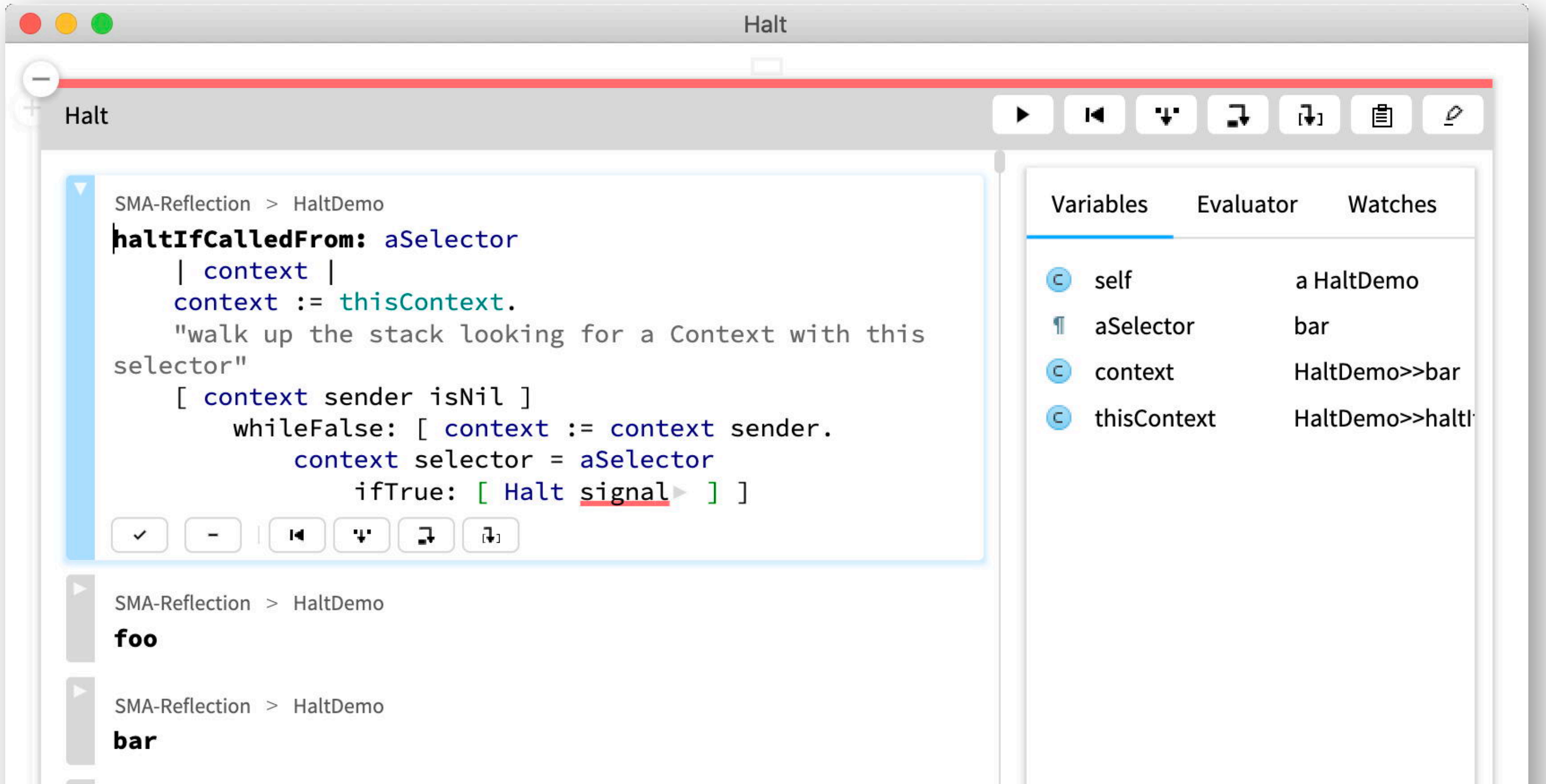
```
HaltDemo>>foo
  self haltIfCalledFrom: #bar.
  ^ 'foo'

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

```
HaltDemo new foo
```

```
'foo'
```

```
HaltDemo new bar
```



SMA-Reflection > HaltDemo

```
haltIfCalledFrom: aSelector
    | context |
    context := thisContext.
    "walk up the stack looking for a Context with this
selector"
    [ context sender isNil ]
        whileFalse: [ context := context sender.
            context selector = aSelector
                ifTrue: [ Halt signal ] ]
```

SMA-Reflection > HaltDemo
foo

SMA-Reflection > HaltDemo
bar

self	a HaltDemo
aSelector	bar
context	HaltDemo>>bar
thisContext	HaltDemo>>haltI

Roadmap

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

Overriding doesNotUnderstand:

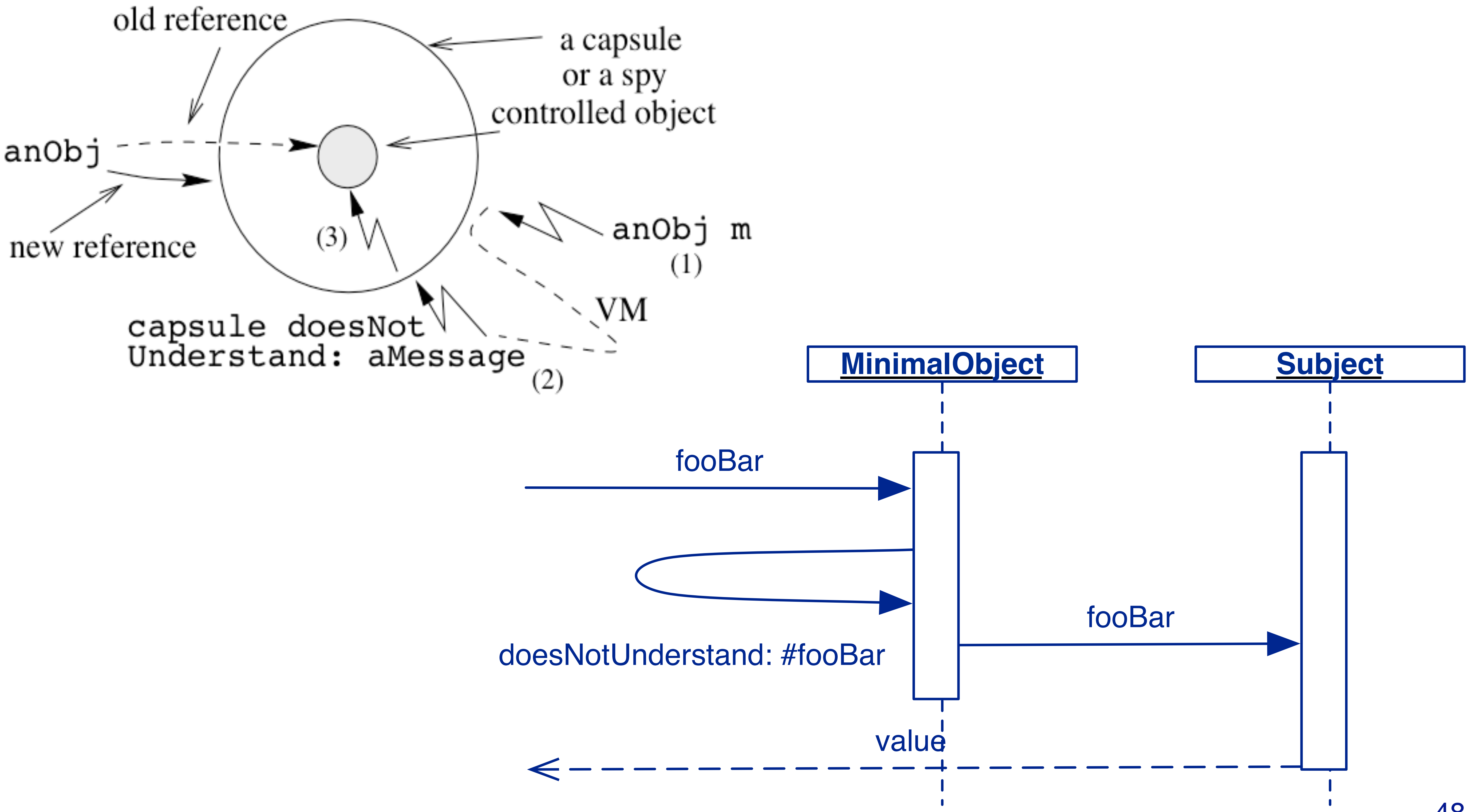
- Introduce a *Minimal Object*
  - Wraps a normal object
  - Does not understand very much
  - Redefines `doesNotUnderstand:`
  - Superclass is `nil` or `ProtoObject`
  - Uses `become:` to substitute the object to control

The idea of a “minimal object” is that it implements almost no methods, except for `doesNotUnderstand:`. Whenever it is sent any message, it will be reified and trapped by `doesNotUnderstand:`, at which point you do anything you like, for example, dynamically compile or load a default method, forward the message to another object, or fire up a different tool than the debugger.

A problem in Pharo (and most Smalltalk implementations) is that `Object` implements many methods. A minimal object should therefore not inherit from `Object` but from `nil`, or in Pharo, from `ProtoObject`.

In order to use a minimal object as a proxy for another object, we will use `become:`.

Minimal Object at Work



Here we see a minimal object used as a proxy or “wrapper” for another object. The message sent is not understood, causing it to be trapped. The minimal object then does its “proxy stuff” (such as logging), and forwards the message to the wrapped subject.

Logging message sends with a minimal object

```
ProtoObject subclass: #LoggingProxy
   instanceVariableNames: 'subject invocationCount'
   classVariableNames: ''
   package: 'SMA-Reflection'
```

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

```
LoggingProxy class>>for: aSubject
   ^ self new become: aSubject
```

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

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

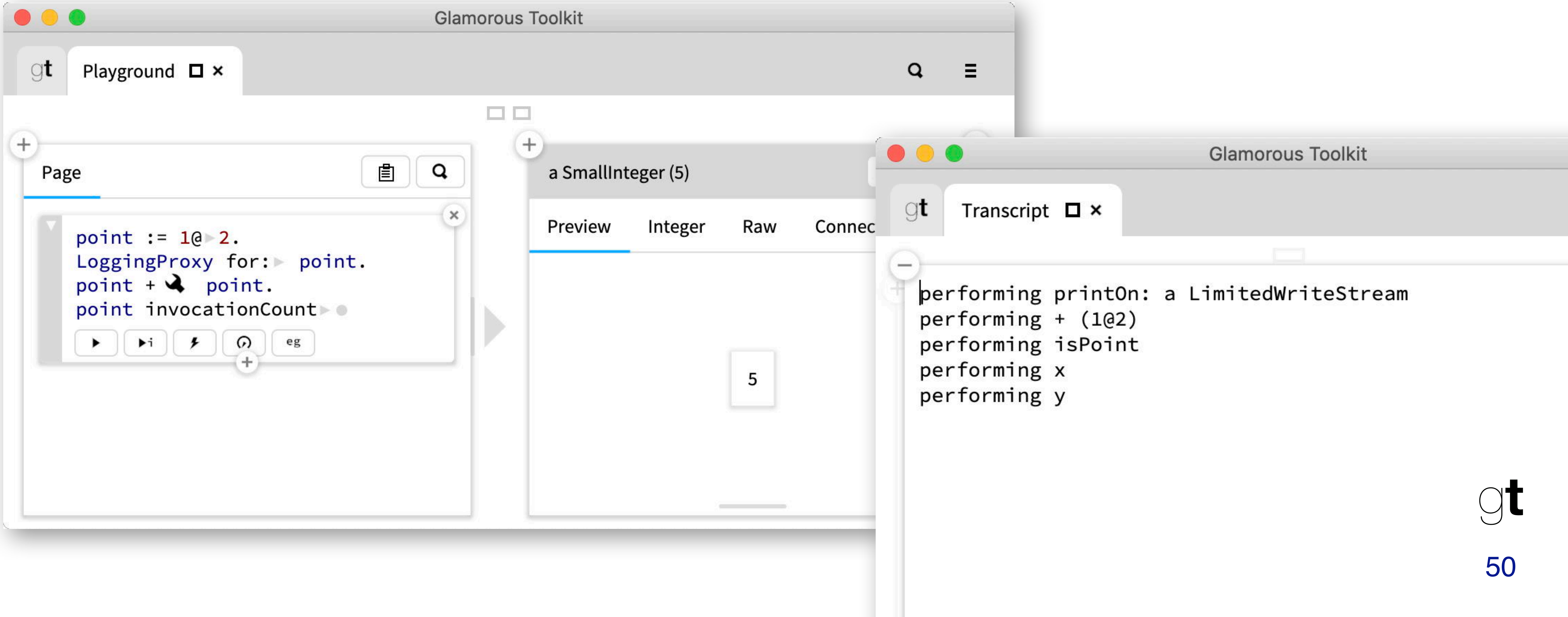
An initial `LoggingProxy` has itself as its subject. When we create an instance with

```
LoggingProxy for: aSubject
```

the references to the proxy and its subject are swapped, and `subject` will correctly refer to the subject, whereas any object that previously referred to the subject now refers to the proxy.

Using become: to install a proxy

```
point := 1@2.
LoggingProxy for: point.
point + point.
point invocationCount  5
```



```
point := 1@2.
LoggingProxy for: point.
point + point.
point invocationCount
```

```
performing printOn: a LimitedWriteStream
performing + (1@2)
performing isPoint
performing x
performing y
```

Computing the sum of two points causes the proxy to increase the invocation count.

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?

There are several shortcomings of proxies implemented as minimal objects.

First, *self-sends are not trapped*. See `LoggingProxyTest>>#testSelf` for a demonstration.

Although `Point>>#rectangle:` does two self-sends, these are not captured by the proxy.

Second, we can only wrap individual objects, not classes. We cannot use the logging proxy to log all messages sent to all instances of `Point`, without individually wrapping every `Point` object!

Finally, even though a minimal object has few methods, there may still be some conflicts with messages understood by the 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.

Here an accessor is generated if the ivar exists but no getter is defined. The same technique could be used, for example, to lazily load and compile code from a remote repository.

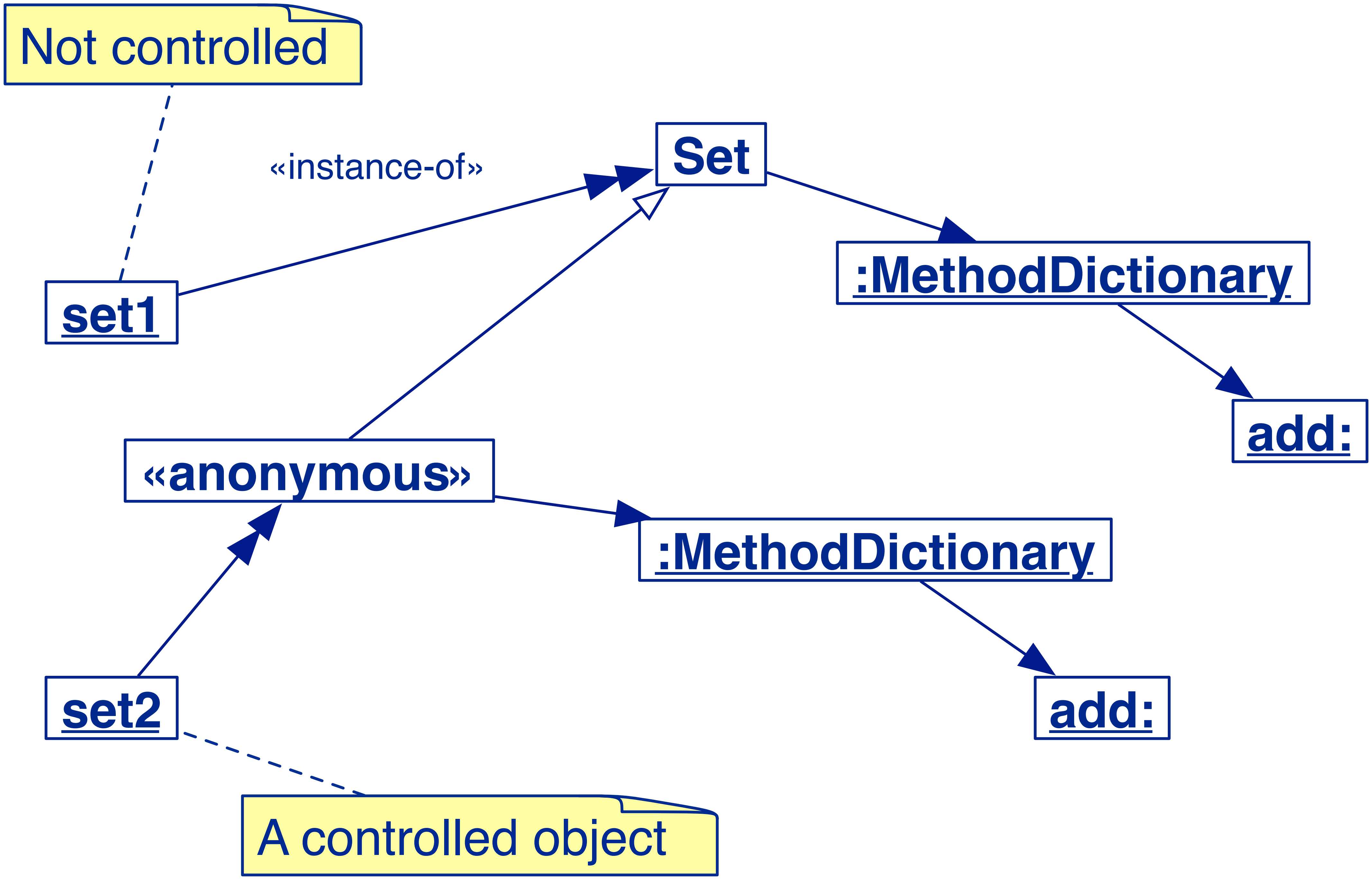
Roadmap

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

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



Not controlled

«instance-of»

Set

set1

:MethodDictionary

add:

«anonymous»

:MethodDictionary

add:

set2

A controlled object

In this scenario we introduce an *anonymous subclass* of `Set` that overrides the method `#add:`. The object `set1` is a normal instance of `Set`, while `set2` is an instance of the anonymous subclass. When we send the message `#add:` to `set2`, it is intercepted by the anonymous class, while all other messages are handled by `Set` as before.

Anonymous class in Pharo

```
anonClass := Class 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.
```

```
adding 2
```

Just for fun, inspect the anonymous class and navigate to the source code you have compiled. Note that although you can inspect the class (since it is an object), you cannot browse it.

Evaluation

- Either instance-based or group-based
- Selective control
- No self-send problem
- Good performance
- Transparent to the user
- Requires a bit of compilation

Roadmap

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

Method Substitution

First approach:

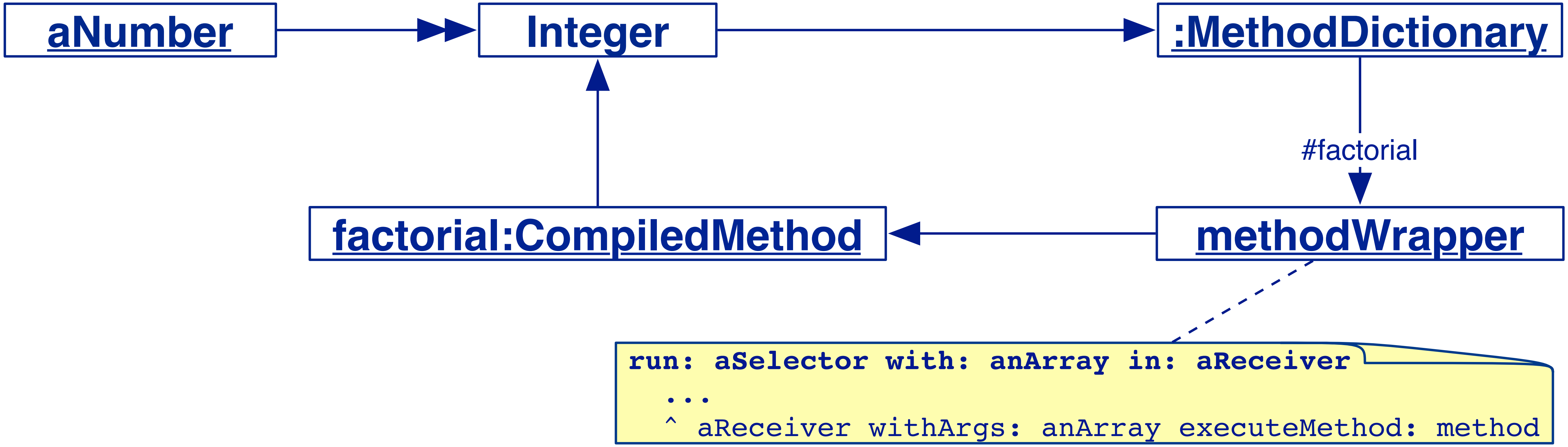
> Add methods with mangled names
  - — but the user can see them

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.



```
run: aSelector with: anArray in: aReceiver
  ...
  ^ aReceiver withArgs: anArray executeMethod: method
```

The Smalltalk VM expects that objects in a method dictionary are all either instances of CompiledMethod, or implement `#run:with:in`:. The arguments to `#run:with:in` are (1) the message selector, (2) the arguments array, and (3) the receiver.

Method wrappers exploit this to replace a compiled method with a wrapper implementing `#run:with:in`. The method wrapper can perform any action before or after evaluating the original compiled method (such as logging).

A LoggingMethodWrapper

```
LoggingMethodWrapper class>>on: aCompiledMethod
   ^ self new initializeOn: aCompiledMethod
```

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

```
LoggingMethodWrapper>>install
   method methodClass methodDictionary
      at: method selector
      put: self
```

`uninstall` is analogous …

```
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
```

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?
- How would you find all duck-typed methods in the image?