11. A bit of Smalltalk

Oscar Nierstrasz

b
UNIVERSITÄT
BERN

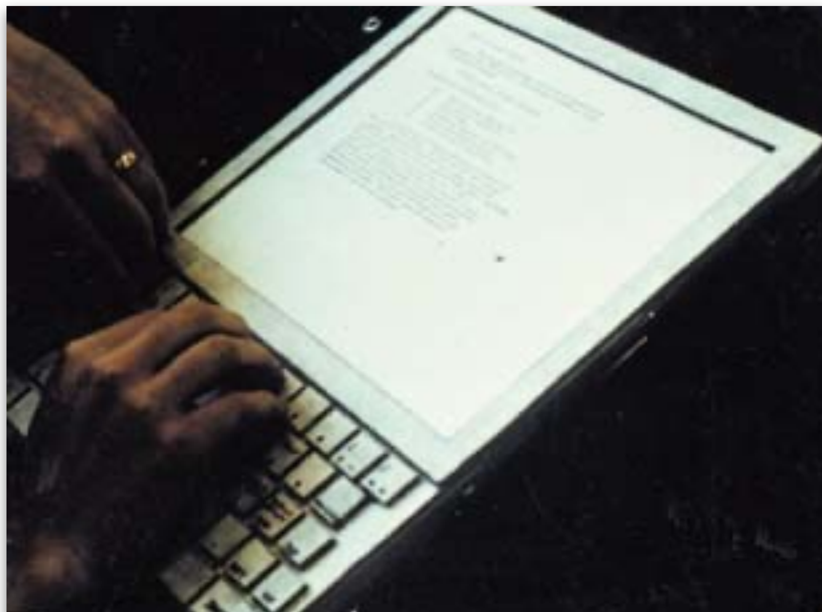
Roadmap

- The origins of Smalltalk
- Syntax in a nutshell
- Pharo and Gt
- Demo — the basics
- Demo — live programming with Gt

Roadmap

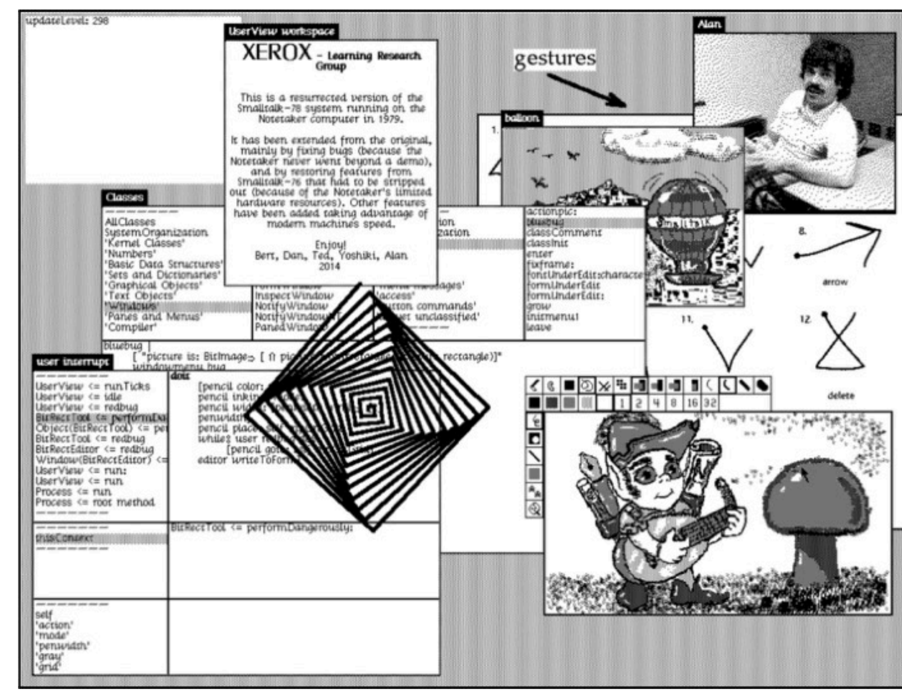
> **The origins of Smalltalk**
> Syntax in a nutshell
> Pharo and Gt
> Demo — the basics
> Demo — live programming with Gt

The origins of Smalltalk



Alan Kay’s Dynabook project (1968)

Alto — Xerox PARC (1973)

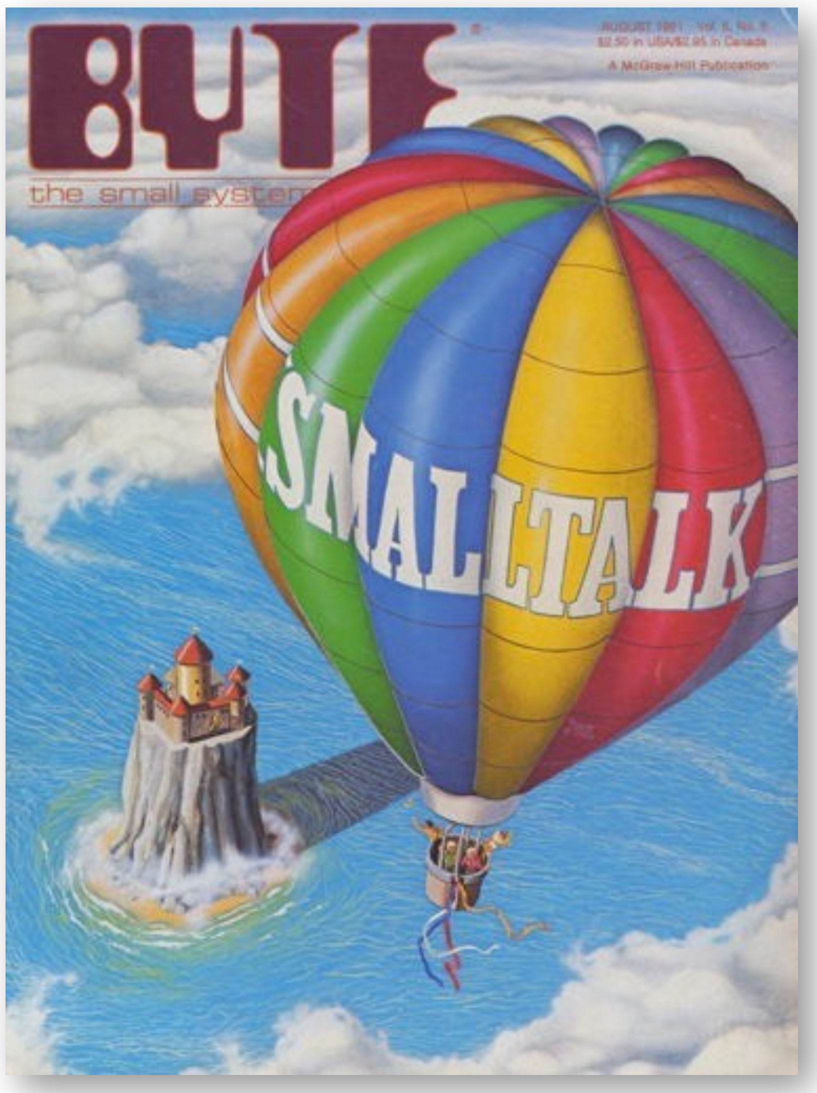


Smalltalk-78 windows

In the late 60s, Alan Kay predicted that in the foreseeable future handheld multimedia computers would become affordable. He called this a “Dynabook”. (The photo shows a mockup, not a real computer.)

He reasoned that such systems would need to be based on object from the ground up, so he set up a lab at the Xerox Palo Alto Research Center (PARC) to develop such a fully object-oriented system, including both software and hardware. They developed the first graphical workstations with windowing system and mouse.

Smalltalk-80

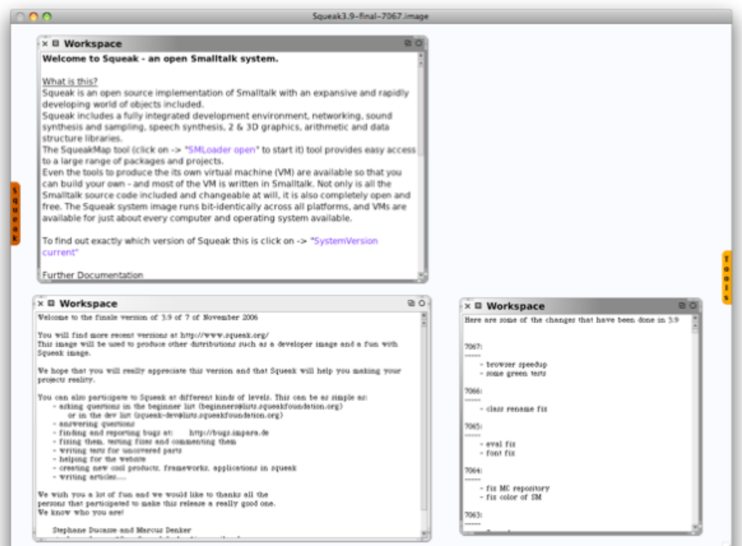


Everything is an object.
Everything is there, all the time.
First windowing system with mouse.
First graphical IDE.

Smalltalk-80 was introduced to the world in 1981 in a now-famous issue of Byte Magazine. The “Smalltalk balloon” refers to this issue.

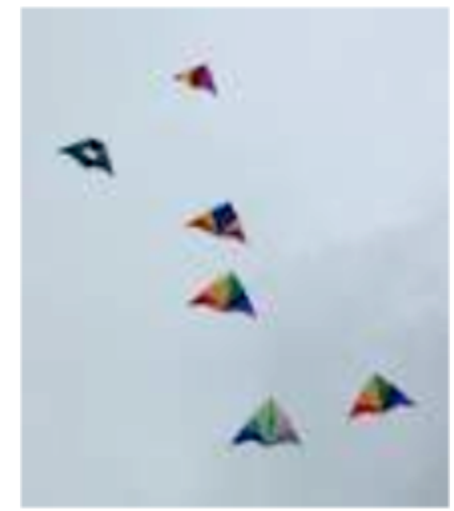
https://archive.org/details/byte-magazine-1981-08

Smalltalk — a *live* programming environment

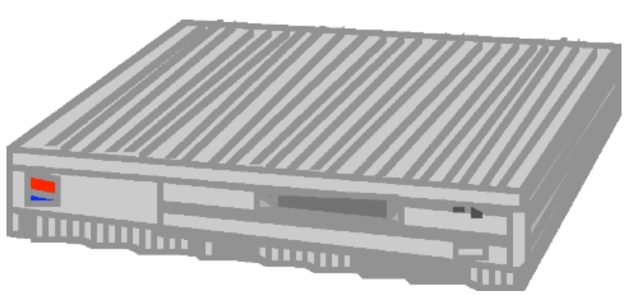


Image

+



Changes



Virtual machine

+

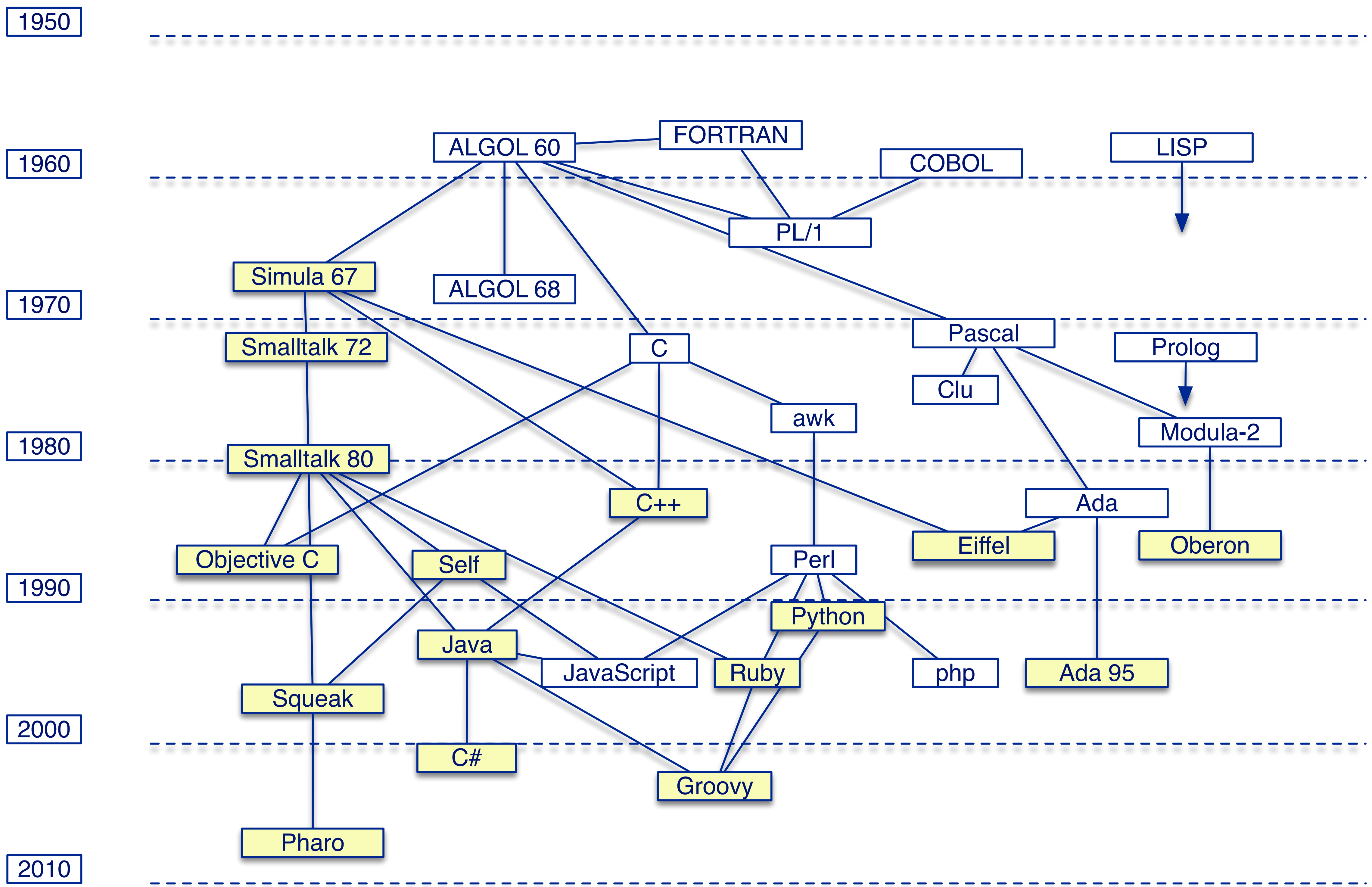
Sources

Smalltalk is often bundled into a single, “one-click” application, but there are actually four pieces that are important to understand.

Every user of Smalltalk can work with one or more Smalltalk images. The *image* file contains a snapshot of all the objects of the running system. Every time you quit Smalltalk, you can save and update this snapshot. In addition, the *changes* file consists of a log of all changes to the source code of that image, i.e., all new or changed classes and all compiled methods. If your image crashes (which is possible since Smalltalk allows you to do anything, even if that might be fatal), you can restart your image and *replay your changes*, so nothing is lost.

In addition, the virtual machine and sources files may be shared between users. The *VM* runs the bytecode of compiled methods and manages the image and changes file. Finally the *sources* file (optional) contains all the source code of objects in the base image (so you can not only explore this but modify it if you want).

Object-oriented language genealogy



Simula was the first object-oriented language, designed by Kristen Nygaard and Ole Johan Dahl. Simula was designed in the early 60s, to support simulation programming, by adding classes and inheritance to Algol 60. The language was later standardized as Simula 67. Programmers quickly discovered that these mechanisms were useful for general-purpose programming, not just simulations.

Smalltalk adopted the ideas of objects and message-passing as the core mechanisms, not just add-ons to a procedural language.

Stroustrup ported the ideas of Simula to C to support simulation programming. The resulting language was first called "C with classes", and later C++.

Cox added Smalltalk-style message-passing syntax to C and called it "Objective-C".

Java integrated implementation technology from Smalltalk and syntax from C++.

Squeak and Pharo are modern descendants of Smalltalk-80.

Smalltalk vs. Java vs. C++

	Smalltalk	***Java***	***C++***
Object model	Pure	Hybrid	Hybrid
Garbage collection	Automatic	Automatic	Manual
Inheritance	Single	Single	Multiple
Types	Dynamic	Static	Static
Reflection	Fully reflective	Introspection	Introspection
Modules	Categories, namespaces	Packages	Namespaces

The most important difference between Smalltalk, Java and C++, is that Smalltalk supports “live programming”. Whereas in Java and C++ you must first write source code and compile it before you run anything, in Smalltalk you are always programming in a live environment. You incrementally add classes and compile methods within a running system.

As a consequence, Smalltalk has to be fully reflective, allowing you to reify (“turn in objects”) all aspects of the system, and change them at run time. The only thing you cannot change from within Smalltalk is the virtual machine.

Roadmap

> The origins of Smalltalk
> **Syntax in a nutshell**
> Pharo and Gt
> Demo — the basics
> Demo — live programming with Gt

Literals and constants

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

Everything is an object in Smalltalk, including these literal and constant values.

Strings are just special kinds of ordered collections holding character values.

Smalltalk supports various kind of numbers, and also supports radix notation for numbers in different bases.

Symbols behave much like strings, but are guaranteed to be globally unique. They always start with a hash (#).

In addition to `self`, `super`, `true` and `false`, there are only two further reserved names in Smalltalk: `nil` and `thisContext`. (The latter is only needed for meta-programming!)

Three kinds of messages

> *Unary messages*

```
5 factorial
Transcript cr
```

> *Binary messages*

```
3 + 4
```

> *Keyword messages*

```
3 raisedTo: 10 modulo: 5

Transcript show: 'hello world'
```

Smalltalk has a very simple syntax. There are just three kinds of messages:

1. *Unary messages* consist of a single world sent to an object (the result of an expression). Here we send `factorial` to the object `5` and `cr` (carriage return) to the object `Transcript`. (Aside: upper-case variables are global in Smalltalk, usually class names. Transcript is one of the few globals that is not a class.)

2. *Binary messages* are operators composed of the characters `+`, `-`, `*`, `/`, `&`, `=`, `>`, `|`, `<`, `~`, and `@`.
Here we send the message "`+ 4`" to the object `3`.

3. *Keyword messages* take multiple arguments. Here we send "`raisedTo: 10 modulo: 5`" to 3 and "`show: 'hello world'`" to `Transcript`.

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

The precedence rules for Smalltalk are exceedingly simple: unary messages are sent first, then binary, and finally keyword messages. Use parentheses to force a different order.

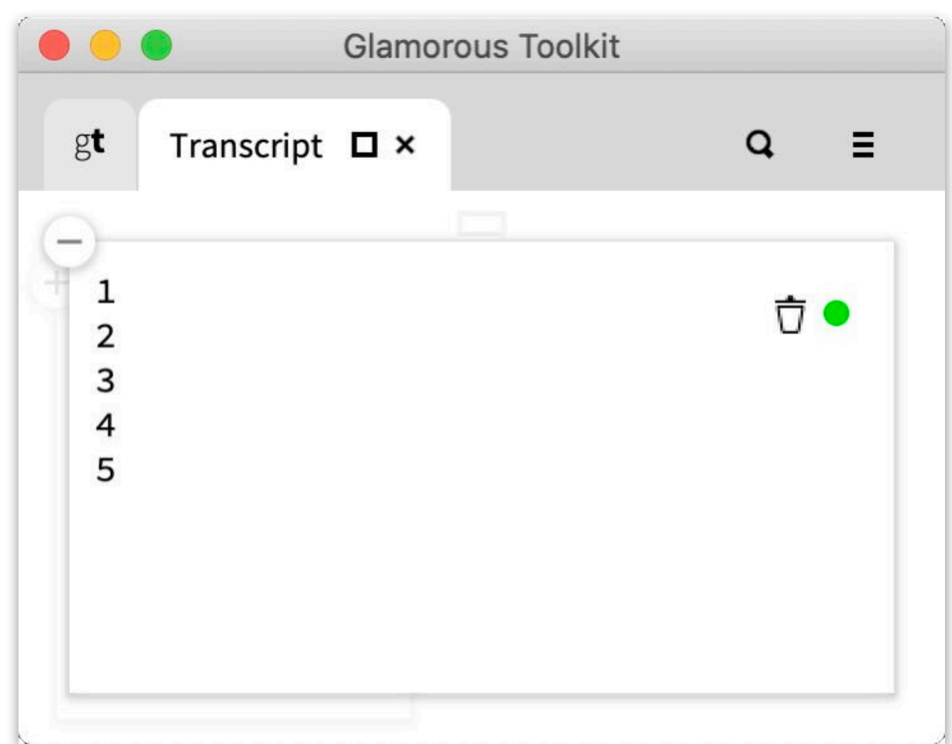
Note that there is no difference in precedence between binary operators.

Blocks

Block

```
1 to: 5 do: [:n | Transcript show: n; cr ]
```

Keyword message   Block argument   Cascade



A typical method in the class Point

Method name

Argument

Comment

```
<= aPoint
  "Answer whether the receiver is neither
  below nor to the right of aPoint."

  ^ x <= aPoint x and: [y <= aPoint y]
```

Return

Binary message

Block

Instance variable

Keyword message

```
(2@3) <= (5@6)
```

`true`

X

The slide shows the `<=` method of the `Point` class as it appears in the IDE.

The first line lists the method name and its formal parameters. In this case we are defining the method for the `<=` selector. (In Smalltalk, method names are called "*selectors*", because when a message is received, the selector is used to select the method to respond.)

Comments are enclosed in double quotation marks (*strings* are enclosed in single quotes).

The body of this method consists of a single expression. The caret (`^`) is a reserved symbol in Smalltalk and denotes a *return value*. A *block* is enclosed in square brackets and denotes an expression that may be evaluated. In this case, the Boolean `and:` method will only evaluate the block if its receiver (i.e., the subexpression to the left of the `and:`) evaluates to `true`.

Statements and cascades

Temporary variables

Statement

```
| p pen |
p := 100@100.
pen := Pen new.
pen up.
pen goto: p; down; goto: p+p
```

Assignment

Cascade

This is a code snippet (not a method) that may be evaluated in the Playground.

Here we see that *statements* are expressions separated by periods (.).

Even though Smalltalk does not support type declarations, *local variables must still be declared*, appearing within or-bars (|).

A variable is bound to a value using the *assignment* operator (`:=`).

Smalltalk supports a special syntax, called a *cascade*, to send multiple messages to the same receiver. Messages in a cascade are separated by semi-colons (`;`). In this case we send the messages "`goto: p`", "`down`", and finally "`goto: p+p`" to the receiver `p`. (This draws a line from the `Point` `100@100` to `200@200`.)

Note that `100@100` looks like special syntax for `Point` objects, but it is really just a Factory method of the `Number` class, which creates a new `Point` instance.

Variables

> *Local variables* are delimited by `|var|`
> *Block variables* by `:var|`

```
OrderedCollection>>collect: aBlock
    "Evaluate aBlock with each of my elements as the argument."
    | newCollection |
    newCollection := self species new: self size.
    firstIndex to: lastIndex do:
        [ :index |
        newCollection addLast: (aBlock value: (array at: index))].
    ^ newCollection
```

```
(OrderedCollection with: 10 with: 5) collect: [:each| each factorial ]
```

```
an OrderedCollection(3628800 120)
```

NB: Since source code for methods in the IDE does not show the class of the method, it is a common convention in documentation to add the missing class name, followed by two greater-than signs (>>), as in this example.

This example serves mainly to show that blocks can take arguments. The arguments are after the opening left square bracket, and each is preceded by a colon (`:`).

The block:

```
[:each| each factorial ]
```

takes its arguments from the receiver of `collect:`, the collection holding `10` and `5`.

Control Structures

> *Every control structure is realized by message sends*

```
max: aNumber
    ^ self < aNumber
        ifTrue: [aNumber]
        ifFalse: [self]
```

```
4 timesRepeat: [Beeper beep]
```

There are no built-in control constructs in Smalltalk. *Everything happens by sending messages!*

Even a simple if statement is achieved by sending a message to a boolean expression, which will then evaluate the block argument only if it boolean is true.

Here we see that the `max:` method is implemented by sending `ifTrue:ifFalse:` to the Boolean expression `self<aNumber`. The `ifTrue:ifFalse:` method is itself defined in the Boolean classes `True` and `False`.

(Try to imagine how it would be implemented, and then check in the image to see how it is done.)

Creating objects

> *Class methods*

```
OrderedCollection new
Array with: 1 with: 2
```

> *Factory methods*

```
1@2
1/2
```

a Point
a Fraction

Ultimately all objects (aside from literals) are created by sending the message `new` to a class. (The message `new:` is used to create arrays of a given length.) Further constructors may be defined as convenience methods on classes, for example,

```
Array with: 1 with: 2
```

will create an `Array` of length 2 using `new:`, and then initialize it with the two arguments.

Other instance creation methods may be defined on the classes of arguments used to create the objects. For example, to create a `Fraction`, we send the message `/` to an `Integer`, with the numerator as its argument. This method will then actually create a new `Fraction` for us.

Creating classes

> Send a message to a class (!)

```
Number subclass: #Complex
    instanceVariableNames: 'real imaginary'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ComplexNumbers'
```

Everything is an object, ergo *classes are objects too*!

To create a new class, you must send a message to an existing class, asking it to create (or redefine) a subclass.

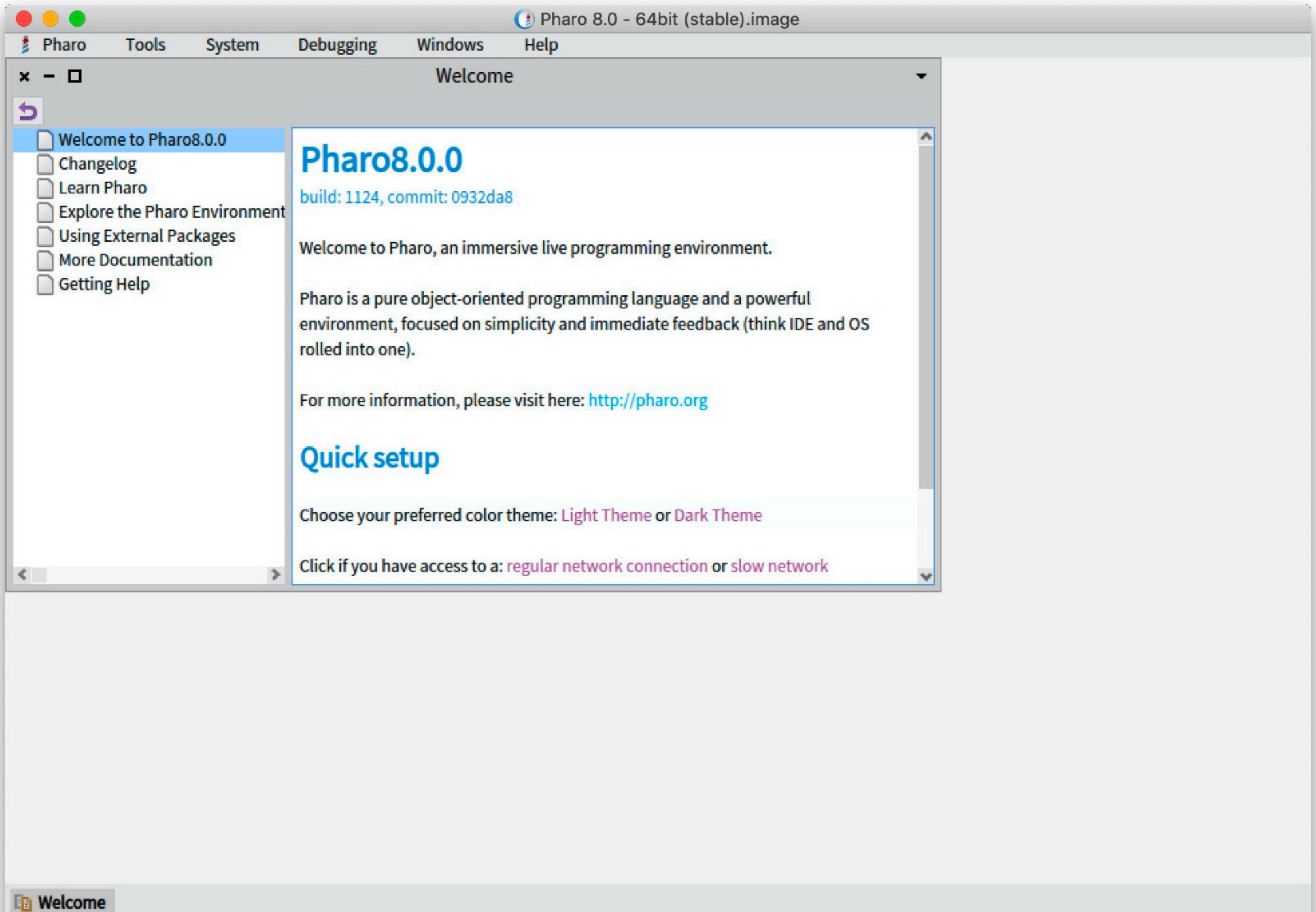
Since the class to be created probably does not yet exist, its name is not defined globally, so we must pass in the name as a symbol (here `#Complex`).

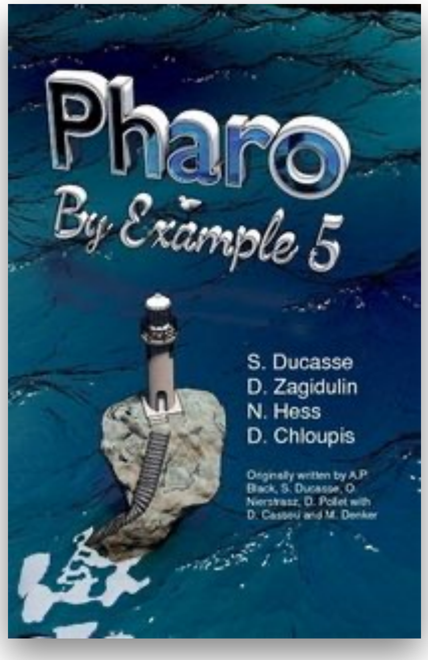
We can also provide the names of its instance variables (or we can update this later). Please ignore classVariableNames and PoolDictionaries — they are almost never needed. The "*category*" is the name of a related group of classes (something like a poor man's package).

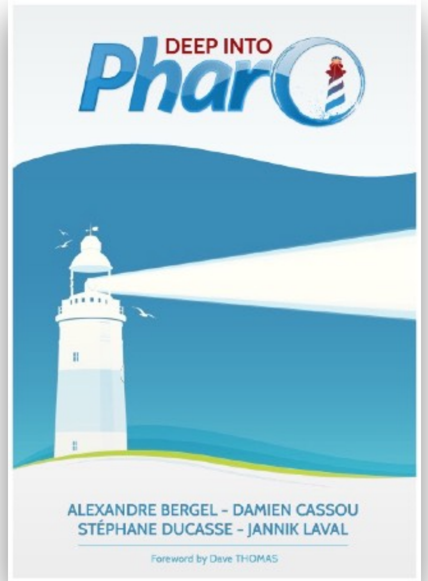
Roadmap

> The origins of Smalltalk
> Syntax in a nutshell
> **Pharo and Gt**
> Demo — the basics
> Demo — live programming with Gt

Pharo — a modern Smalltalk







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

Download it from:

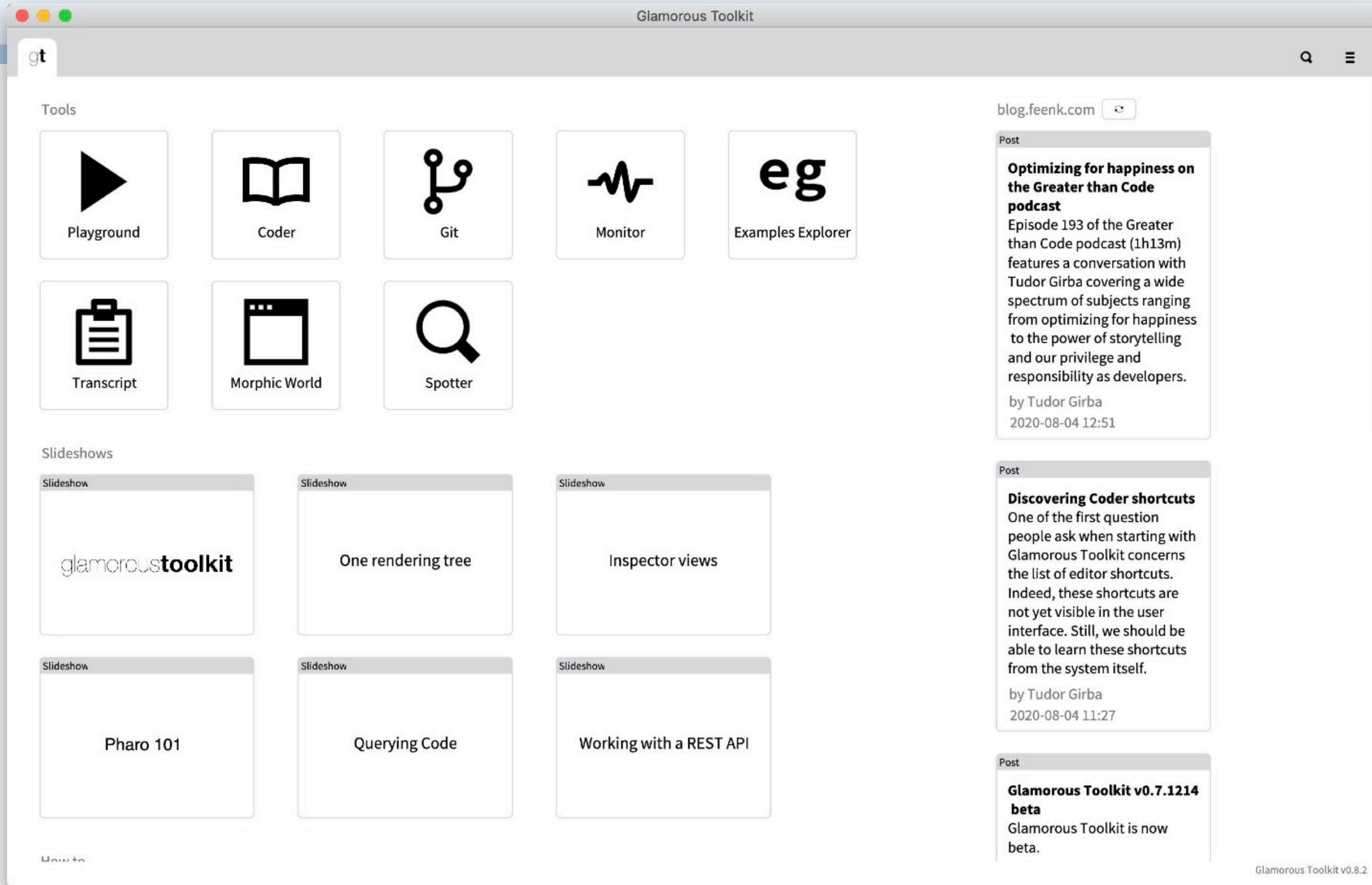
http://pharo.org

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

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.

Two rules to remember

Everything is an object

(Nearly) everything in Smalltalk is an object, which means that you can “grab it” and talk to it. Everything that you see on the screen is an object, so you can interact with it programmatically.

The implementation of Smalltalk itself is build up of objects, so you can grab these objects and explore them. In particular, all the tools are objects, but also classes and methods are objects. This feature is extremely powerful and leads to a style of programming that is different from the usual edit/compile/run development cycle.

Everything happens by sending messages

The only way to make anything happen is by sending messages. To ask "what can I do with this object?" is the same as asking "what messages does it understand?"

The terminology of "message sending" is perhaps unfortunate, as those new to Smalltalk often assume it has something to do with network communication, but one should understand it as a metaphor: you do not "call an operation" of an object, but you politely ask it to do something by sending it a request (a "*message*"). The object then decides how to respond by checking to see if its class has a "*method*" for handling this request. If it does, it performs the method. If not, it asks its superclass if it has such a method, and so on. If this search fails, the object does not understand the message (but let's not get into that now!).

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 slide is 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'.`

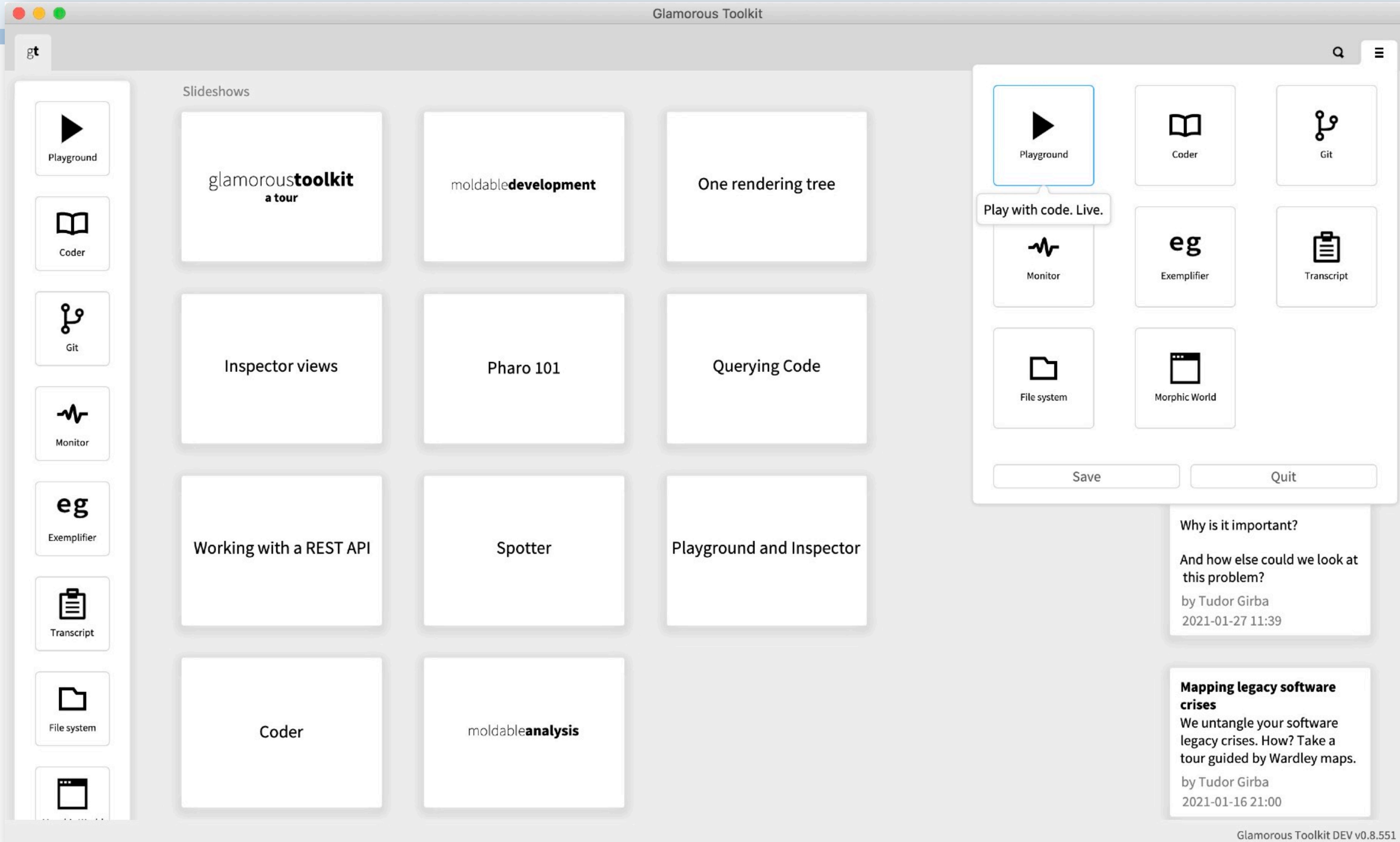
One of the great leaps in OO is to be able to answer the question "How does this work?" with "I don't care".

alanknightsblog.blogspot.ch

Roadmap

> The origins of Smalltalk
> Syntax in a nutshell
> Pharo and Gt
> **Demo — the basics**
> Demo — live programming with Gt

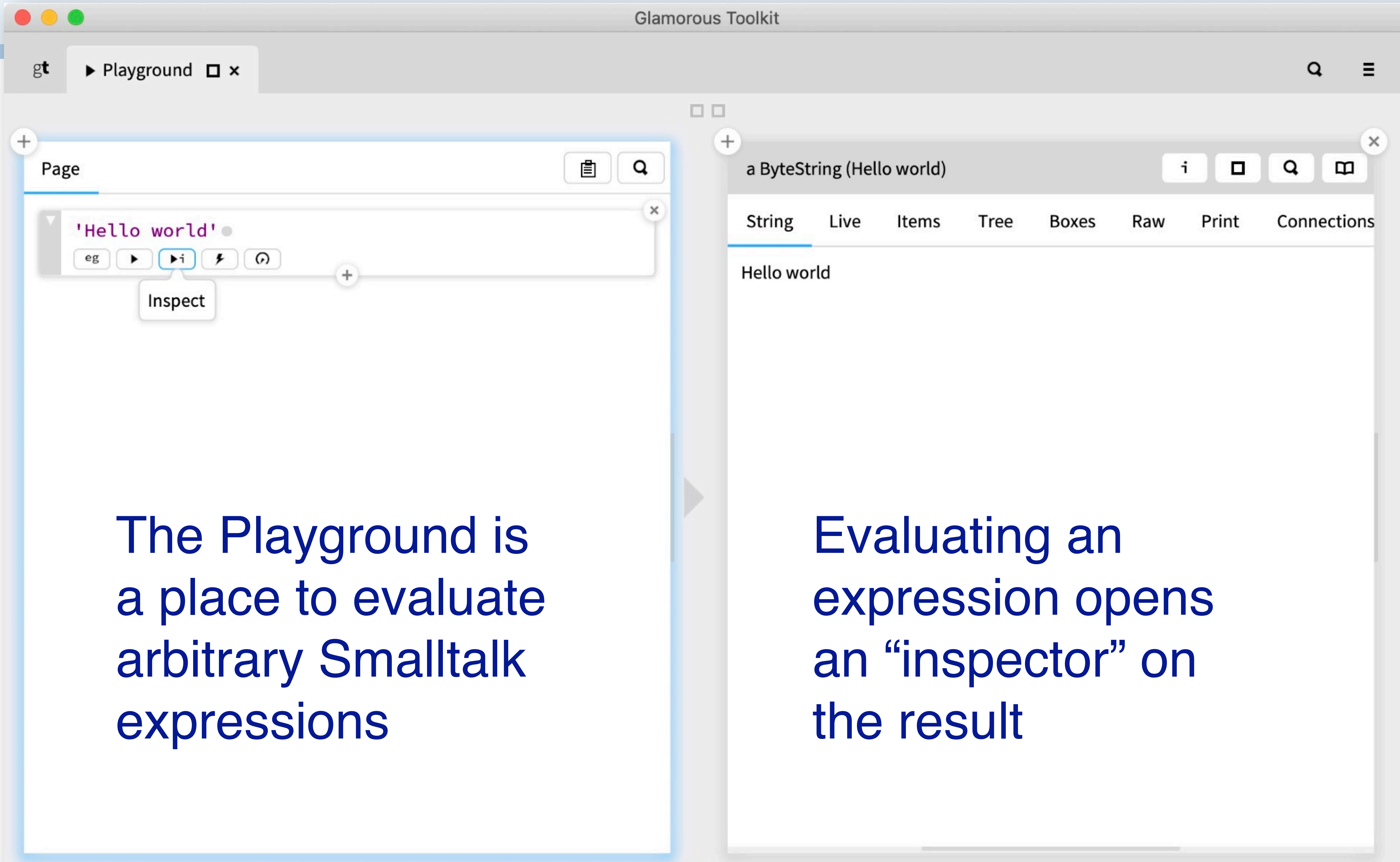
Glamorous Toolkit



The Glamorous Toolkit is both a live programming environment and a “moldable” IDE providing support for data exploration and visualization. The core tools include a Playground for live exploration of code, a Coder for editing and managing code packages, a Git tool for managing repositories, and others.

Various tutorials and blogs are also available from the home window.

The Playground



The Playground is a place to evaluate arbitrary Smalltalk expressions

Evaluating an expression opens an “inspector” on the result

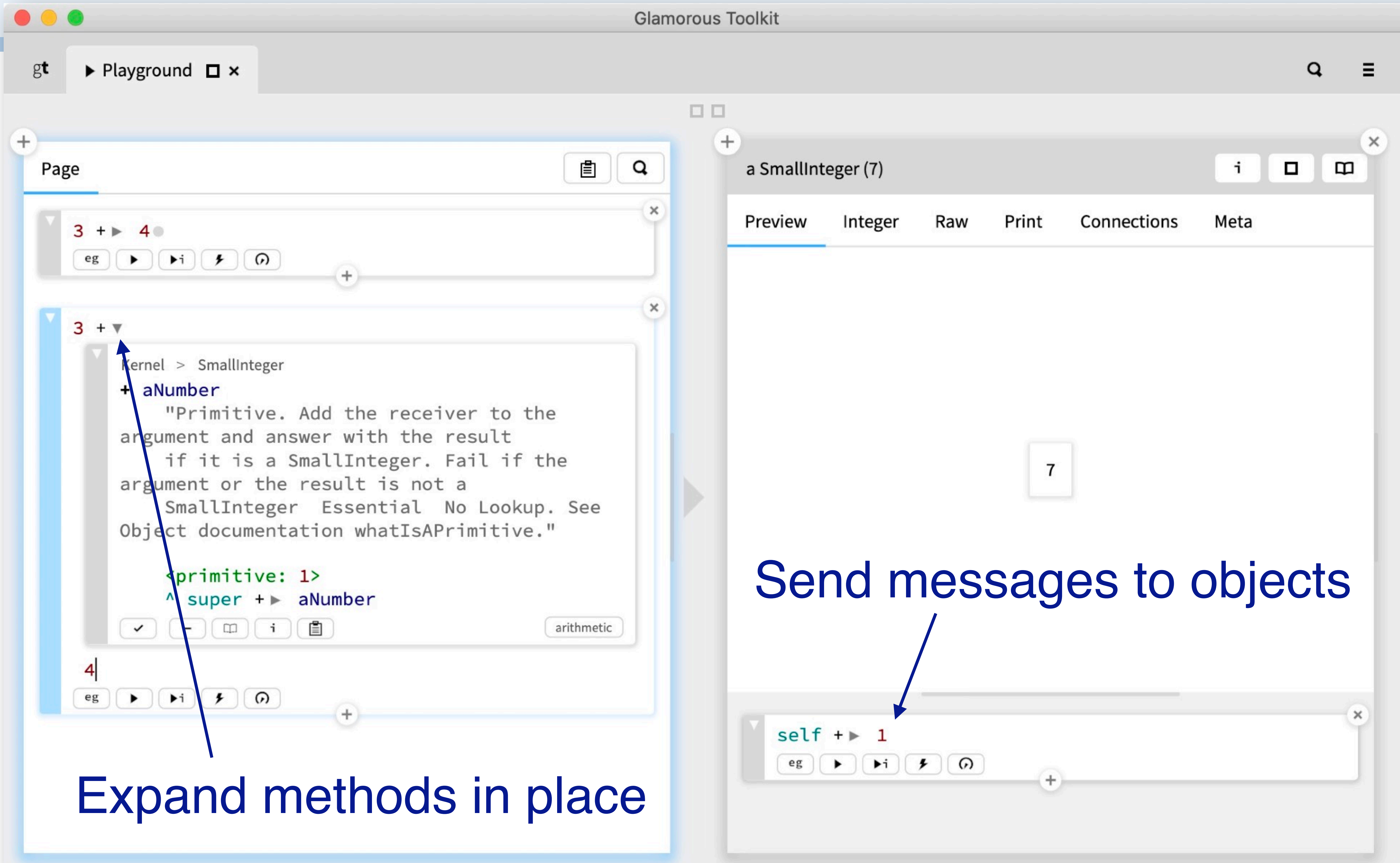
X

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!

The inspector tabs provide various views of the object, such as the “raw” view showing the raw representation. The buttons open various tools, such as a new inspector, or a Coder view of the class.

Exploring objects and code



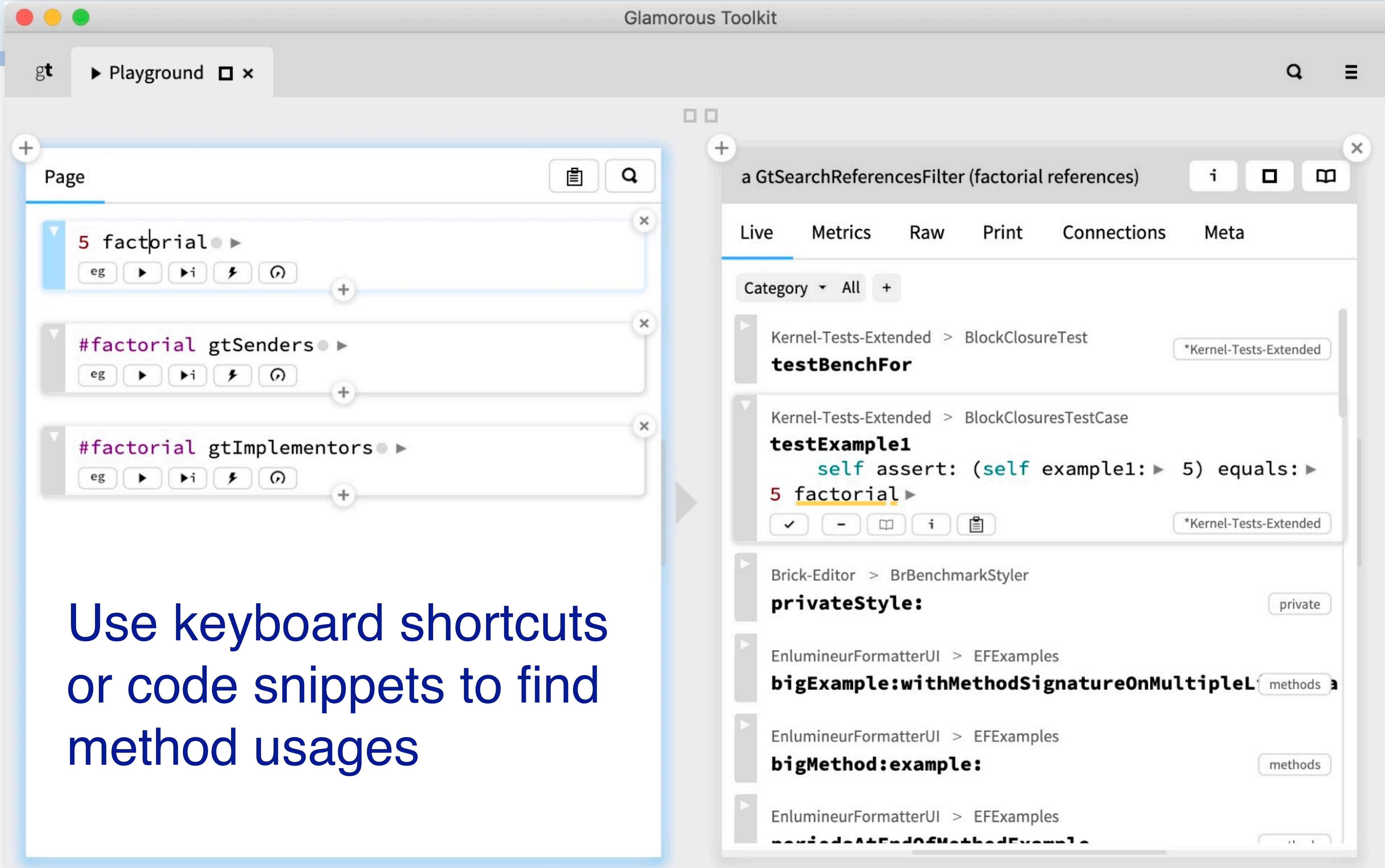
X

You can expand methods in place by clicking on the grey triangle.

You can also pull up a new playground from the bottom of any inspector to evaluate arbitrary code.

NB: `self` is bound to the inspected object.

Finding seNders and iMplementors



Use keyboard shortcuts or code snippets to find method usages

X

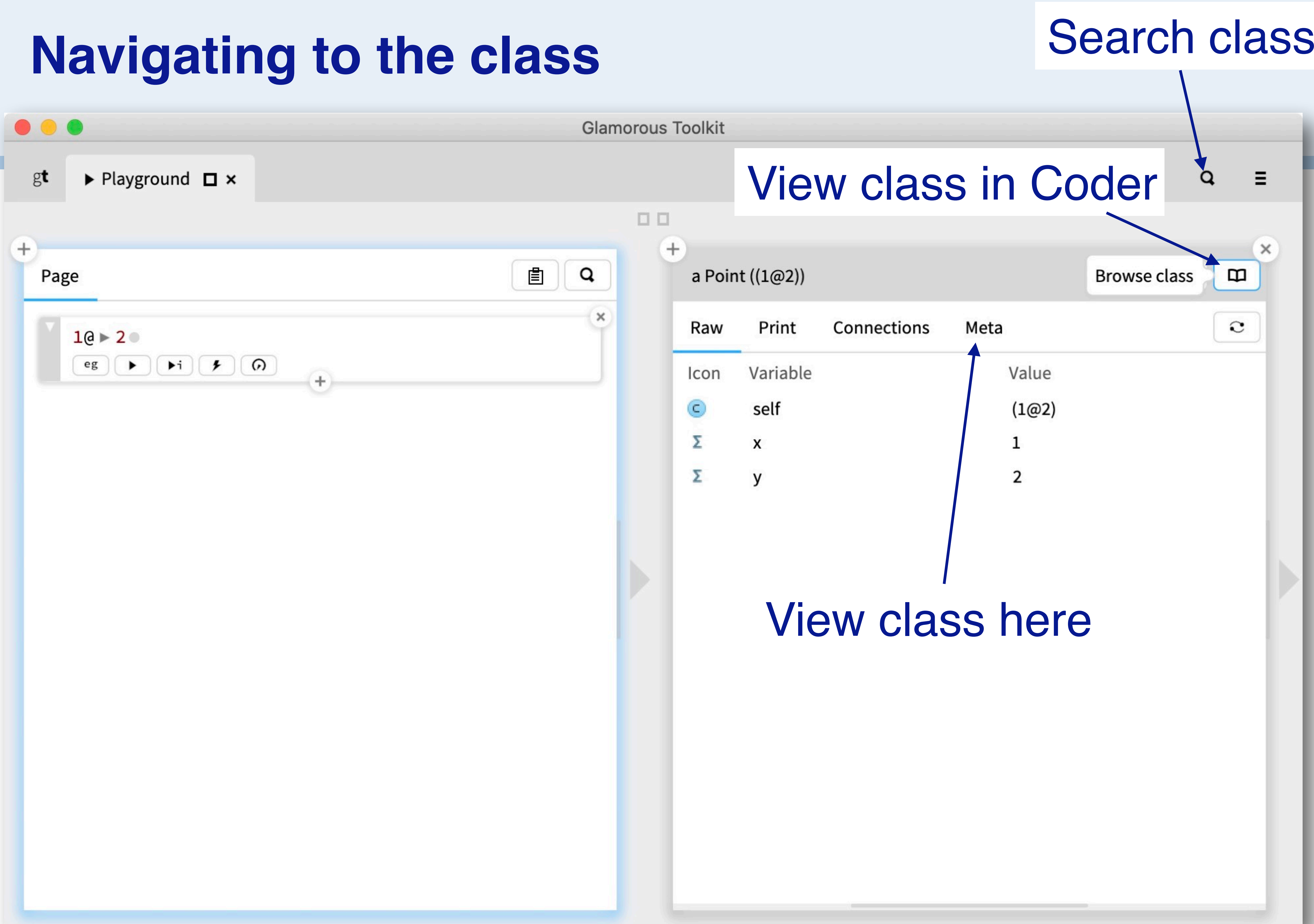
To find all the implementations of a method, just position the mouse within the method's name, and evaluate Command+M (for iMplementors). You can also find all methods that send it as a message by evaluating Command+N (for seNders).

Gt also has extensive support for programmatically querying code. For example, you can find the senders and implementors of the `factorial` method by evaluating these snippets:

```
#factorial gtSenders
```

```
#factorial gtImplementors
```

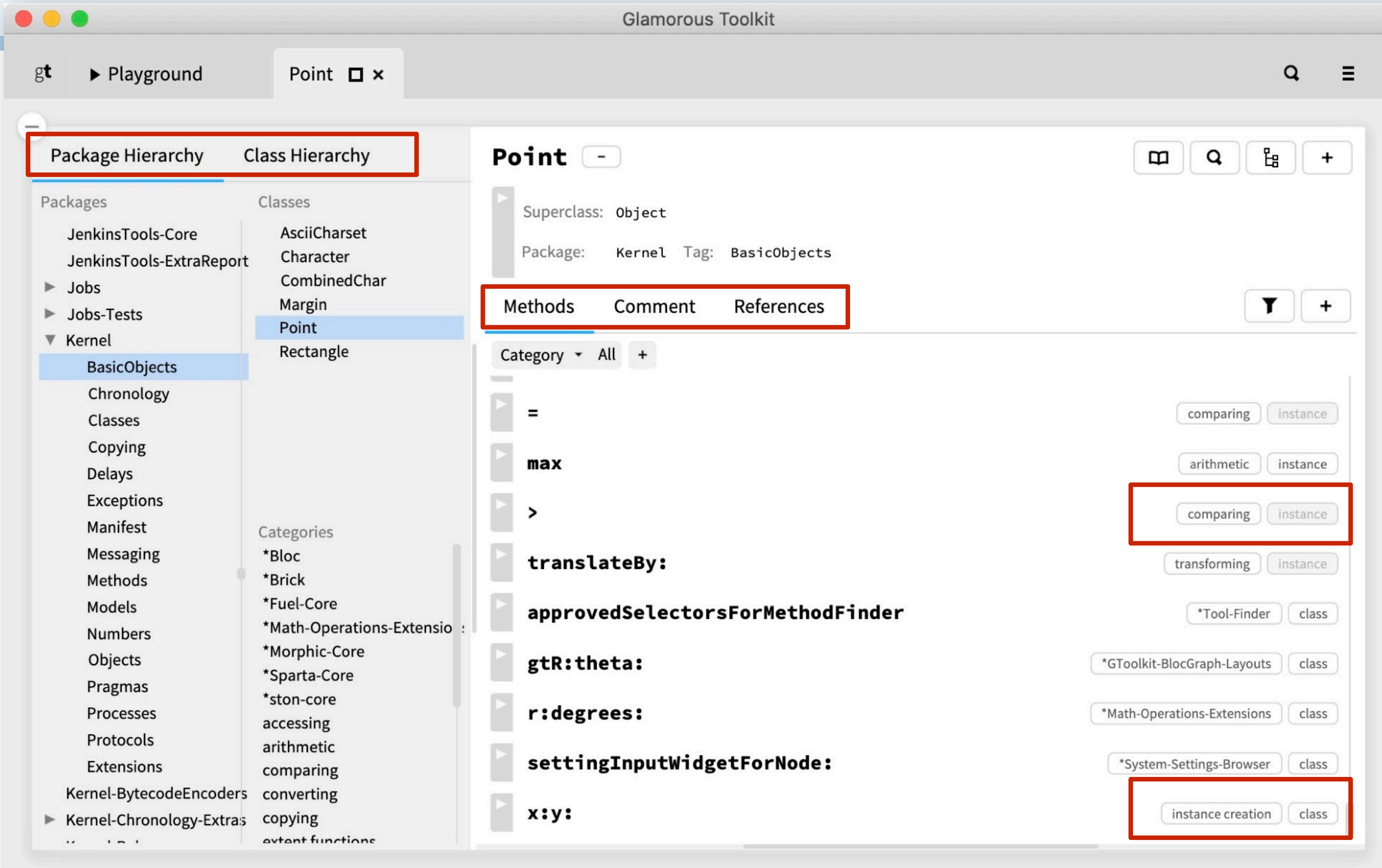
Navigating to the class



There are numerous ways to navigation to the class of an object. You can view the class directly in the “Meta” tab, or open a dedicated Coder pane with the “Browse” button.

Alternatively you can search for a class (or anything else) with the Spotter, or open a new Coder pane and search there.

The Coder



Glamorous Toolkit
gt
Playground
Point
Package Hierarchy
Class Hierarchy
Packages
JenkinsTools-Core
JenkinsTools-ExtraReport
Jobs
Jobs-Tests
Kernel
BasicObjects
Chronology
Classes
Copying
Delays
Exceptions
Manifest
Messaging
Methods
Models
Numbers
Objects
Pragmas
Processes
Protocols
Extensions
Kernel-BytecodeEncoders
Kernel-Chronology-Extras
Classes
AsciiCharset
Character
CombinedChar
Margin
Point
Rectangle
Categories
*Bloc
*Brick
*Fuel-Core
*Math-Operations-Extensio
*Morphic-Core
*Sparta-Core
*ston-core
accessing
arithmetic
comparing
converting
copying
Point
Superclass: Object
Package: Kernel Tag: BasicObjects
Methods
Comment
References
Category All
=
comparing instance
max
arithmetic instance
>
comparing instance
translateBy:
transforming instance
approvedSelectorsForMethodFinder
*Tool-Finder class
gtR:theta:
*GToolkit-BlocGraph-Layouts class
r:degrees:
*Math-Operations-Extensions class
settingInputWidgetForNode:
*System-Settings-Browser class
x:y:
instance creation class

X

The Coder is a dedicated tool for editing and managing classes and methods. You can view classes either within their package hierarchy or class hierarchy.

You can also view the methods of a class, or the class comment, or you can browse references to the class. Other panes will appear if they are relevant such as examples.

Methods in Smalltalk are tagged by their category, such as “comparing” or “instance creation”. Note that “class methods” are analogous to static methods in Java — you invoke them by sending the message to the class, not the object.

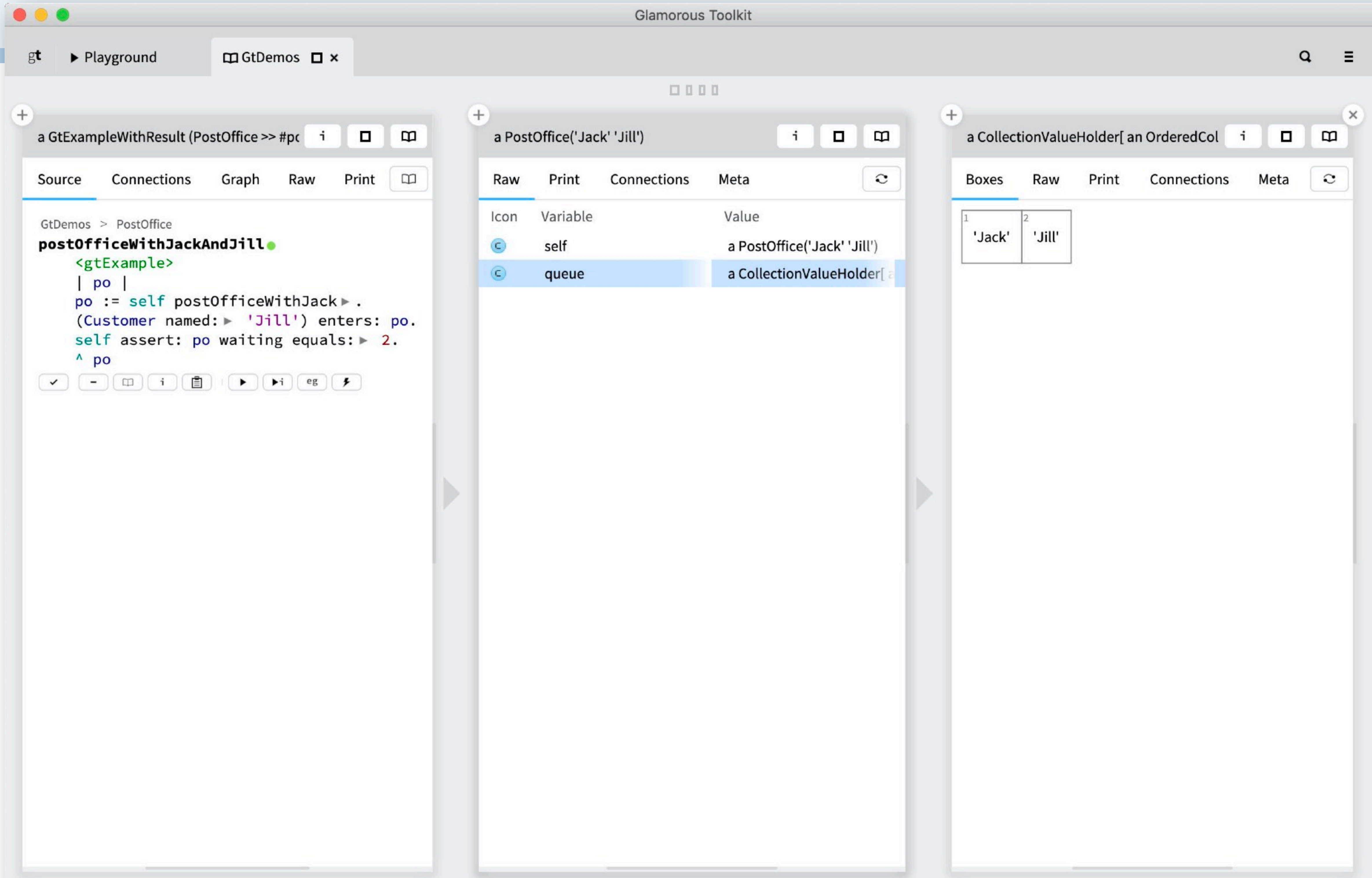
```
Point x: 1 y: 2
```

will create a new `Point` object `1@2`.

Roadmap

> The origins of Smalltalk
> Syntax in a nutshell
> Pharo and Gt
> Demo — the basics
> **Demo — live programming with Gt**

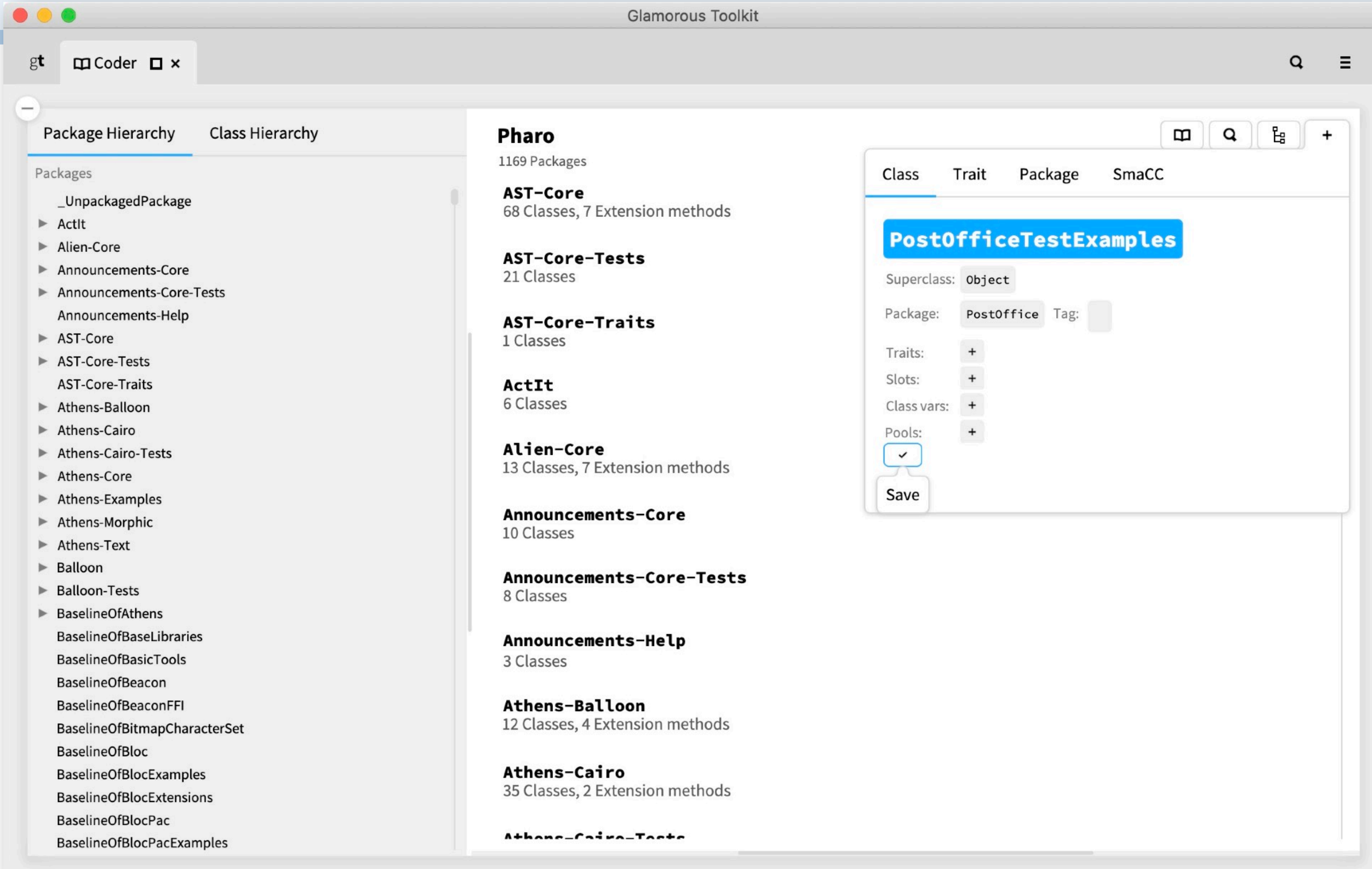
Demo: Defining classes and methods



This demo script can also be found in the same github repo listed earlier.

Here we apply test-driven development to simulate a Post Office serving customers.

Creating a class



X

Use the Coder to create a new class, specifying its name (`PostOfficeTestExamples`), superclass (`Object`), and package (`PostOffice`). You can also specify a tag (sub-package), instance variables (slots), and other properties.

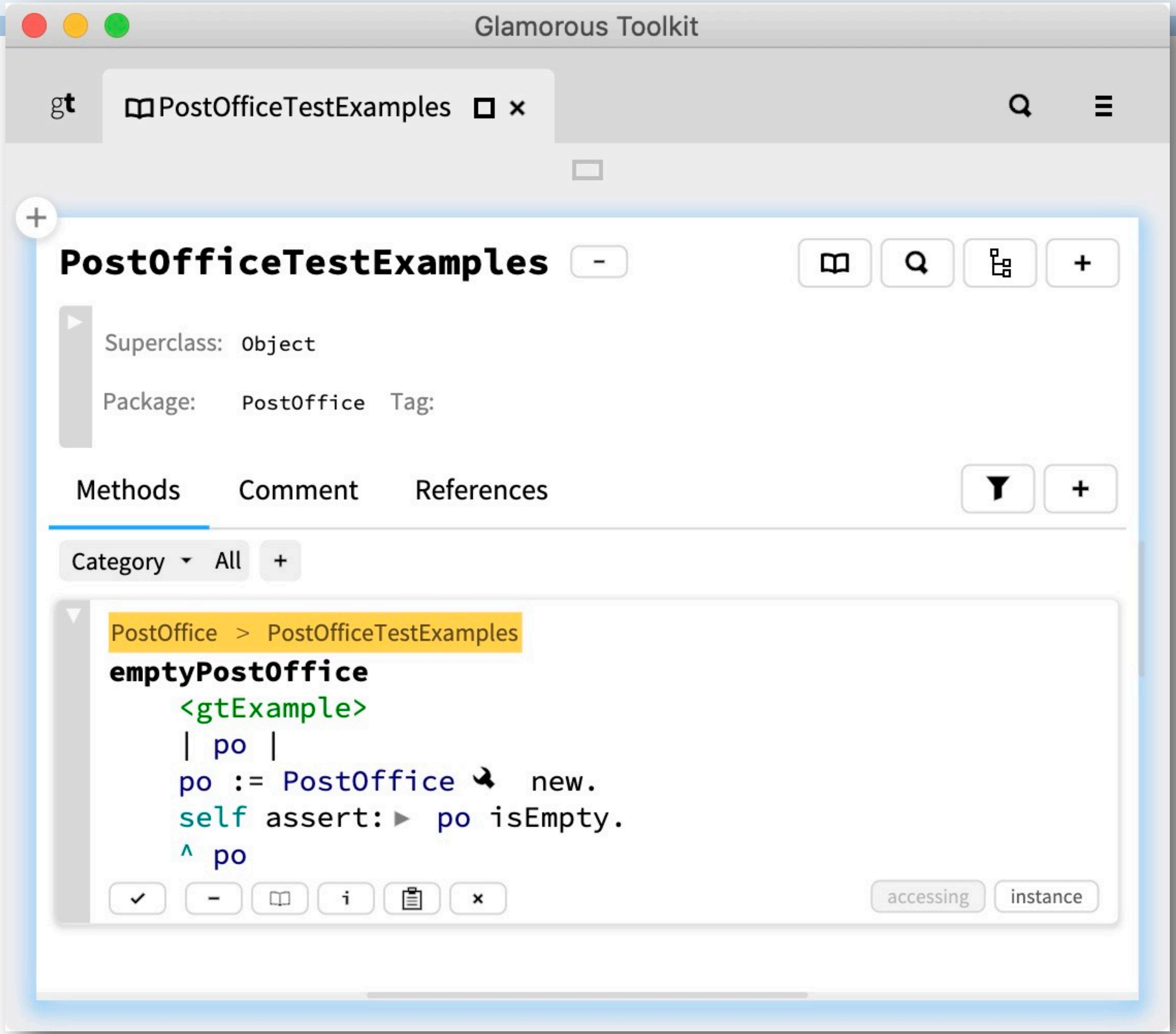
Click the checkmark (√) to commit.

Note that you can also create class programmatically by sending a message to its superclass ("everything happens by sending messages"):

```
Object subclass: #PostOfficeTestExamples
     instanceVariableNames: ''
     classVariableNames: ''
     package: 'PostOffice'
```

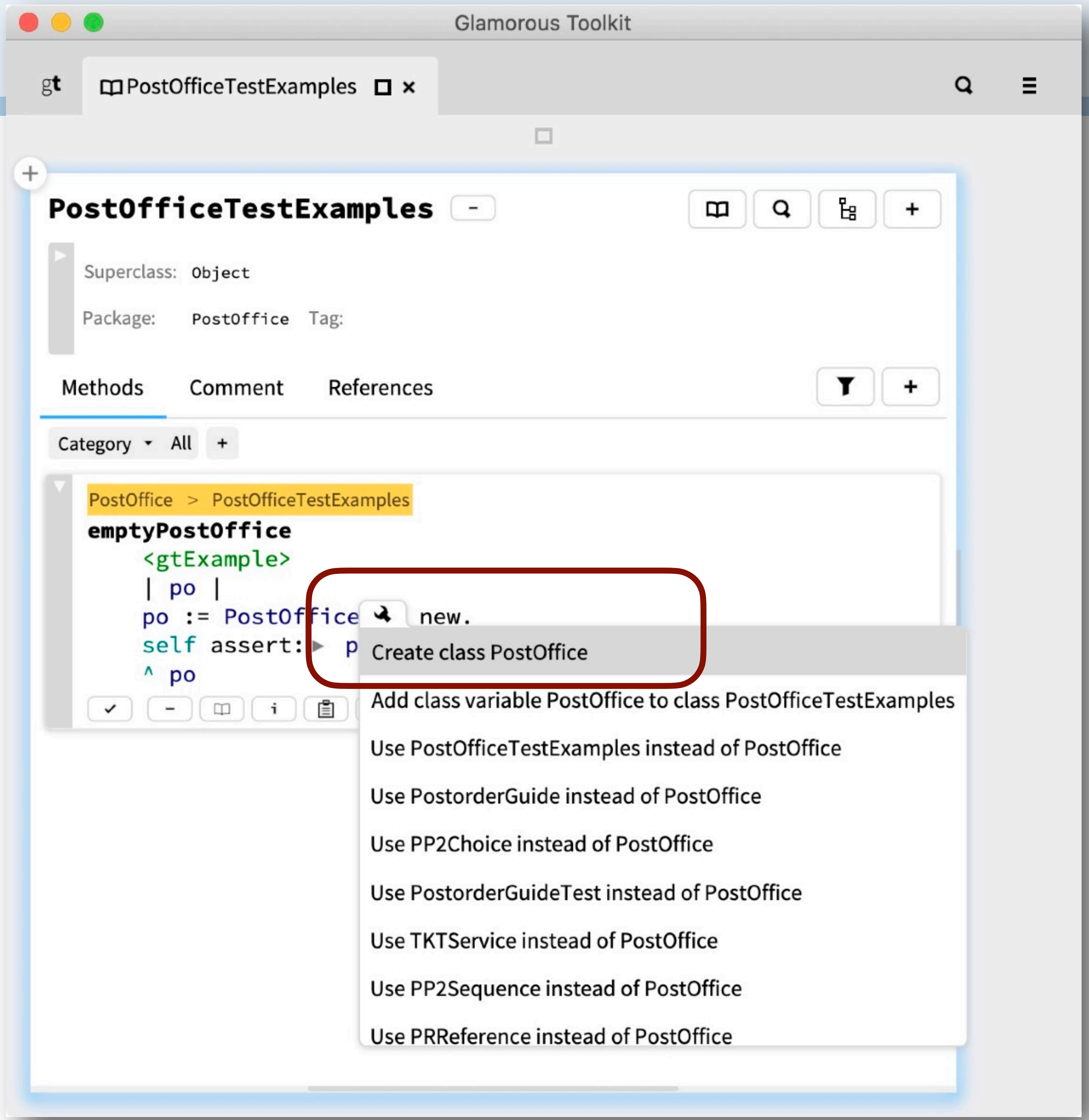
Creating test examples

In Gt, tests are written as *example methods* that return an example object.
This allows tests to be composed, and also allows the results to be inspected and explored.
Just add the annotation `<gtExample>` to turn a method into a (test) example.



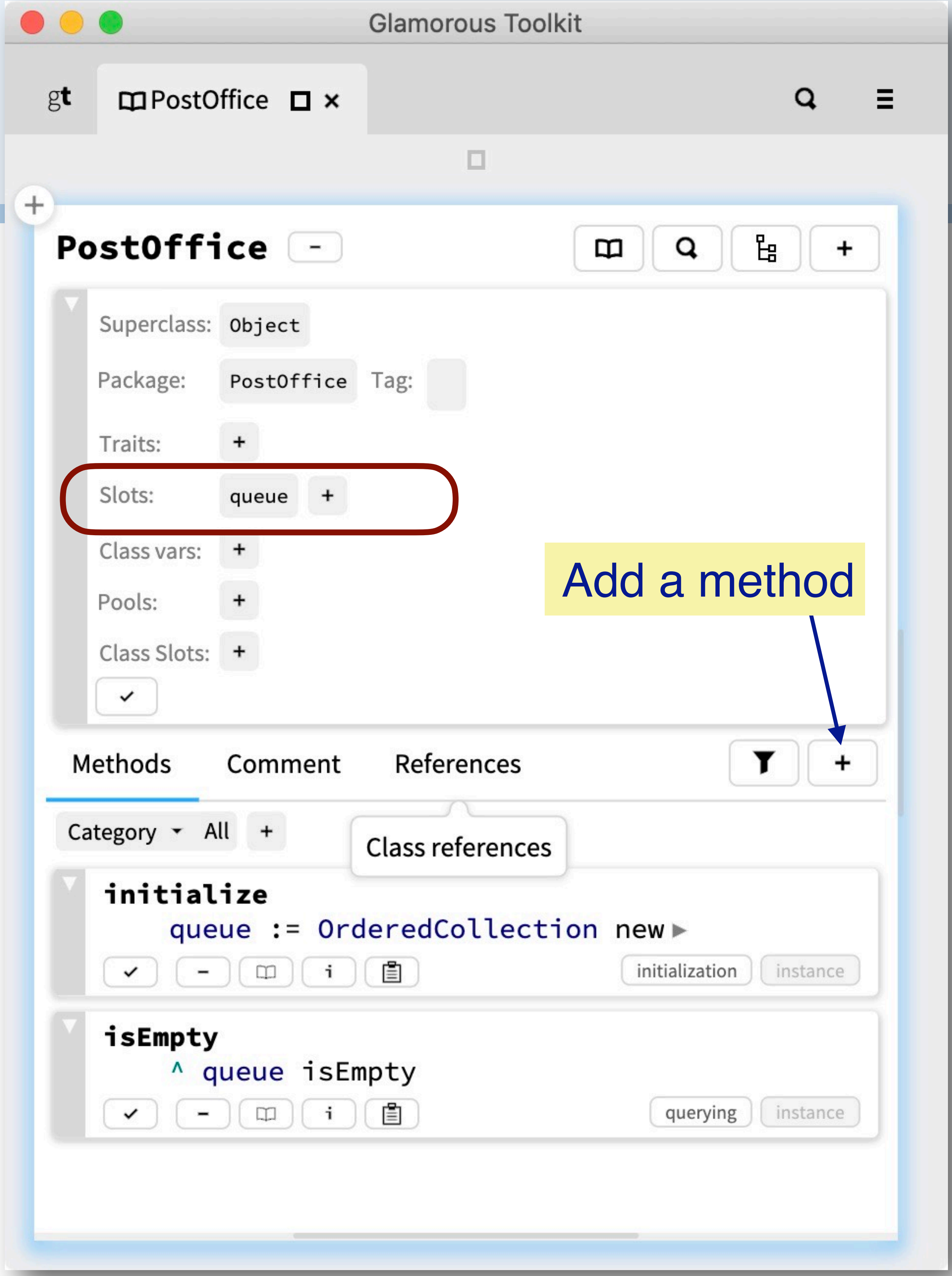
Quick fixes

Like most modern IDEs, Gt provides quick fixes.
They appear as a "wrench" icon, not only within the Coder, but anywhere you might type a code snippet (such as the Playground).



Initialization

The `initialize` method is run by default in Pharo Smalltalk for all newly created objects.
Here we initialize a `queue` slot (instance variable) for new `PostOffice` instances.

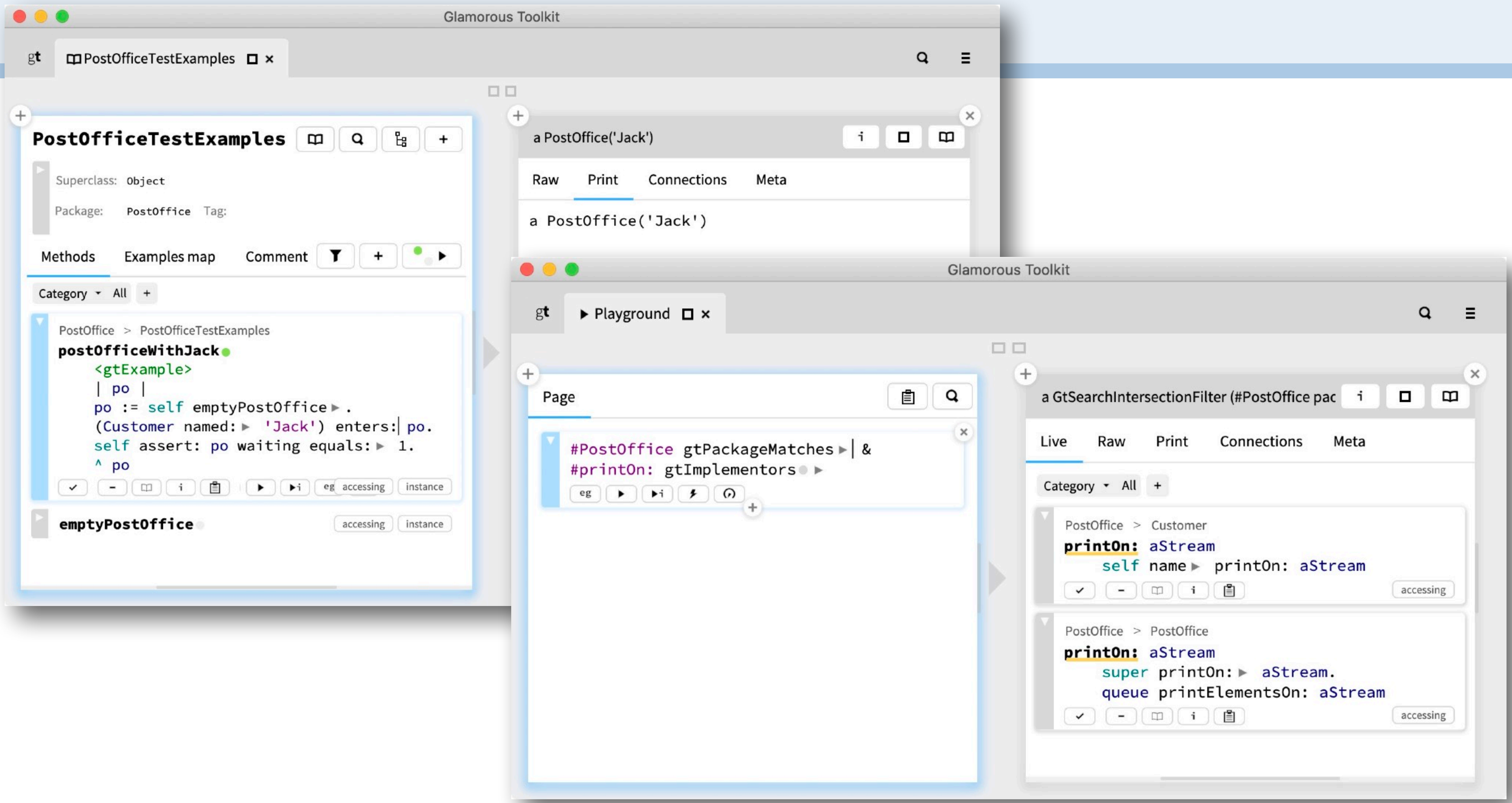


X

Unless your class is a direct subclass of `Object`, it is best practice to perform `super initialize` as the first statement in your initialize method (just as in all OO languages).

We initialize queue to `OrderedCollection`, as it provides everything we need to model a queue, and there is no dedicated Queue class.

Printing objects



We can compose test examples, and implement `#printOn:` to make objects printable

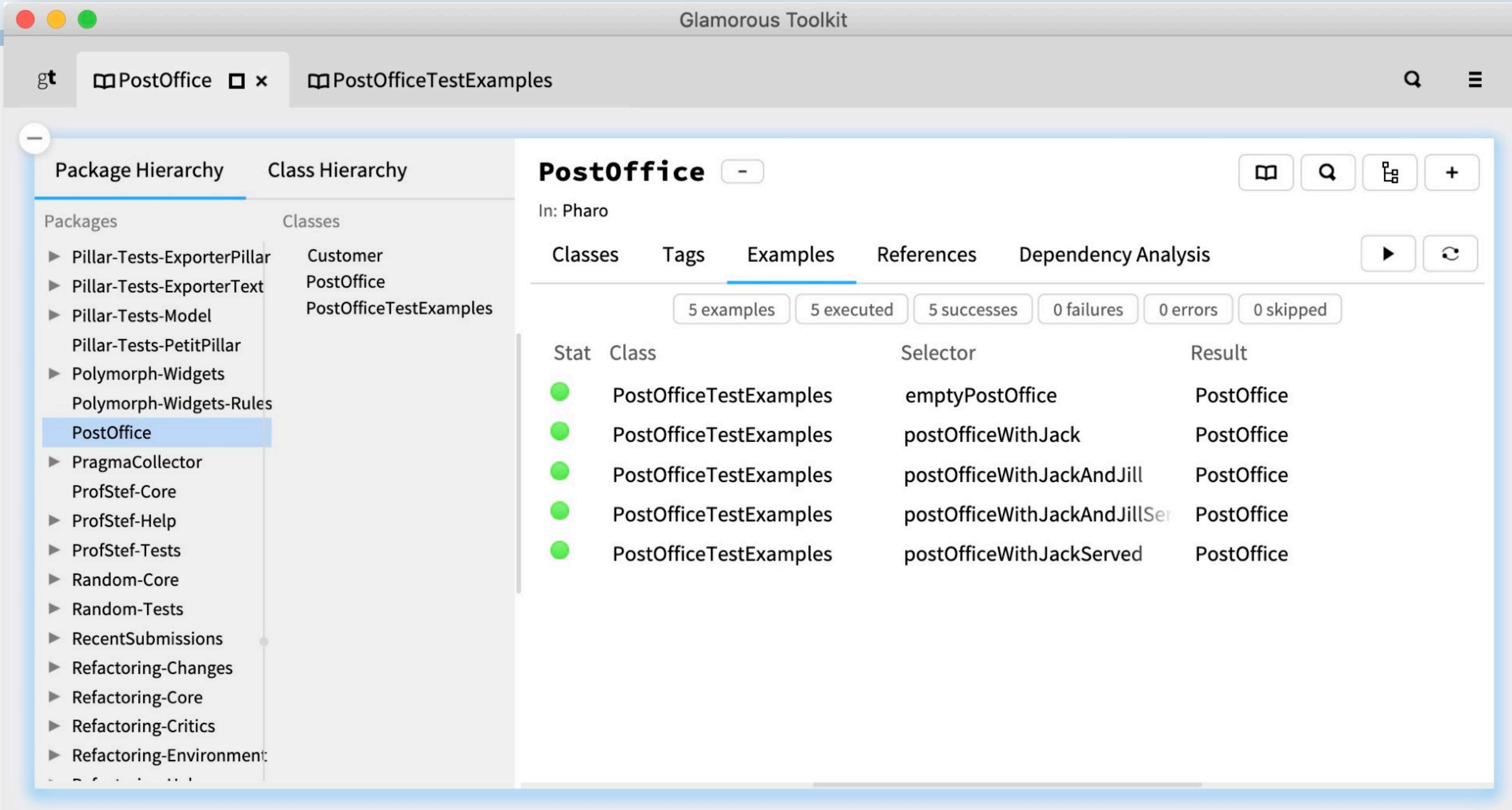
x

The `postOfficeWithJack` test example is composed from the `emptyPostOffice` example.

The default print method of classes just show the class name, so we override it in both `PostOffice` and `Customer` to show the list of names of customers in the queue.

Note the use of a Gt query to find all the `printOn:` method implementations in our package.

Running all the tests



The package view provides a way to run all the tests

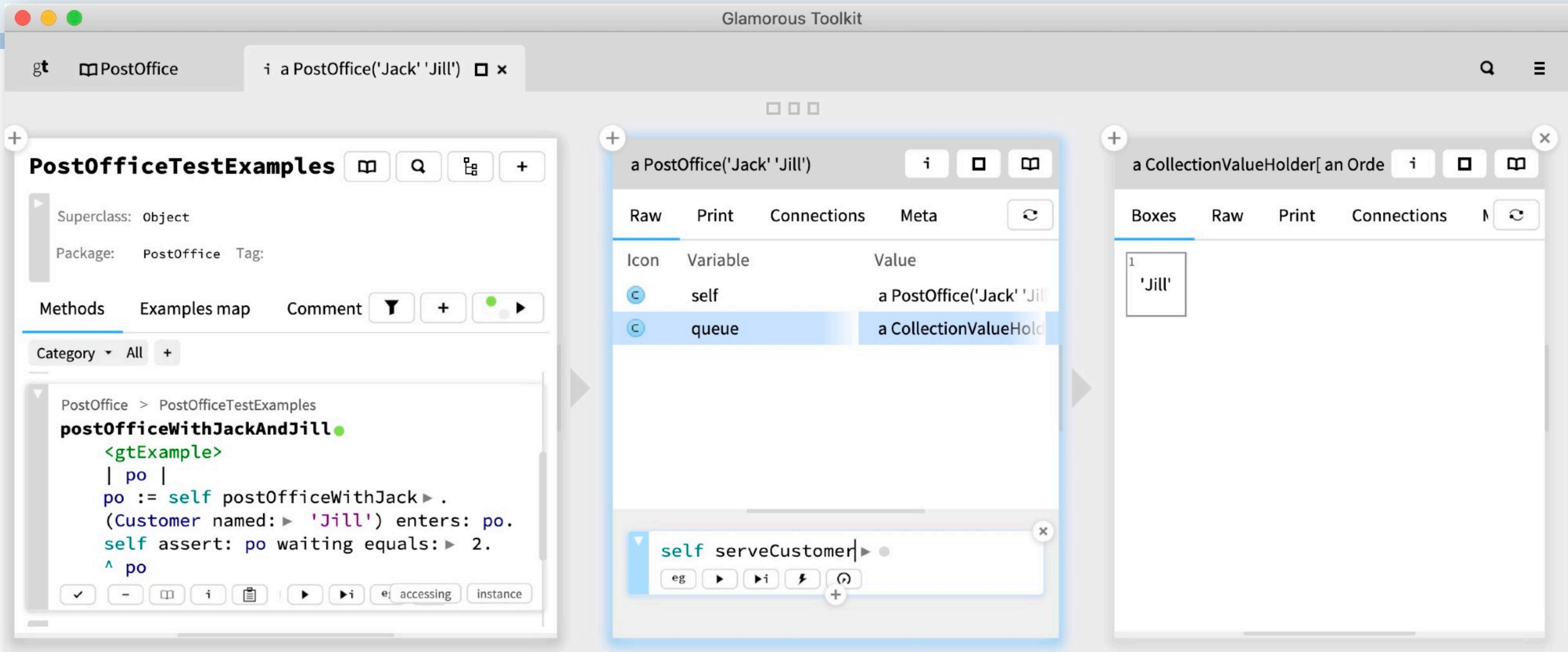
X

You can also run a query to extract all the test examples from a package:

```
(#PostOffice gtPackageMatches
& #gtExample gtPragmas) gtExamples
```

(Everything happens by sending messages.)

Enabling a “live” view



By wrapping the queue as a “value holder” obeying MVC, we obtain a live view of the PostOffice for free

X

If we change the initailization method of the PostOffice as follows:

```
initialize
    queue := OrderedCollection new asValueHolder
```

the queue will be wrapped as a “value holder” that produces `ValueChanged` events when the collection is updated. The Boxes view then updates itself automatically.

What we didn’t see

- Smalltalk is fully reflective
  - Classes are objects too; the entire system is implemented in itself
- The debugger is your friend
  - Sophisticated live debugging
  - You can change the system while debugging
- You can’t lose code
  - All changes are stored and can be replayed
- “Moldable” views in Gt
  - You can create dedicated live visualizations for objects

What you should know!

- ✎ What are the key differences between Smalltalk, C++ and Java?
- ✎ What is at the root of the Smalltalk class hierarchy?
- ✎ What kinds of messages can one send to objects?
- ✎ What is a cascade?
- ✎ Why does `1+2/3 = 1` in Smalltalk?
- ✎ How are control structures realized?
- ✎ How is a new class created?
- ✎ What are categories for?
- ✎ What are Factory methods? When are they useful?

Can you answer these questions?

- ✎ *Which is faster, a program written in Smalltalk, C++ or Java?*
- ✎ *Which is faster to develop & debug, a program written in Smalltalk, C++ or Java?*
- ✎ *How are Booleans implemented?*
- ✎ *Is a comment an Object? How would you check this?*
- ✎ *What is the equivalent of a static method in Smalltalk?*
- ✎ *How do you make methods private in Smalltalk?*
- ✎ *What is the difference between = and ==?*
- ✎ *If classes are objects too, what classes are they instances of?*