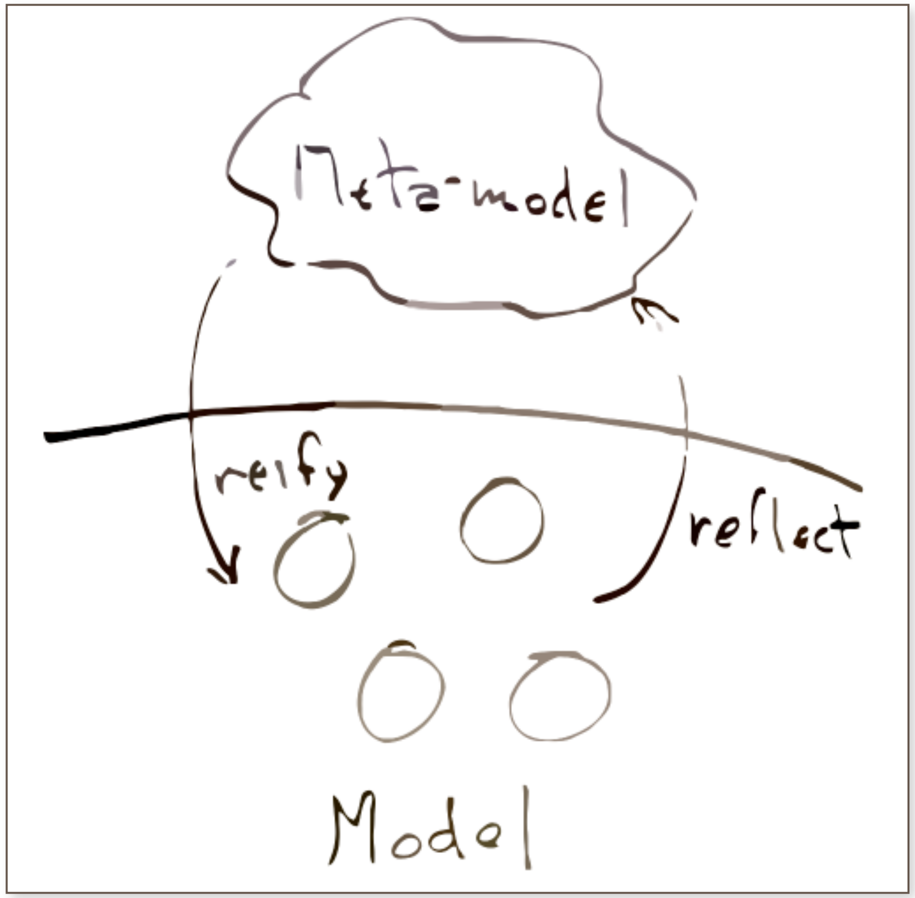
Smalltalk — a reflective language

Oscar Nierstrasz

UNIVERSITÄT BERN

Birds-eye view



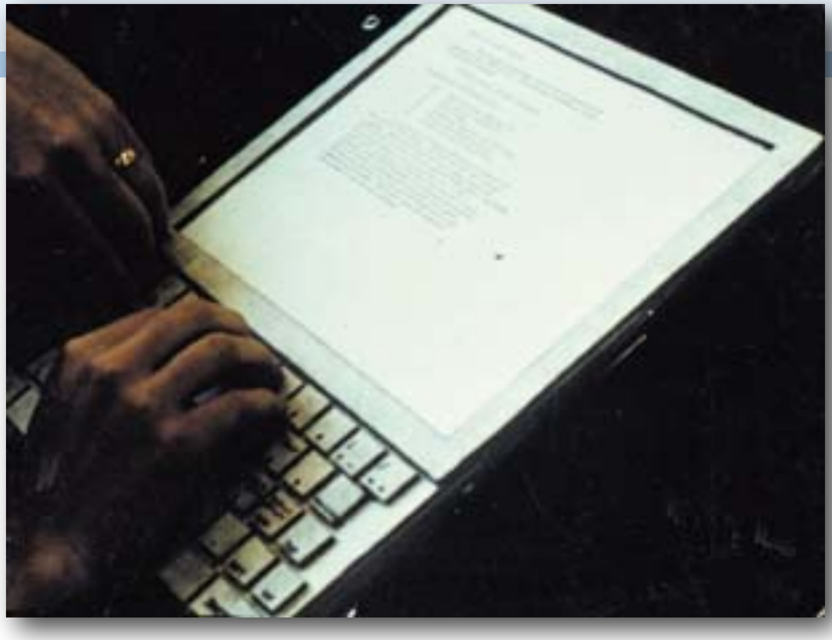
Smalltalk is still today one of the few fully reflective, fully dynamic, object-oriented development environments.

We will see how a simple, uniform object model enables live, dynamic, interactive software development.

Roadmap

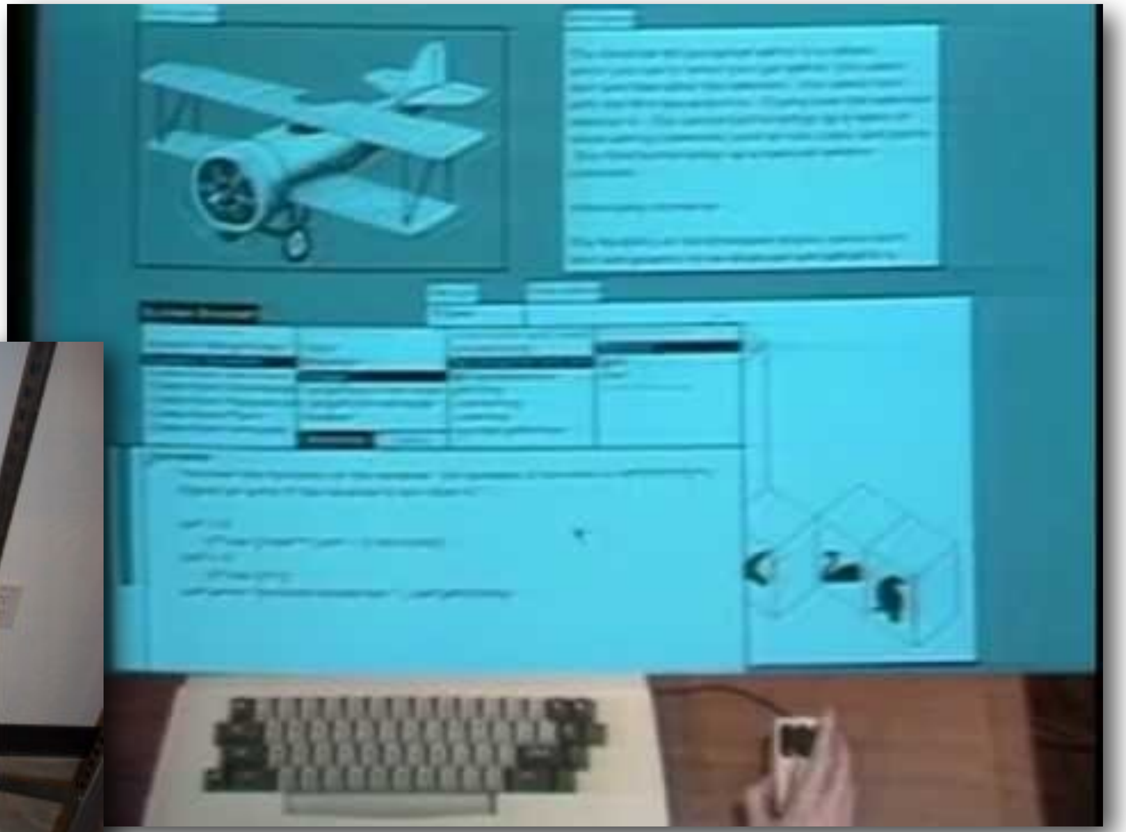
- > **Smalltalk Basics**
- > Demo: modeling Call Graphs

The origins of Smalltalk



Dynabook project (1968)

“simple things should be very simple, ... and, complex things should be very possible”



Alto — Xerox PARC (1973)

Smalltalk was invented to support the development of a new generation of graphical hardware devices. It was designed to be object-oriented “from the ground up”.

The DynaBook project imagined a future handheld device that could hold huge libraries of information. The Xerox PARC Smalltalk project started by building graphics workstations, with a view to a DynaBook-like device in the future.

http://esug.org/data/HistoricalDocuments/Smalltalk80/SmalltalkHistory.pdf

Excerpt from Alan Kay. *Personal Computing*. In “Meeting on 20 Years of Computing Science”, pp. 2-30, Instituto di Elaborazione della Informazione, Pisa, Italy, 1975:

> Smalltalk is a very simple, comprehensive way of simulating dynamic models. The built-in primitives of most programming languages (such as numbers, files, data structures, etc.), in Smalltalk, are actually simulations built from more comprehensive ideas, including states-in-process, communication using messages, and classes and instances.
>
> *Two of its basic goals are that simple things should be very simple*, one should not have to read a manual to do obvious things; *and, complex things should be very possible*, comprehensive interactive systems should be easily programmed without ‘hair or prayer’.

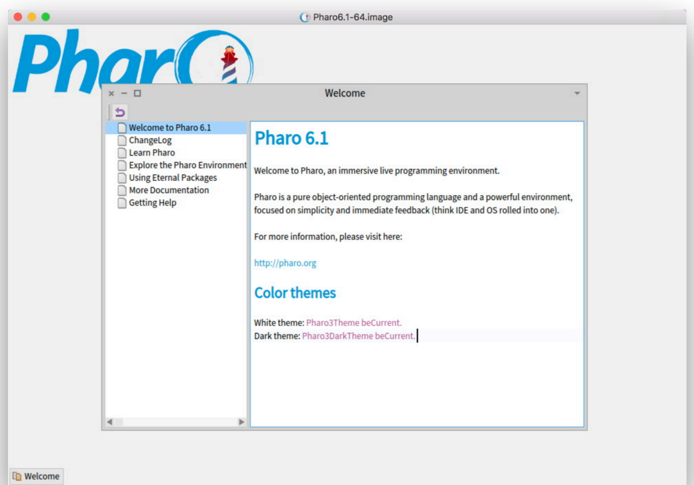
http://scgresources.unibe.ch/Literature/Smalltalk/Kay75a.pdf

What is interesting about Smalltalk?

- Everything is an object
- Everything happens by sending messages
- All the source code is there all the time
- You can’t lose code
- You can change everything
- You can change things without restarting the system
- The Debugger is your Friend

How does Smalltalk work?

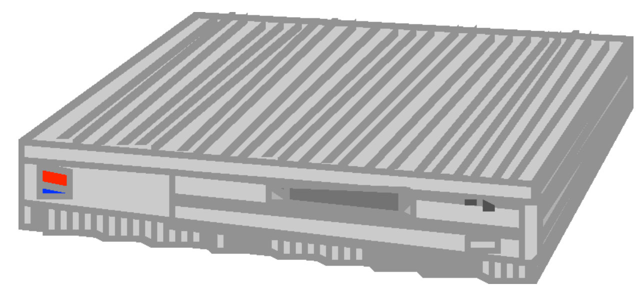
Image



+

Changes

Virtual machine



+

Sources

A running Smalltalk systems consists of 4 parts:

1. The *image* contains all the objects (the "heap")
2. The *changes file* logs all the source code changes you make (i.e., classes and methods)
3. The *virtual machine* executes bytecode and manages objects in the image
4. The "*sources file*" contains all system source code of the base image

Note that the image and changes file must be kept together. Although the VM and sources may be shared by multiple users, nowadays all four files are commonly kept together within a single "one-click" application.

Don’t panic!

New Smalltalkers often think they need to understand all the details of a thing before they can use it.

Try to answer the question

“How does this work?”

with

“I don’t care”.

— Alan Knight. Smalltalk Guru

This is actually a paraphrase of:

Try not to care — Beginning Smalltalk programmers often have trouble because they think they need to understand all the details of how a thing works before they can use it. This means it takes quite a while before they can master `Transcript show: 'Hello World'`*. One of the great leaps in OO is to be able to answer the question "How does this work?" with "I don't care".*

http://alanknightsblog.blogspot.ch/2011/10/principles-of-oo-design-or-everything-i.html

Two things to remember ...

Everything is an object

Integers, Booleans, classes, methods, compiled methods, the tools, you name it, they are all objects. When you finally understand deeply that everything in the Smalltalk system is an object, you start to think differently about how to interact with that world.

Here's a relevant fake quote from *A Brief, Incomplete, and Mostly Wrong History of Programming Languages:*

> *1980 — Alan Kay creates Smalltalk and invents the term "object oriented." When asked what that means he replies, "Smalltalk programs are just objects." When asked what objects are made of he replies, "objects." When asked again he says "look, it's all objects all the way down. Until you reach turtles."*

http://james-iry.blogspot.ch/2009/05/brief-incomplete-and-mostly-wrong.html

Everything happens by sending messages

To understand why something happens, figure out what message was sent. One consequence of this is that anything can be done programmatically. You just have to figure out what objects are involved and what messages they understand.

The Smalltalk object model

> **Every object is an instance of one class**
  — ... which is also an object
  — Single inheritance
> **Dynamic binding**
  — All variables are dynamically typed and bound
> **State is private to objects**
  — “Protected” for subclasses
  — Encapsulation boundary is the object, not the class!
> **Methods are public**
  — “private” methods by convention only

Smalltalk Syntax

Every expression is a message send

> Unary messages

```
5 factorial
Transcript cr
```

> Binary messages

```
3 + 4
'hi', ' there'
```

> Keyword messages

```
Transcript show: 'hello world'
2 raisedTo: 32
'hello' at: 1 put: $y
```

Precedence

First unary, then binary, then keyword:

```
2 raisedTo: 1 + 3 factorial
```

128

Same as:

```
2 raisedTo: (1 + (3 factorial))
```

Use parentheses to force order:

```
1 + 2 * 3
1 + (2 * 3)
```

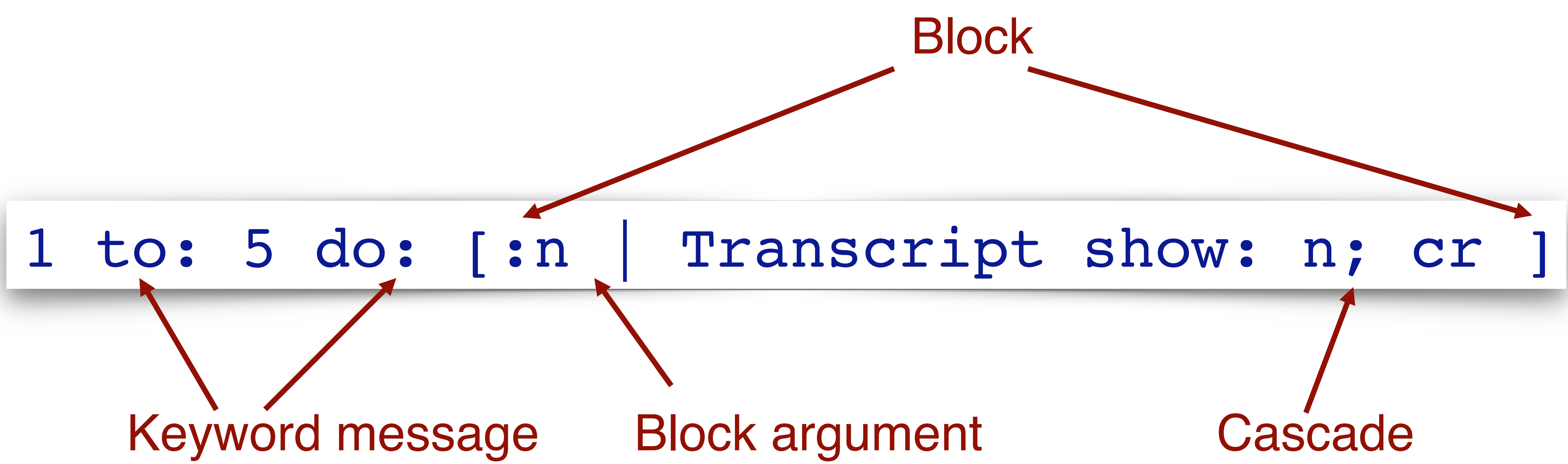
9 (!)
7

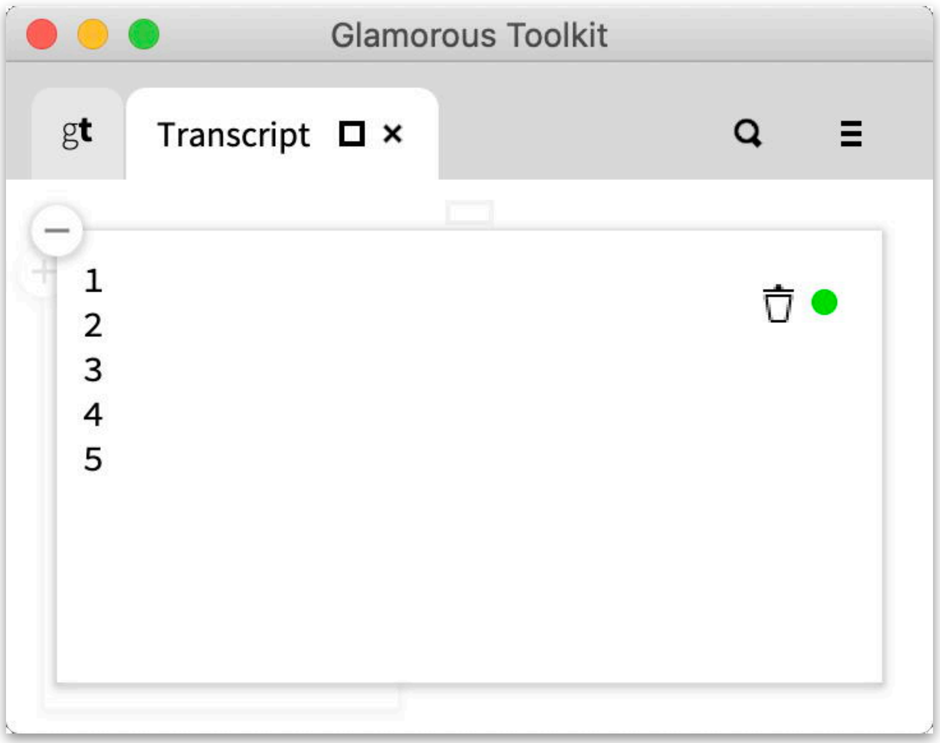
Literals and constants

Strings & Characters	`'hello'   $a`
Numbers	`1   3.14159`
Symbols	`#yadayada`
Arrays	`#(1 2 3)`
Pseudo-variables	`self super`
Constants	`true false`

There are only 6 keywords in Smalltalk: `self`, `super`, `true`, `false`, `nil` and `thisContext`. (This last one we will encounter in the lecture on reflection.)

Blocks


Block
1 to: 5 do: [:n | Transcript show: n; cr]
Keyword message
Block argument
Cascade


Glamorous Toolkit
gt
Transcript
1
2
3
4
5

Roadmap

> Smalltalk Basics
> **Demo: modeling Call Graphs**
> — The call graph model
> — Pharo and Glamorous Toolkit
> — Implementing the CallGraph class
> — Version control in Pharo
> — Modeling Calls, Methods and Classes
> — The Debugger is your Friend!
> — Expressing queries

Roadmap

- Smalltalk Basics
- **Demo: modeling Call Graphs**
  - **The call graph model**
  - Pharo and Glamorous Toolkit
  - Implementing the CallGraph class
  - Version control in Pharo
  - Modeling Calls, Methods and Classes
  - The Debugger is your Friend!
  - Expressing queries

Task: analyze call graph logs from Javassist

Return type — "Owner" — **Called method** — Formal argument types

```
|java.lang.String:org.clapper…HTMLUtil.convertCharacterEntities:java.lang.String
|STATIC_METHOD
|java.lang.String
|org.clapper…convertHTMLEntities:47
…
```

Dynamic argument types

Receiver class (or "static")

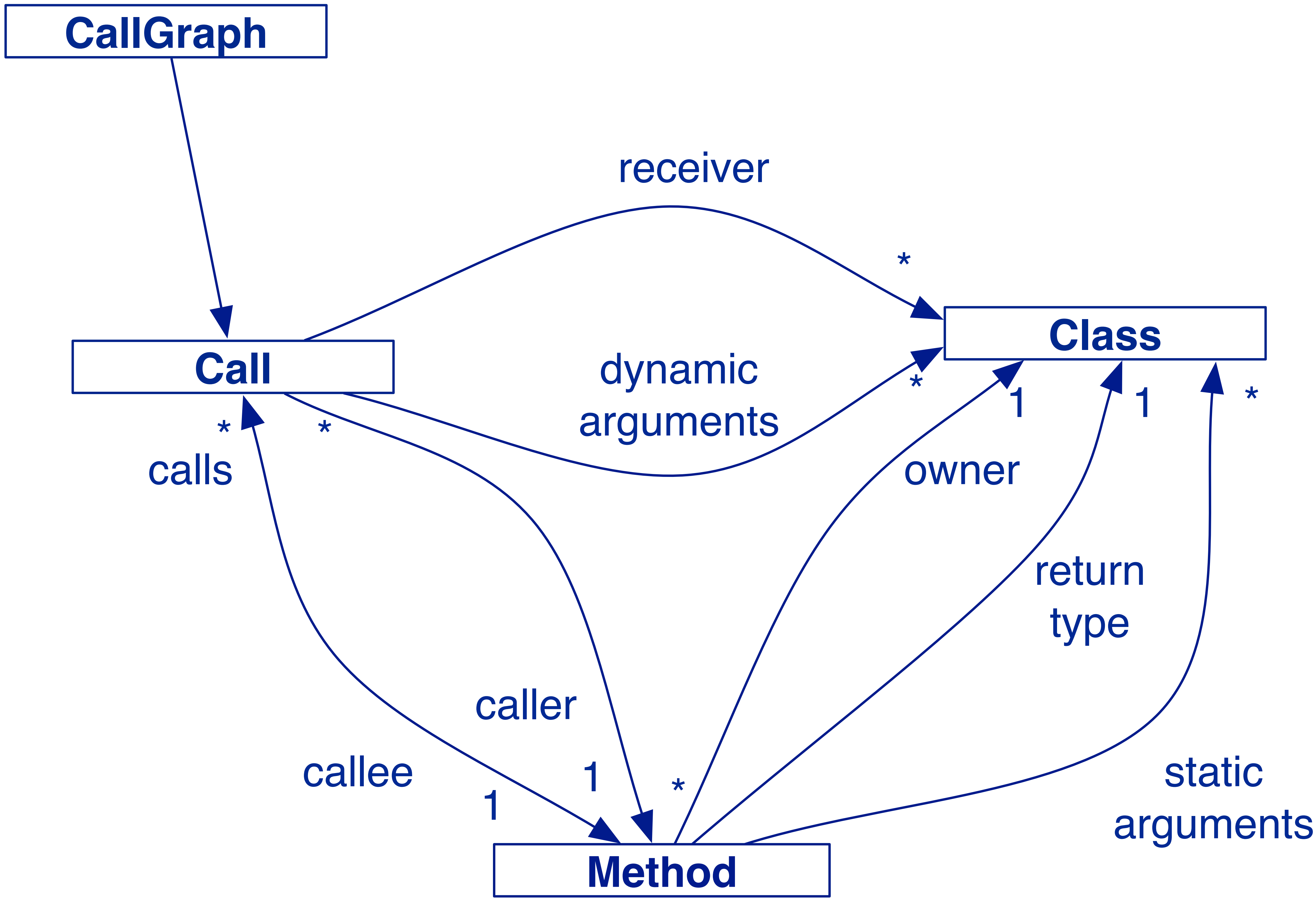
Call site (method + line #)

```
|java.lang.String:org.clapper.util.html.HTMLUtil.convertCharacterEntities:java.lang.String|STATIC_METHOD|
  java.lang.String|org.clapper.util.html.test.HTMLEntitiesTest.convertHTMLEntities:47
|org.clapper.util.text.XStringBufBase:org.clapper.util.text.XStringBufBase.append:java.lang.String|
  org.clapper.util.text.XStringBuffer|java.lang.String|org.clapper.util.html.HTMLUtil.convertCharacterEntities:240
|java.lang.Appendable:org.clapper.util.text.XStringBuffer.getBufferAsAppendable|org.clapper.util.text.XStringBuffer|
   |org.clapper.util.text.XStringBufBase.append:469
|java.lang.String:org.clapper.util.html.HTMLUtil.convertEntity:java.lang.String|STATIC_METHOD|java.lang.String|
  org.clapper.util.html.HTMLUtil.convertCharacterEntities:253
|java.util.ResourceBundle:org.clapper.util.html.HTMLUtil.getResourceBundle|STATIC_METHOD|  |
  org.clapper.util.html.HTMLUtil.convertEntity:424
|java.lang.String:org.clapper.util.html.HTMLUtil.textFromHTML:java.lang.String|STATIC_METHOD|java.lang.String|
  org.clapper.util.html.test.HTMLEntitiesTest.textFromHTML:82
```

The data is generated from some Java code instrumented using Javassist and written to a mysql log. This is a dump of the resulting mysql table.

http://jboss-javassist.github.io/javassist/

How to reconstruct the model from the log?



Our goal is to reconstruct from the run-time log an object-oriented model of the call graph that can be queried to asnwer questions about the calling relationships.

This UML class diagram summarized the information encoded in the log:

A `Method` is implemented in a `Class` (its owner). The arguments and return types are also statically-known classes.

A `Call` is a run-time activation of a specific `Method` (caller) calling another `Method` (callee). The receiver and the arguments are instances of specific classes (which may not be identical to the owner or static arguments of the caller!).

There may be multiple `Call`s of the same `Method`.

Questions of interest

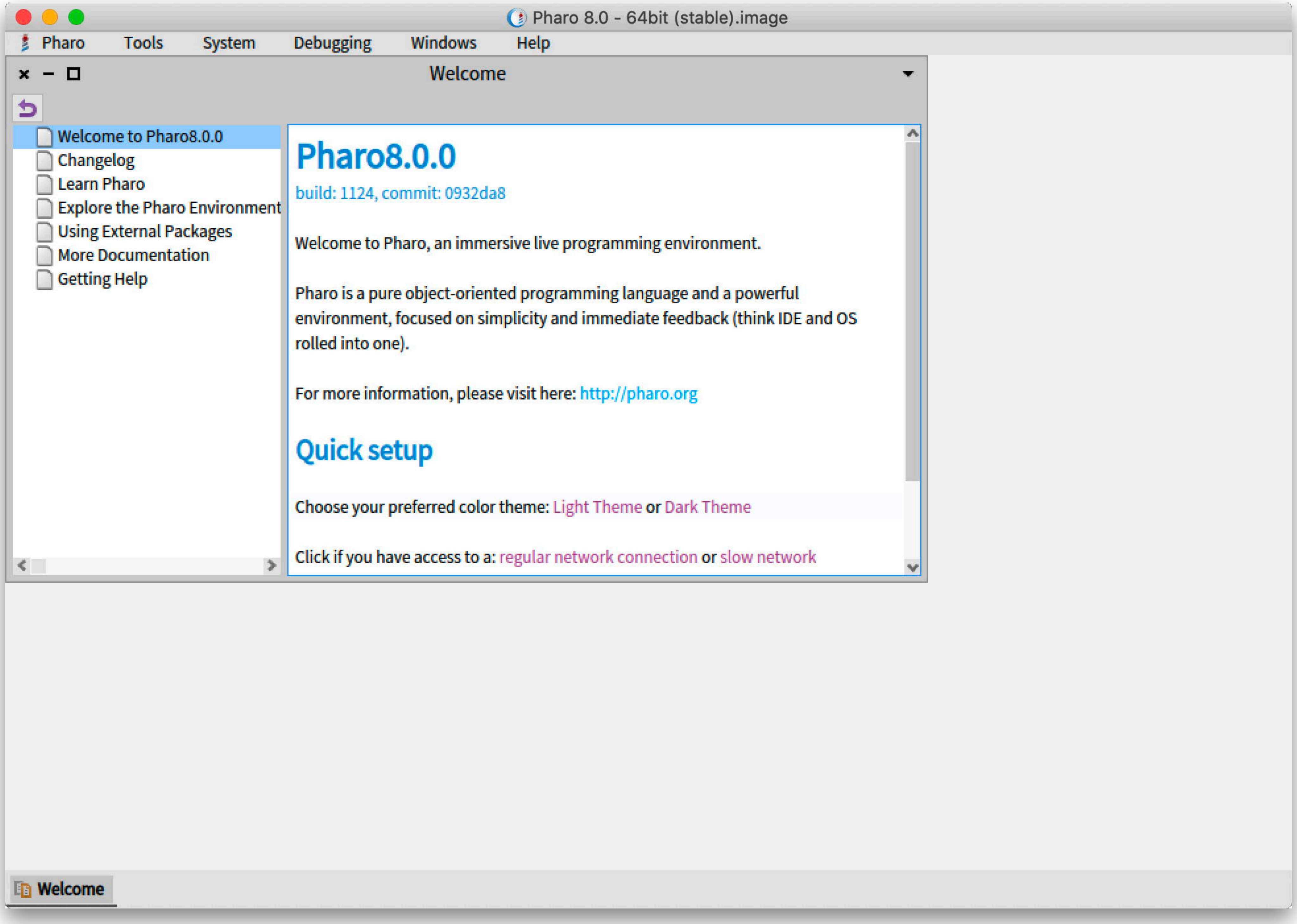
- How many calls are there?
- How many methods are called?
- How many classes are accessed?
- Which methods are static?
- Which methods are called most frequently?
- What is the depth of the call graph?
- Which methods are called by more than one caller?
- Which methods are potentially polymorphic? (multiple receivers/implementations)
- What are the polymorphic call sites? (methods called with different receiver/argument types)
- …

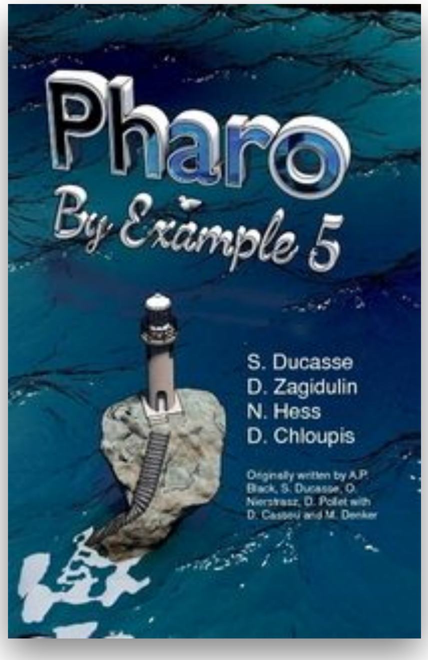
We would like to build up the model in such a way that such questions can easily be posed as queries, i.e., expressions over the objects representing the model.

Roadmap

> Smalltalk Basics
> **Demo: modeling Call Graphs**
  — The call graph model
  — **Pharo and Glamorous Toolkit**
  — Implementing the CallGraph class
  — Version control in Pharo
  — Modeling Calls, Methods and Classes
  — The Debugger is your Friend!
  — Expressing queries

Pharo — a modern Smalltalk





Pharo is an open-source evolution of Smalltalk-80.

Download it from:

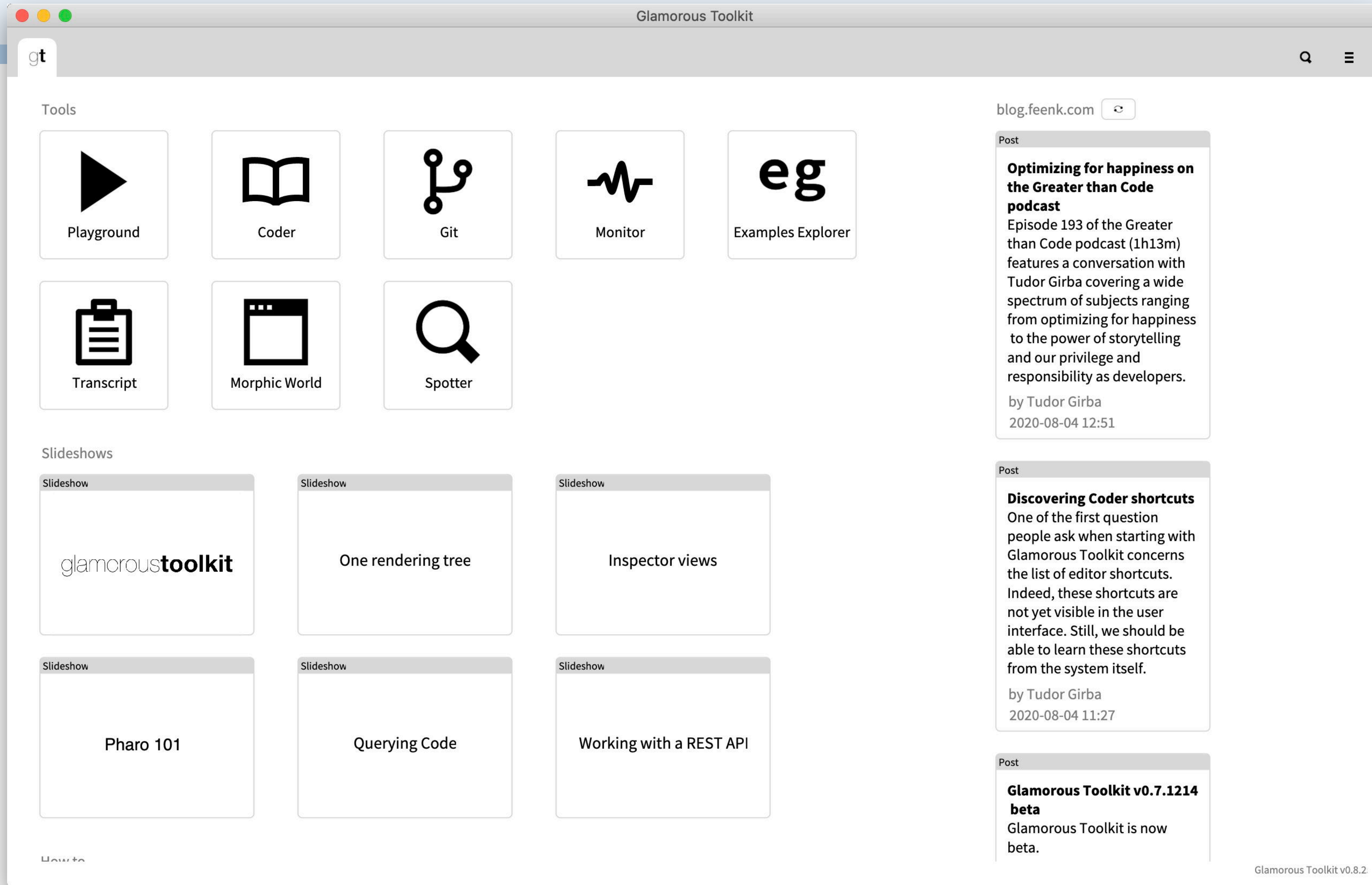
http://pharo.org

To learn how to use Pharo, start with the open-source book, *Pharo by Example:*

http://books.pharo.org

To learn about more advanced features, continue with *Deep into Pharo*

Glamorous Toolkit — a moldable Smalltalk



Gt is a "moldable" development environment built on Pharo with native windows, software analysis support, and a visualization engine

GT offers a new graphical framework and a new set of tools for software development on top of Pharo.

https://gtoolkit.com/download/

NB: Although GT is quite mature, it does not yet offer replacements for all Pharo tools and features, so it is always possible to escape the the "Morphic World" to access the traditional tool set.

As an alternative to the following slides, you can download and run a live version of the demo.

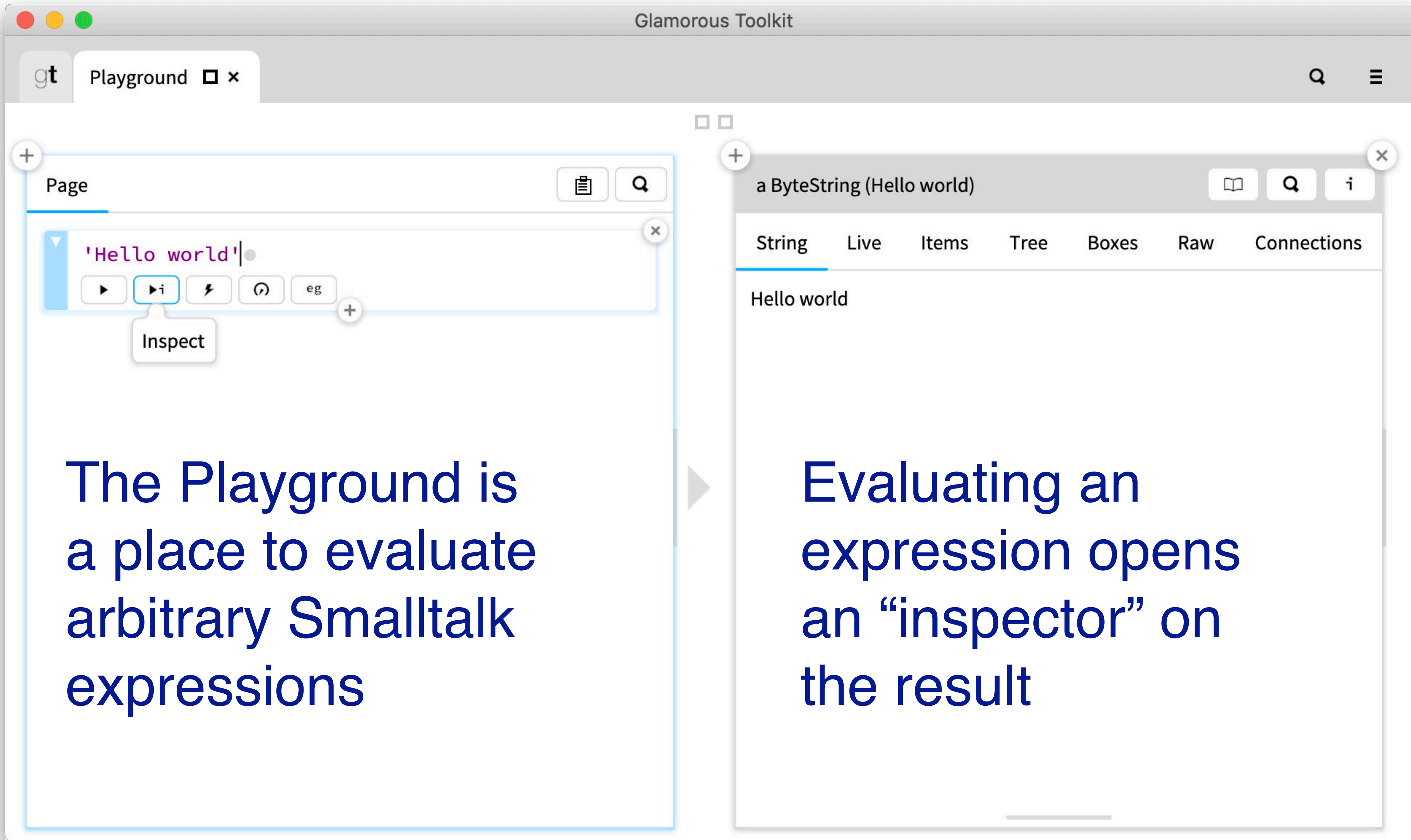
From a Gt Playground, run the following snippet to install the demo examples:

```
Metacello new baseline: 'SMAForGt';
  repository: 'github://onierstrasz/sma-examples/src';
  load.
```

And run this snippet to start open the slideshow demos:

```
SMAForGt openSlideshowsOverview
```

The Playground



The Playground is a place to evaluate arbitrary Smalltalk expressions

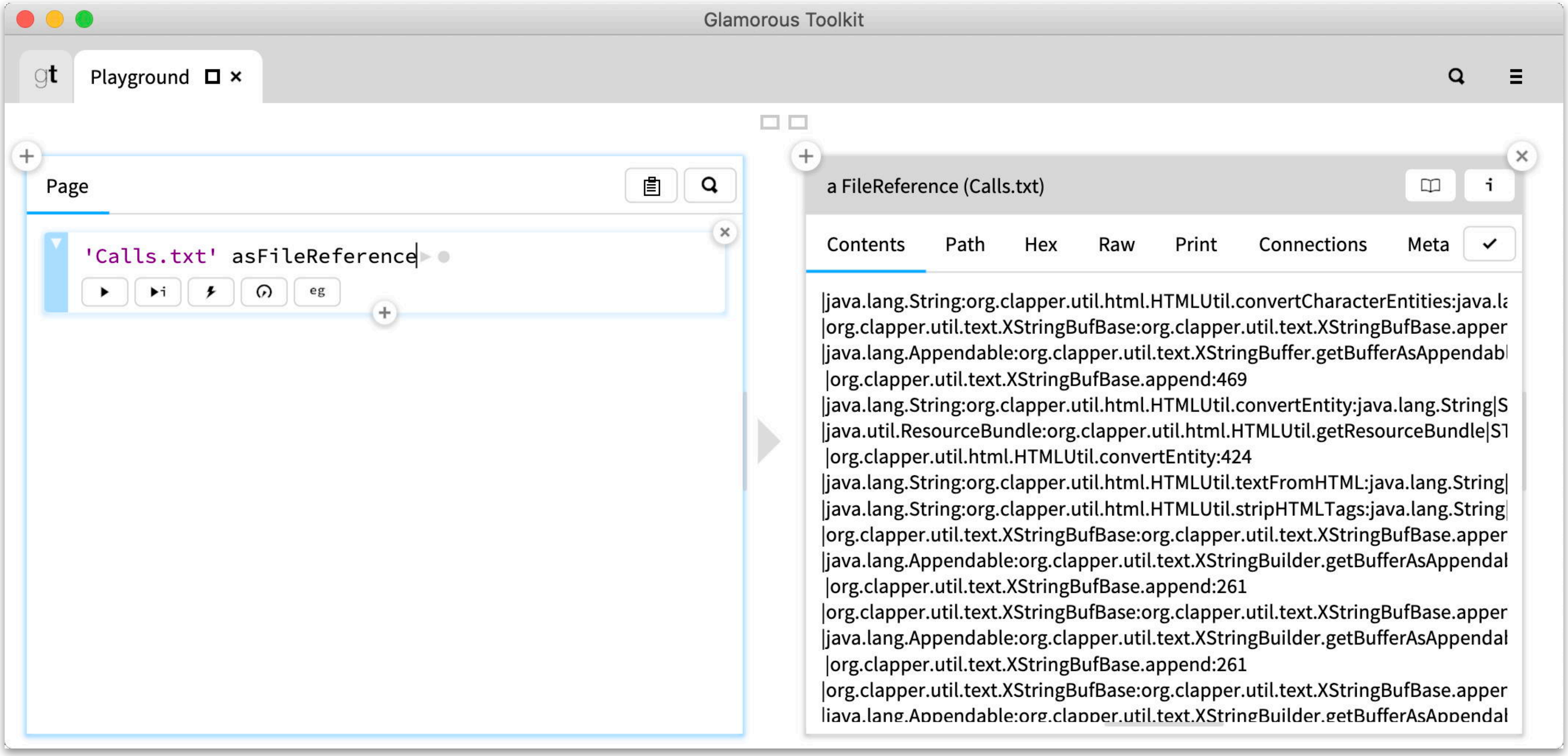
Evaluating an expression opens an “inspector” on the result

You can select an expression in the Workspace and “do it”, “print it”, “inspect it”, or simply “do it and go”.

NB: use the keyboard shortcuts instead of the menu or buttons!

Accessing a file from a Playground

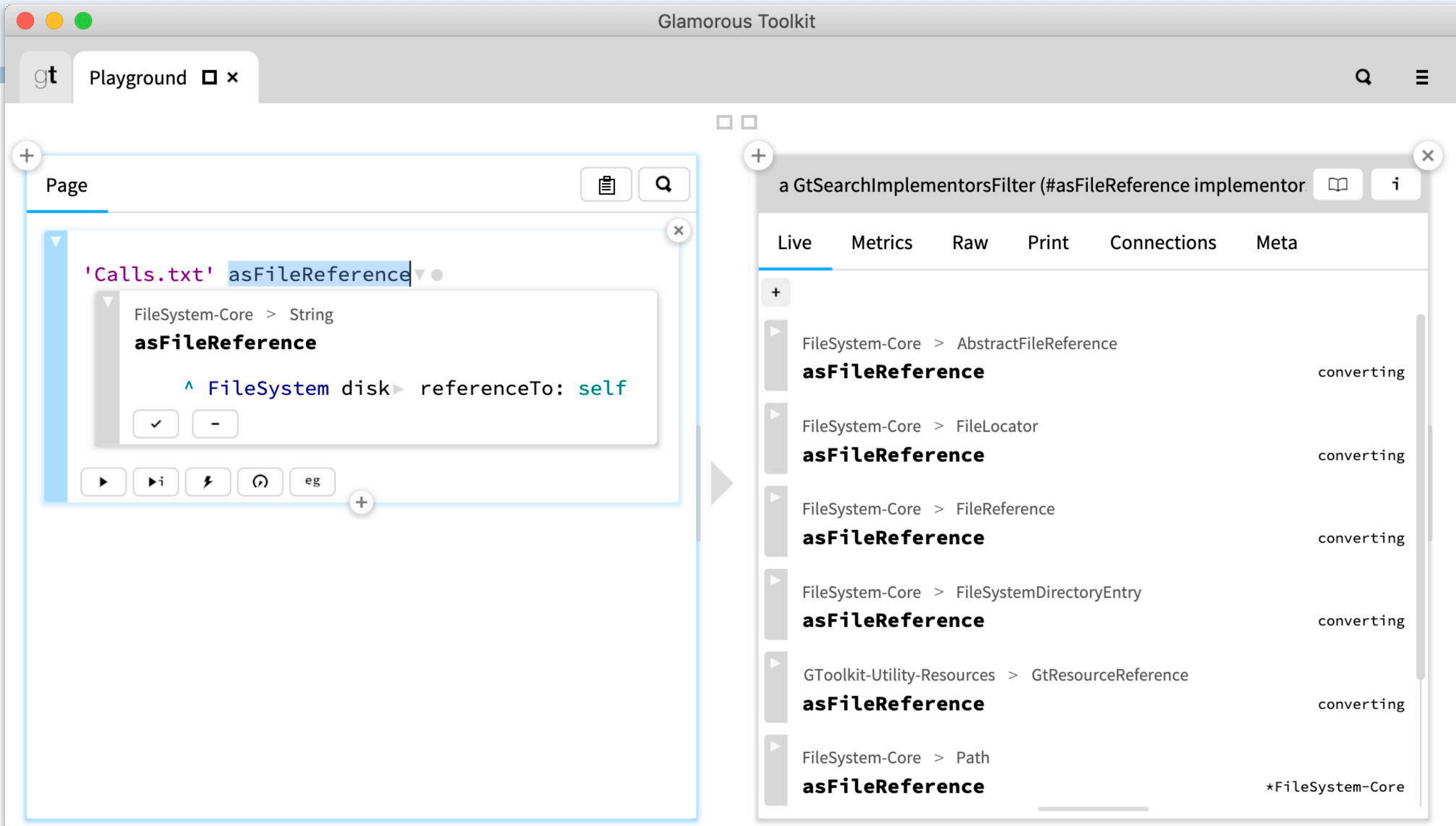
We can open the file named "Calls.txt" and extract its contents as a `String` object



We should encapsulate this data in a ClassGraph object

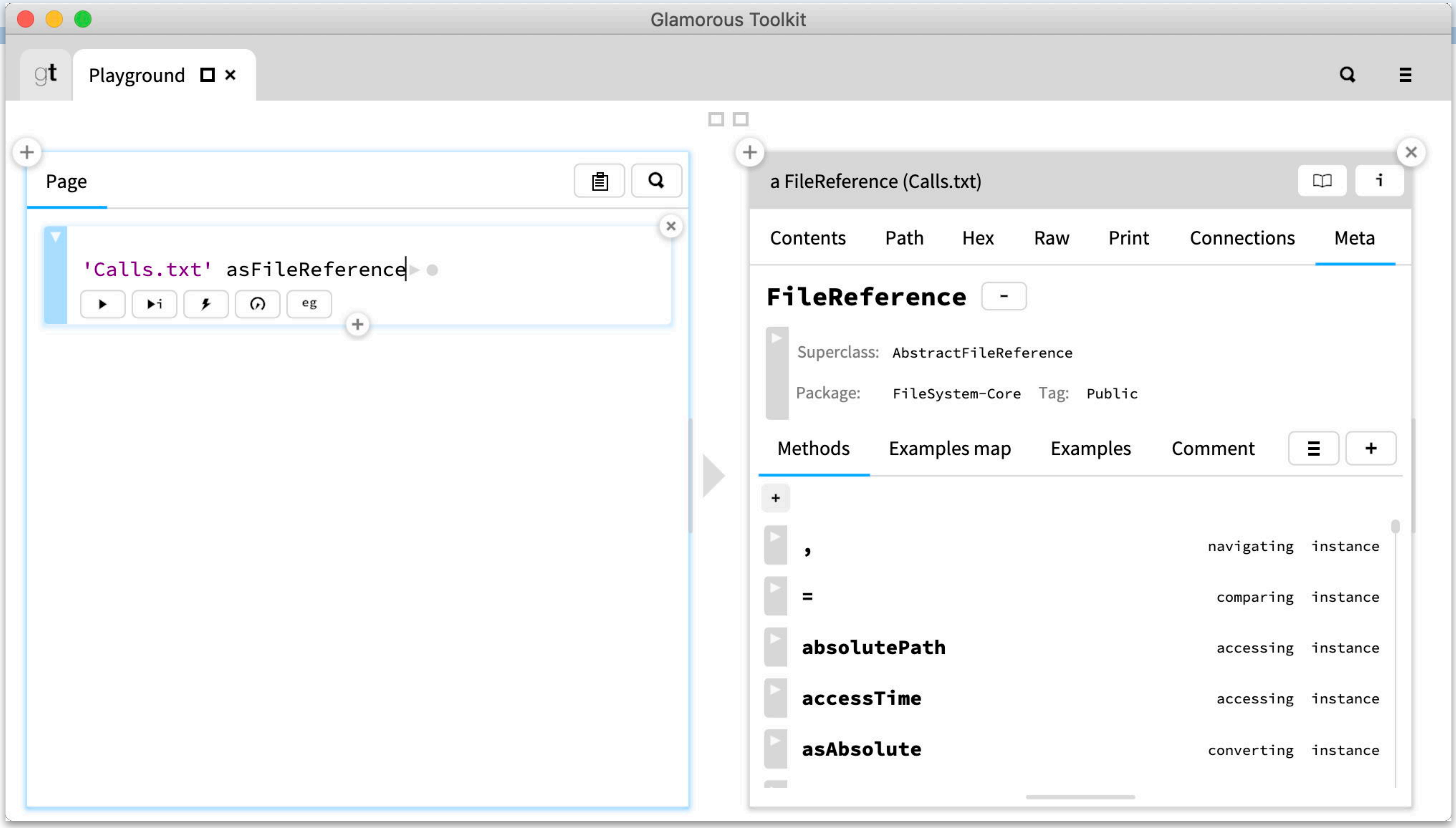
NB: first we must copy the file “Calls.txt” to the folder holding the image.

Navigating to “impleMentors” or “seNders”



You can explore a method’s implementation in place. You can also navigate to iMplementors or seNders by selecting the name and typing <CMD>-M, respectively <CMD>-N.

Navigating to classes



You can browse the class of an object in its Meta tab

There are many ways to navigate to the class of an object.

From the inspector view of an object, you can browse its class in the “Meta” tab. From there you can click on the “book” icon to open a dedicated code browser.

You can also programmatically obtain the class of any object by sending it the message class:

```
('Calls.txt' asFileReference) class
```

There is also a general-purpose search tool called Spotter, which can search for classes, and just about anything else, which we will see later.

Roadmap

> Smalltalk Basics

> **Demo: modeling Call Graphs**

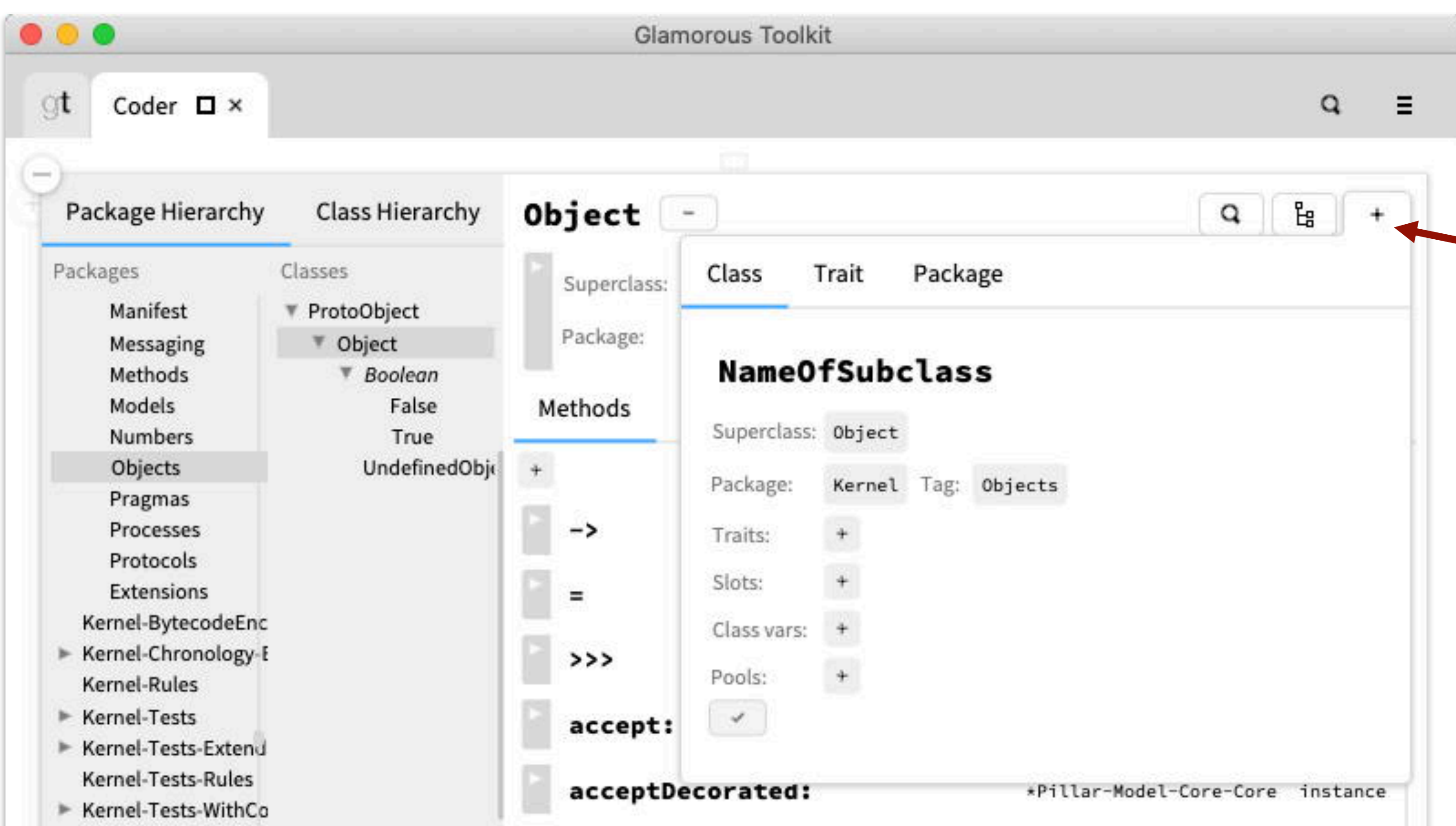
- The call graph model
- Pharo and Glamorous Toolkit
- **Implementing the CallGraph class**
- Version control in Pharo
- Modeling Calls, Methods and Classes
- The Debugger is your Friend!
- Expressing queries

Creating a new class

NB: A symbol

```
Object subclass: #CallGraph
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'CallGraph'
```

To create a new class, send a message to its superclass in the system browser



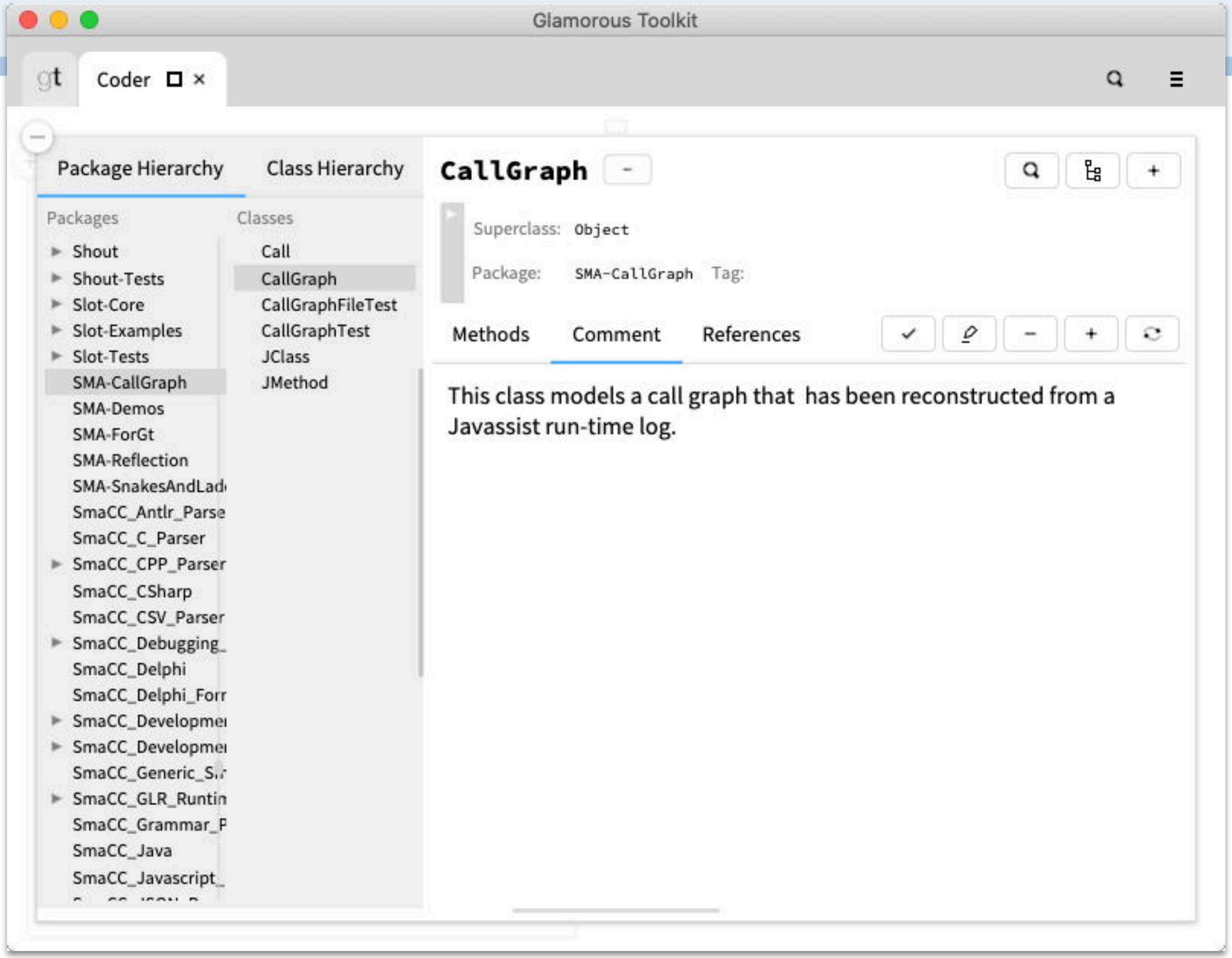
Or create the class from the Coder GUI.

Since *everything happens by sending messages*, it follows that this is also true for creating a class. To create (or update) a class, you simply send a message to its (already existing) superclass.

Note that since the new subclass may not exist yet, you must refer to it using a symbol (i.e., `#ClassGraph`, not `CallGraph`).

In GT, you can also create new classes interactively, using the playground or the class coder.

Class comments



NB: Be sure to write a *class comment*!

In general the idea in Smalltalk is to write literate code that does not require additional comments. Nevertheless, it is very important to *write a class comment for every class you introduce*, and to keep the comment up-to-date.

The class comment is a good place to put some *code snippets* to illustrate how to use the class, or to give pointers to class-side methods to run examples.

Defining methods

Convention to indicate class name

“Selector” (method name)

argument

```
CallGraph>>from: aString
  calls := Character cr split: aString
```

method body

```
CallGraph>>calls
  ^ calls
```

An accessor method

Note that in the slides we usually prefix method names with the class name (`CallGraph>>from: aString`) to make it clear which class it belongs to. This is only a convention for slides, books and papers. It is not needed in the browser because there you can always see what class a method belongs to.

How many calls are there in the call graph?

```
| cg |
cg := CallGraph new from: 'Calls.txt' asFileReference contents.
cg calls size 2476
```

Let’s improve the instantiation interface

Factory methods and other “static” methods are defined on the *class side*

```
CallGraph class>>fromFile: fileName
  ^ self new from: fileName asFileReference contents
```

```
(CallGraph fromFile: 'Calls.txt') calls size. 2476
```

Let’s turn this into a test!

Now we must define a class-side method. `#fromFile:` is a message understood by the `CallGraph` class (as opposed it its instance). We click on the “Class” button to switch to the class-side methods.

Note that the method `Callgraph class>>#fromFile:` must return an instance of `CallGraph`. Instead of evaluating `CallGraph new`, we evaluate `self new` (`self` is anyway this class, but we would also like the code to work for eventual subclasses!).

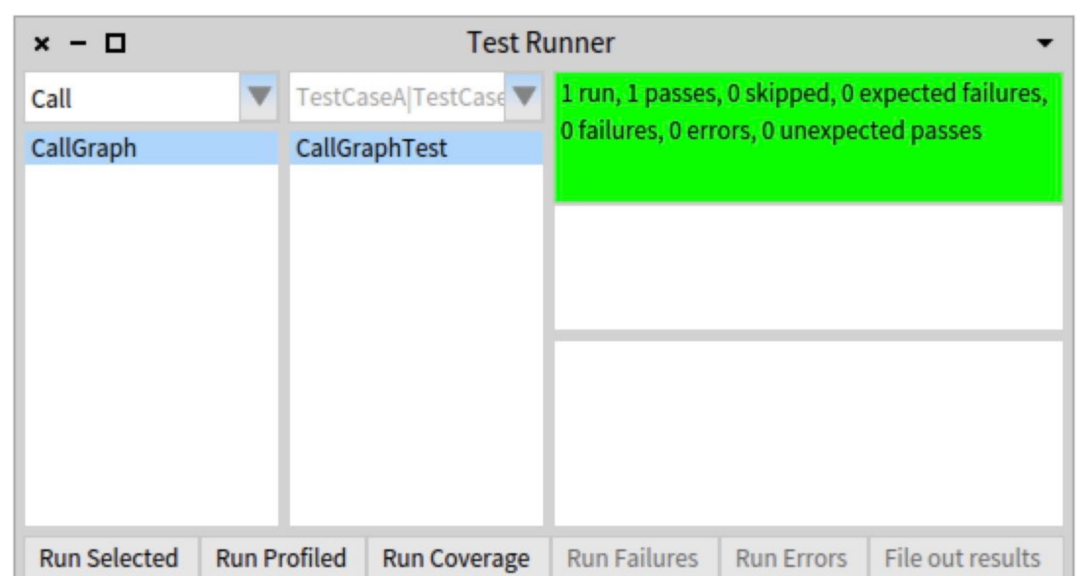
Creating a simple test (in Pharo)

a 5-line excerpt from Calls.txt

```
CallGraph class>>example
  ^ self new from: '|java.lang.String:…'
```

```
TestCase subclass: #CallGraphTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'CallGraph'
```

```
CallGraphTest>>testNumberOfCalls
  self assert: CallGraph example calls size equals: 5
```

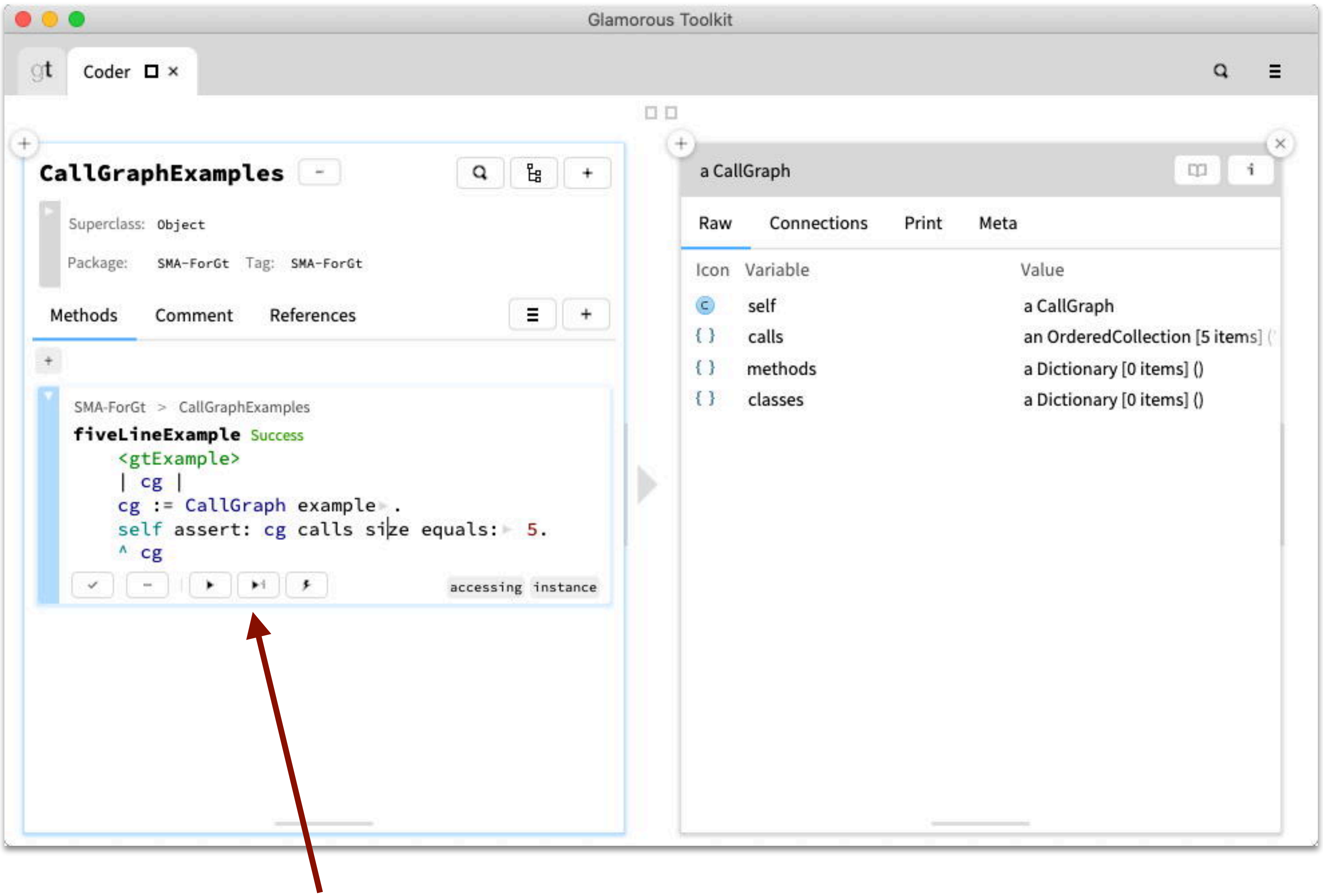


X

Test classes inherit from `TestCase` and are usually named after the class they test + "`Test`".

You can run tests from the TestRunner tool, or directly from the System Browser (by clicking the button next to a test method or a test class).

Test examples in GT



Tests in GT consist of methods containing *assertions* and returning an *example* object. Example objects can be *composed*.

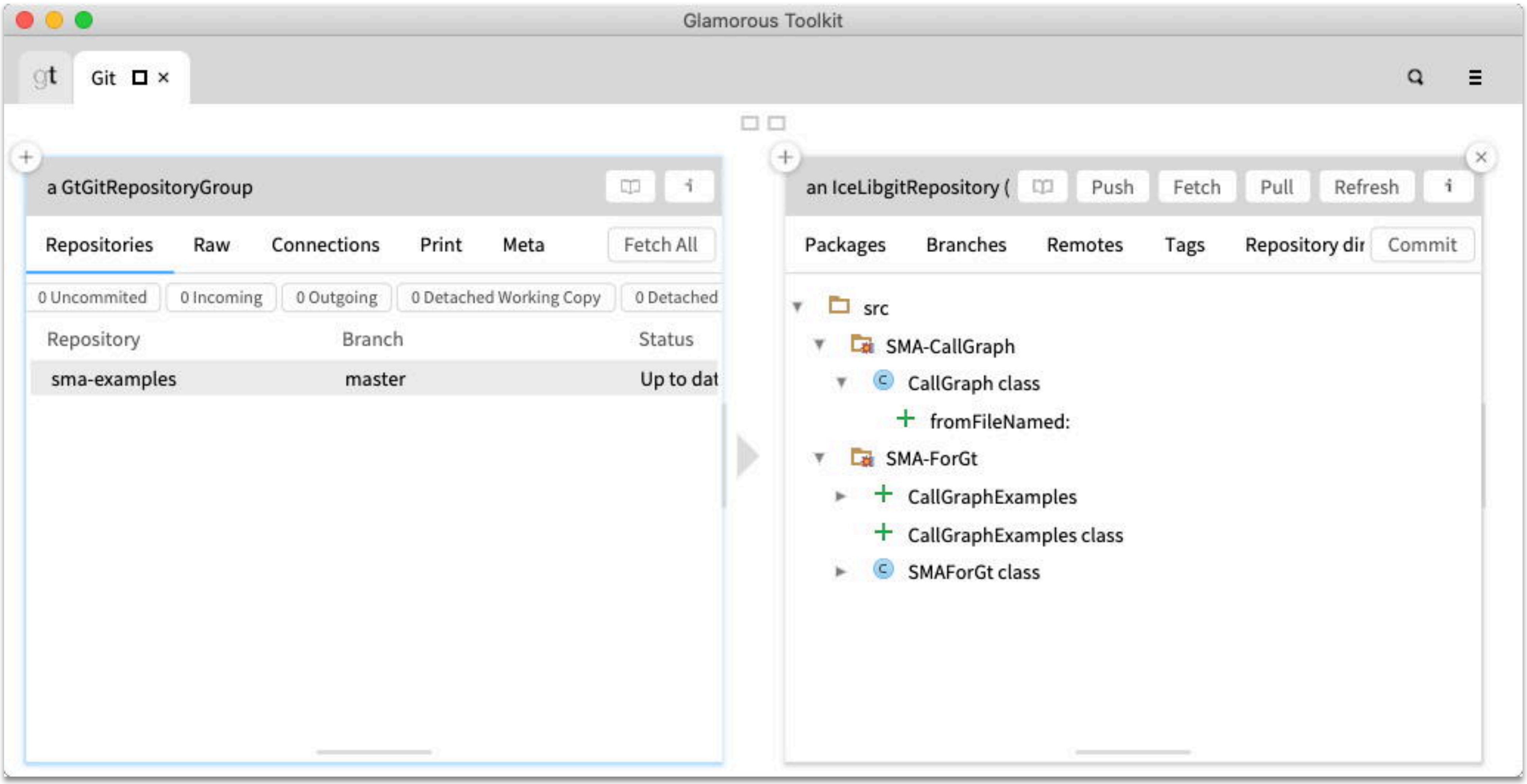
The `<gtExample>` pragma allows example methods to be run from the browser.

Instead of writing tests, we write examples, which we can inspect, interact with, and compose to form scenarios, or more complex objects.

Roadmap

- > Smalltalk Basics
- > **Demo: modeling Call Graphs**
  - — The call graph model
  - — Pharo and Glamorous Toolkit
  - — Implementing the CallGraph class
  - — **Version control in Pharo**
  - — Modeling Calls, Methods and Classes
  - — The Debugger is your Friend!
  - — Expressing queries

Version control for Pharo and GT



Pharo and GT offer version control via git.

Git integration is provided by a library and tool called Iceberg. To use it, you should adopt the convention that all source files are saved in a subfolder called “src”.

You should also define a BaseLineOf… package containing a script to simplify the loading of your packages.

See the SMA demo repo as an example.

https://github.com/onierstrasz/sma-examples

Roadmap

> Smalltalk Basics
> **Demo: modeling Call Graphs**
  — The call graph model
  — Pharo and Glamorous Toolkit
  — Implementing the CallGraph class
  — Version control in Pharo
  — **Modeling Calls, Methods and Classes**
  — The Debugger is your Friend!
  — Expressing queries

Modeling Calls, Methods and Classes

We want to build up a Call object for each line of the log

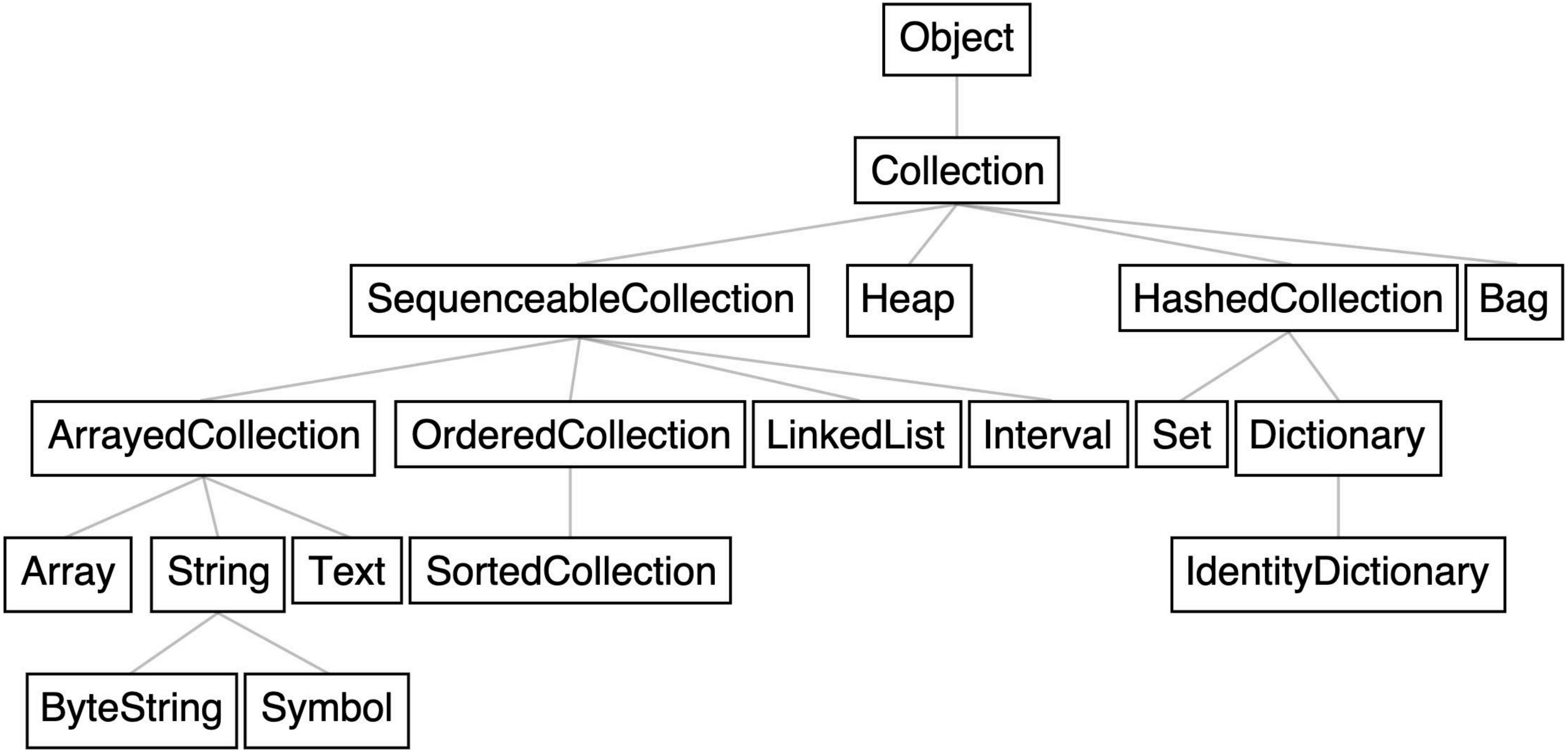
```
CallGraph>>from: aString
  calls := (Character cr split: aString)
      collect: [ :each | self createCall: each ]
```

```
'hello' collect: [ :each | each uppercase ]  'HELLO'
```

Let's look at Collections first …

In order to build up the model, we need to create a `Call` object from each line of the log file. To do this, we will map the `#createCall` method to each line using the `#OrderedCollection>>collect:` method.

Collections



Resist the temptation to program your own collections!

The Smalltalk collection hierarchy offers a mature library of classes to manage various kinds of collections.

Hint: if you need to manage some kind of ordered list, you should normally use the `OrderedCollection` class (i.e., rather than `Array` or `LinkedList`).

NB: The diagram is an interactive visualization generated from the actual class hierarchy using Mondrian:

```
GtMondrianDomainExamples new collectionHierarchy
```

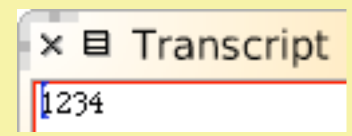
The collection hierarchy is described in detail in chapter 9 of *Pharo by Example*:

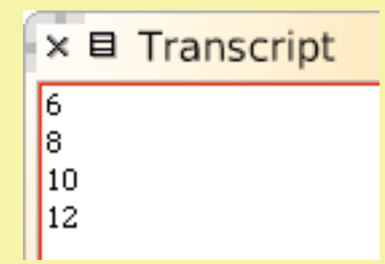
http://files.pharo.org/books/pharo-by-example/

Common messages

```
#(1 2 3 4) includes: 5
#(1 2 3 4) size
#(1 2 3 4) isEmpty
#(1 2 3 4) contains: [:some | some < 0 ]
#(1 2 3 4) do:
   [:each | Transcript show: each ]
#(1 2 3 4) with: #(5 6 7 8)
   do: [:x : y | Transcript show: x+y; cr]
#(1 2 3 4) select: [:each | each odd ]
#(1 2 3 4) reject: [:each | each odd ]
#(1 2 3 4) detect: [:each | each odd ]
#(1 2 3 4) collect: [:each | each even ]
#(1 2 3 4) inject: 0
   into: [:sum :each | sum + each]
```

```
false
4
false
false
```





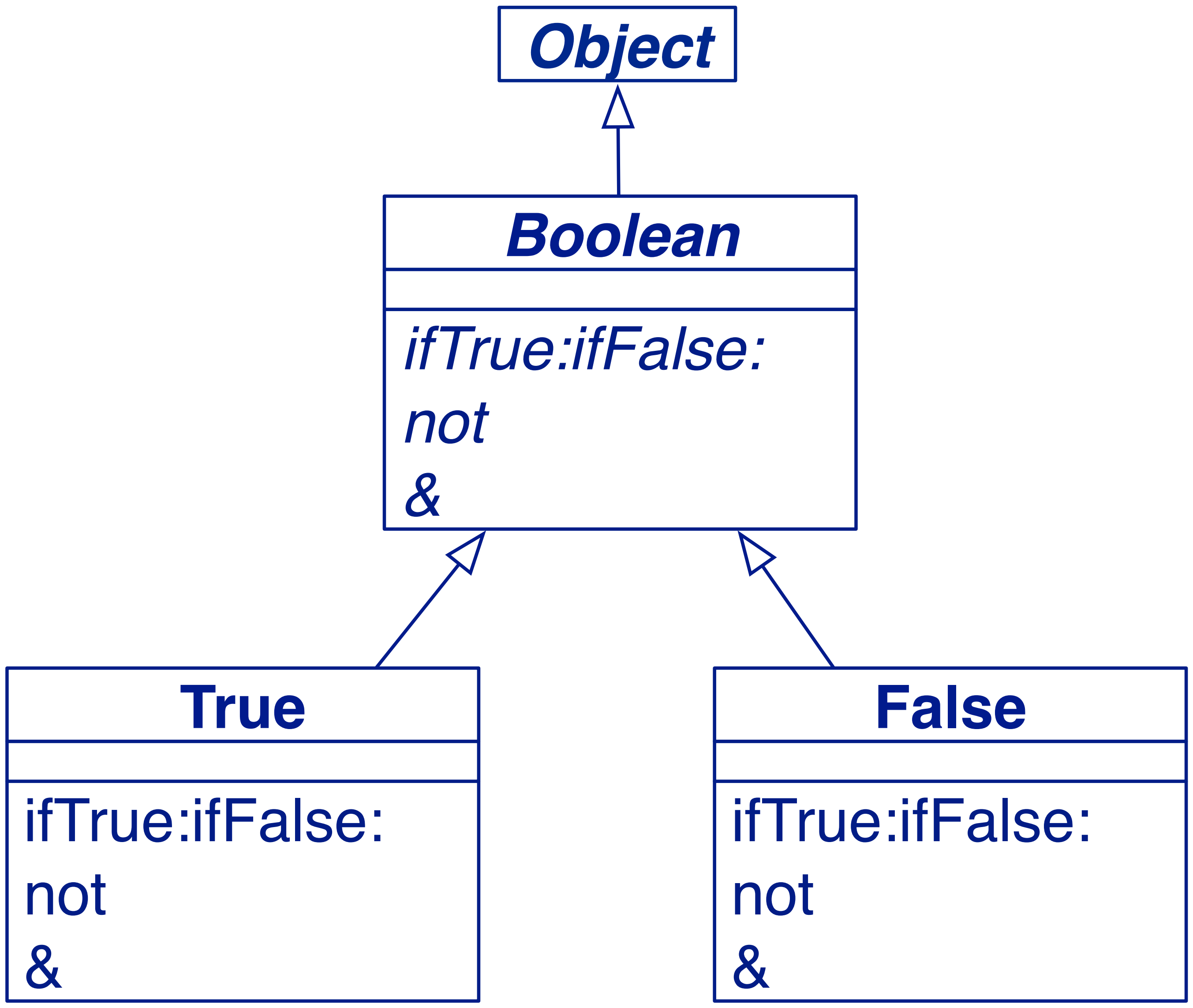

```
#(1 3)
#(2 4)
1
{false.true.false.true}

10
```

Most of these methods should be obvious:

- `#select:` and `#reject` return subcollections matching the block (or not)
- `#detect:` returns the first matching element or raises an error
- `#collect:` is more commonly knows as "*map*" — it returns a new collection of the same size by mapping the argument block to each element
  https://en.wikipedia.org/wiki/Map_(higher-order_function)
- `#inject:into:` is also known as "*fold*" — it takes an initial value and iteratuvely applies the two-argument block to that value and each element in the collection, producing, for example, a sum or a product
  https://en.wikipedia.org/wiki/Fold_(higher-order_function)

Conditionals

```
(11 factorial + 1) isPrime ifTrue: [ 'yes' ] ifFalse: [  'no' ]
      'yes'
```



All control constructs in Smalltalk are implemented by message passing

- No keywords
- Open, extensible
- Built up from Booleans and Blocks

Since *everything is an object* in Smalltalk, it should not come as a surprise that Booleans are objects too. You might ask, *"Well, how do you implement Booleans if you don't have them as primitives?"*

Actually the implementation closely follows the standard encoding in the lambda calculus. A Boolean is simply an object that can make a choice between two alternatives: true and false just make opposite choices.

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

The objects `true` and `false` are (unique) instances of the classes `True` and `False`. Each implements methods like `#ifTrue:ifFalse:` in its own way.

Have a look at the implementation of these methods in the system.

Creating Calls, Methods and Classes

```
CallGraph>>createCall: callString
  | fields callee |
  fields := $| split: callString.
  self assert: fields size = 5.
  self assert: (fields at: 1) size = 0.
  callee := self getMethod: (fields at: 2).
  ^ Call new callee: callee
  "TODO -- handle the remaining fields!"
```

temporary (local) variables

assertions (not tests)

a comment

```
CallGraph>>initialize
  super initialize.
  methods := Dictionary new
```

cache the methods!

```
CallGraph>>getMethod: signature
  | fields methodName |
  fields := $: split: signature.
  methodName := fields at: 2.
  ^ methods at: signature
    ifAbsentPut: [ JMethod new name: methodName ]
```

```
CallGraph>>methods
  ^ methods
```

To create the call graph, we must split each line of the log into its individual fields by the `$|` character.

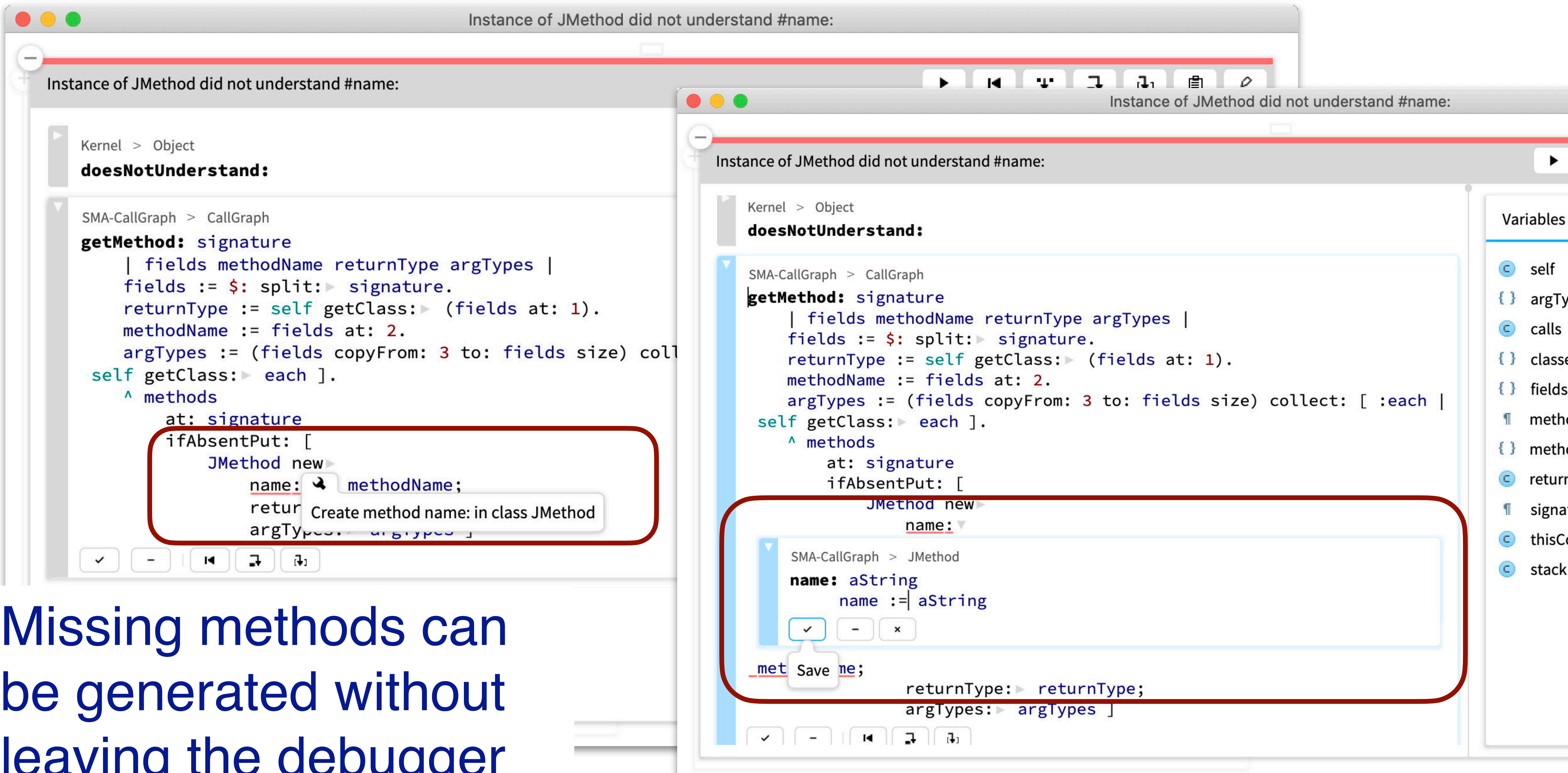
Each `Call` object stores a reference to its callee, a `JMethod` object representing the called Java method. Since each method may be called multiple times, but we only want to have a unique `JMethod` instance representing that method, we cache these objects in a dictionary indexed by the method signature (field 2 of the log).

Roadmap

- > Smalltalk Basics
- > **Demo: modeling Call Graphs**
  - The call graph model
  - Pharo and Glamorous Toolkit
  - Implementing the CallGraph class
  - Version control in Pharo
  - Modeling Calls, Methods and Classes
  - **The Debugger is your Friend!**
  - Expressing queries

The debugger is your friend!

```
(CallGraph fromFile: 'Calls.txt') methods size.
```



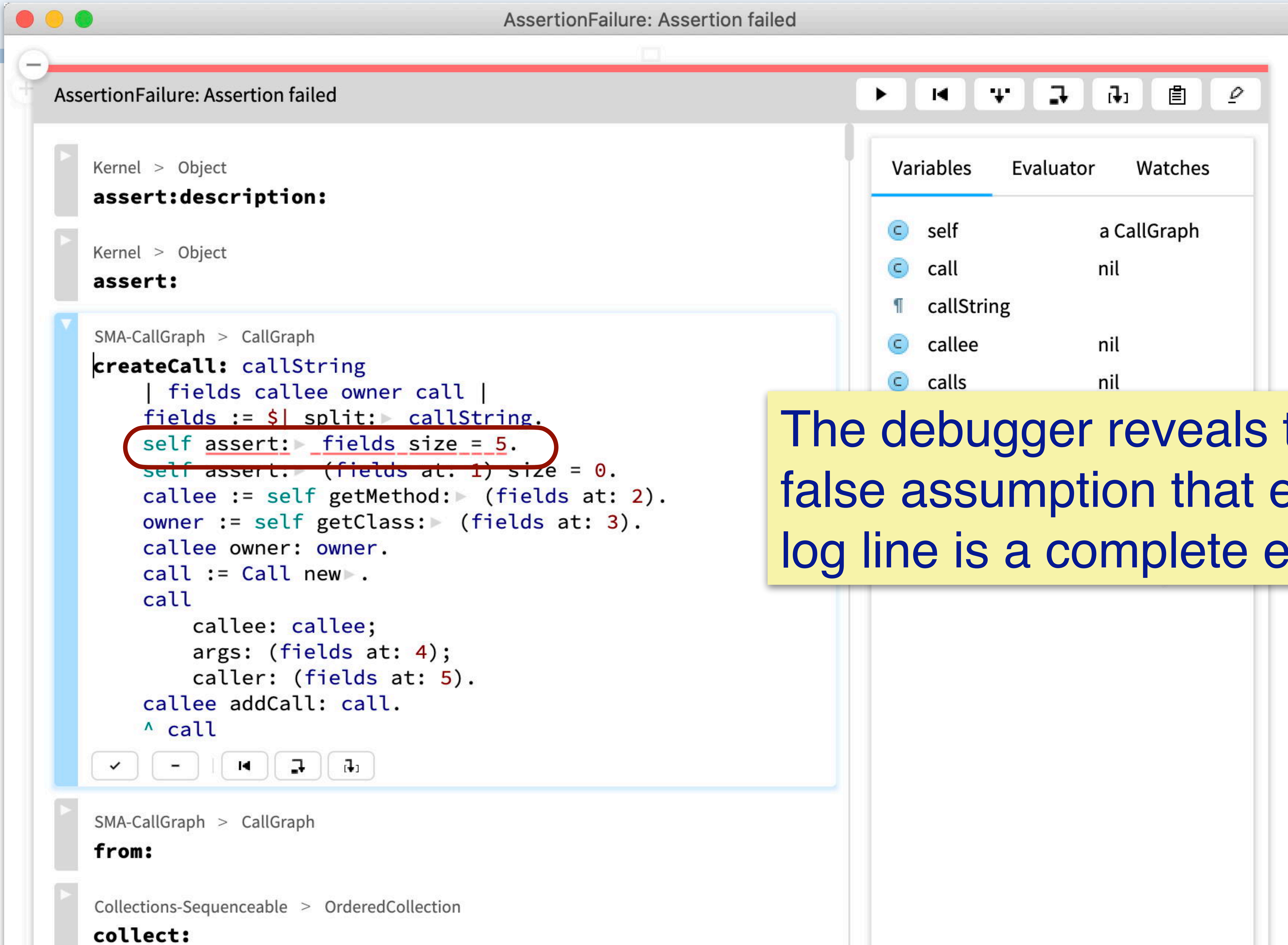
Missing methods can be generated without leaving the debugger

When we evaluate this snippet, it turns out that we have forgotten to implement some methods. (In this case `#JMethod>>name:`) The Debugger window pops up and offers us the possibility to create the missing method.

Aside: this offers you an effective way to follow TDD (test-driven development) in Pharo — implement some tests, then run them, and use the Debugger to prompt you to implement the missing classes and methods.

From the debugger we can generate both `JMethod>>name:` and `Call>>callee:` and proceed with execution!

Using the debugger



The debugger reveals the false assumption that each log line is a complete entry

The standard Pharo debugger shows you the run-time stack of currently executing methods. Here we see that an assertion failed in the `#createCall:` method. The inspector window below shows that the given fields collection is unexpectedly empty.

Roadmap

- Smalltalk Basics
- **Demo: modeling Call Graphs**
  - The call graph model
  - Pharo and Glamorous Toolkit
  - Implementing the CallGraph class
  - Version control in Pharo
  - Modeling Calls, Methods and Classes
  - The Debugger is your Friend!
  - **Expressing queries**

Duck Typing in Smalltalk

```
CallGraph>>from: aString
        calls := ((Character cr split: aString)
                select: #notEmpty)
                collect: [ :each | self createCall: each ]
```

Behaves like:

```
CallGraph>>from: aString
        calls := ((Character cr split: aString)
                select: [:each | each notEmpty])
                collect: [ :each | self createCall: each ]
```

since symbols also understand `value:`

“Duck typing” refers to one object masquerading as another by implementing its interface. (“If it quacks like a duck, it must be a duck”.)

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

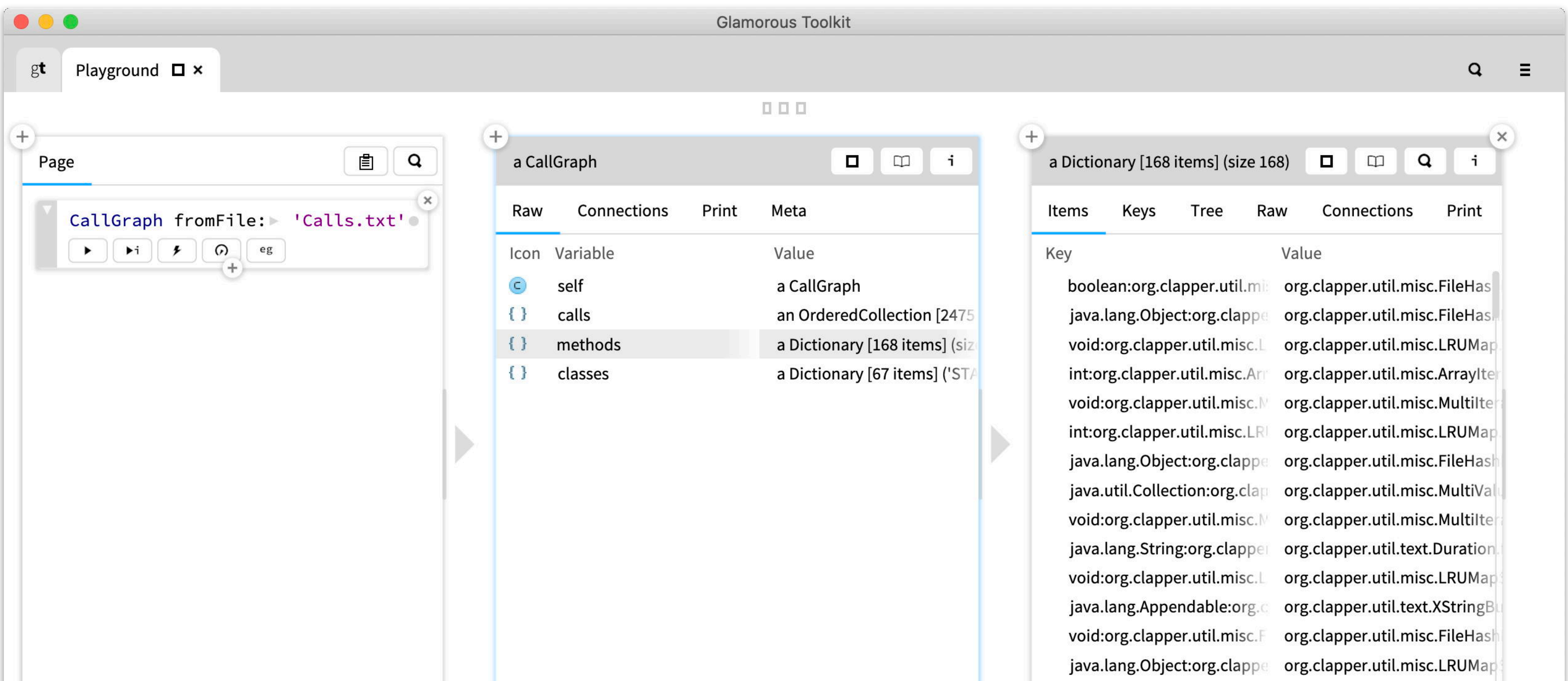
Here we are using a symbol (`#notEmpty`) where we would normally expect a one-argument block. This works simply because the `Symbol` class implements the `#value:` method used to evaluate a block.

Duck typing is unique to dynamically-typed languages like Smalltalk and Ruby. In a statically-typed language like Java you would achieve the same effect by defining an *interface* for objects that can be evaluated with an argument (e.g., `IOneArgumentBlock`) and ensuring that the relevant classes (`Block`, `Symbol`) implement that interface.

Number of methods

```
CallGraphTest>>testNumberOfMethods
  self assert: CallGraph example methods size equals: 5
```

```
(CallGraph fromFile: 'Calls.txt') methods size. 168
```



To do ...

- > Model classes (introduce `JClass` class)
- > Model argument and return types of methods
- > Track which methods are static
- > Determine which methods are polymorphic

To continue from here we introduce a class `JClass` to represent all the Java classes we encounter as owners of methods, or as argument and return types. (`'STATIC_METHOD'` is a dummy class to represent static methods.)

We extend `CallGraph>>#createCall:` and `#CallGraph>>#getMethod:` to track classes as well as methods. `CallGraph>>#getClass:` caches the `JClass` instances with a dictionary, just as we did with `#getMethod`.

We can recognize static methods by checking if their owner is static. A *polymorphic* method is one that takes arguments of different types, so we look at the set of arguments from the calls and check if that set is greater than 1.

Queries

```
(CallGraph fromFile: 'Calls.txt') methods size. 168
```

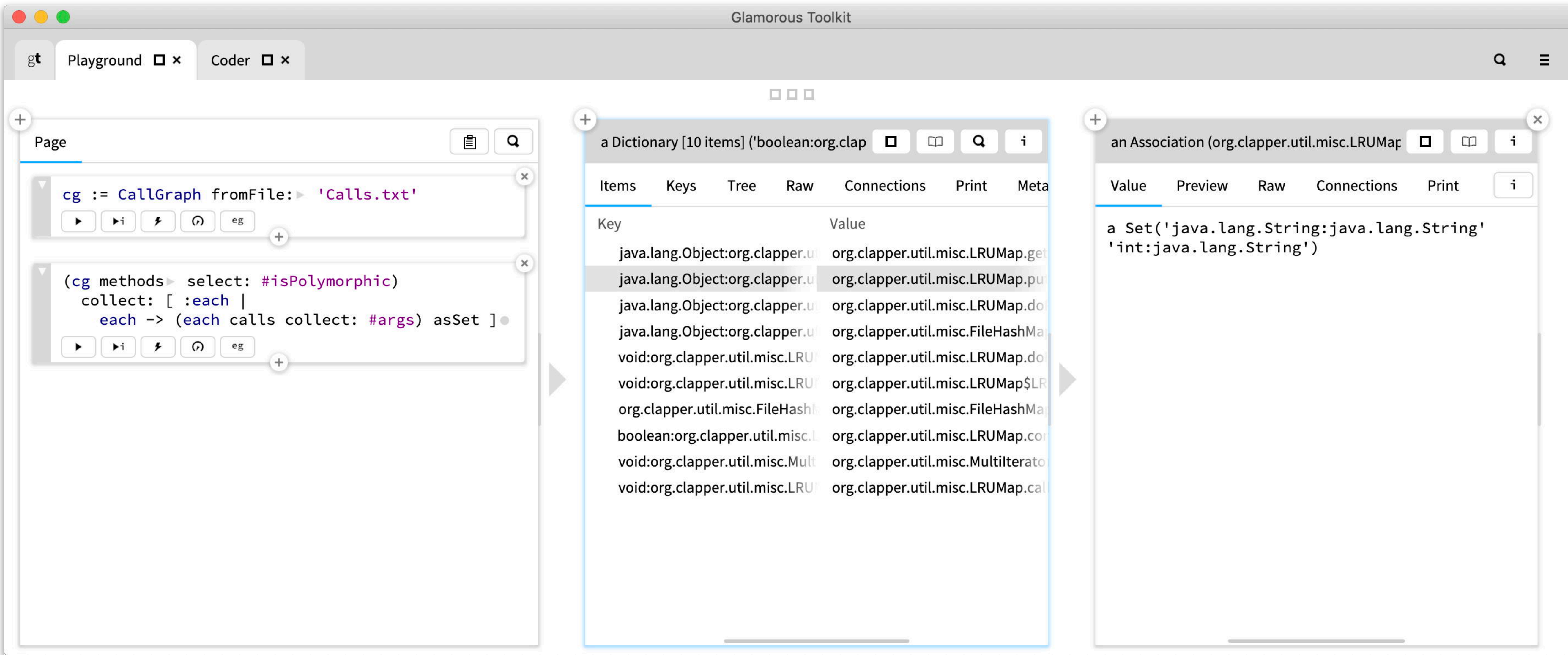
```
(CallGraph fromFile: 'Calls.txt') classes size. 67
```

```
((CallGraph fromFile: 'Calls.txt') methods
      select: [ :m | m calls size > 1 ]) size. 141
```

```
((CallGraph fromFile: 'Calls.txt') methods
      select: #isPolymorphic) size. 10
```

Navigating the CallGraph

The Playground offers a convenient interface to navigate through our CallGraph hierarchy.



What you should know!

- What’s the difference between a *method*, a *selector* and a *message*?
- What are *categories* and *protocols*? What are they for?
- How do you create a new class in Smalltalk?
- What’s the difference between `CallGraph` and `CallGraph class`?
- What are “class side” methods for?
- How is a block like a lambda?
- What’s the difference between a string and a symbol?

Can you answer these questions?

- Can a class access the fields of one of its instances?
- Can you name something that is not an object in Smalltalk?
- What happens to existing instances of a class if you add new fields at run time?
- What will happen if you change the implementation of core classes (like Booleans or Strings)?
- What's the difference between `self` and `super`?