

**Legacy Code**  
**Test Seams**  
**Automated Sketches**  
**Pharo Smalltalk**

**CubeServ<sup>®</sup>**



**Martin Fowler** ✓

@martinfowler

Folge ich



Good software architecture advice is always worth considering, but it should never be always carried out

🌐 Original (Englisch) übersetzen

RETWEETS

264

GEFÄLLT

182



12:07 - 5. Juni 2015

← 8

↻ 264

♥ 182

**Coding:**  
**constructing a machine**

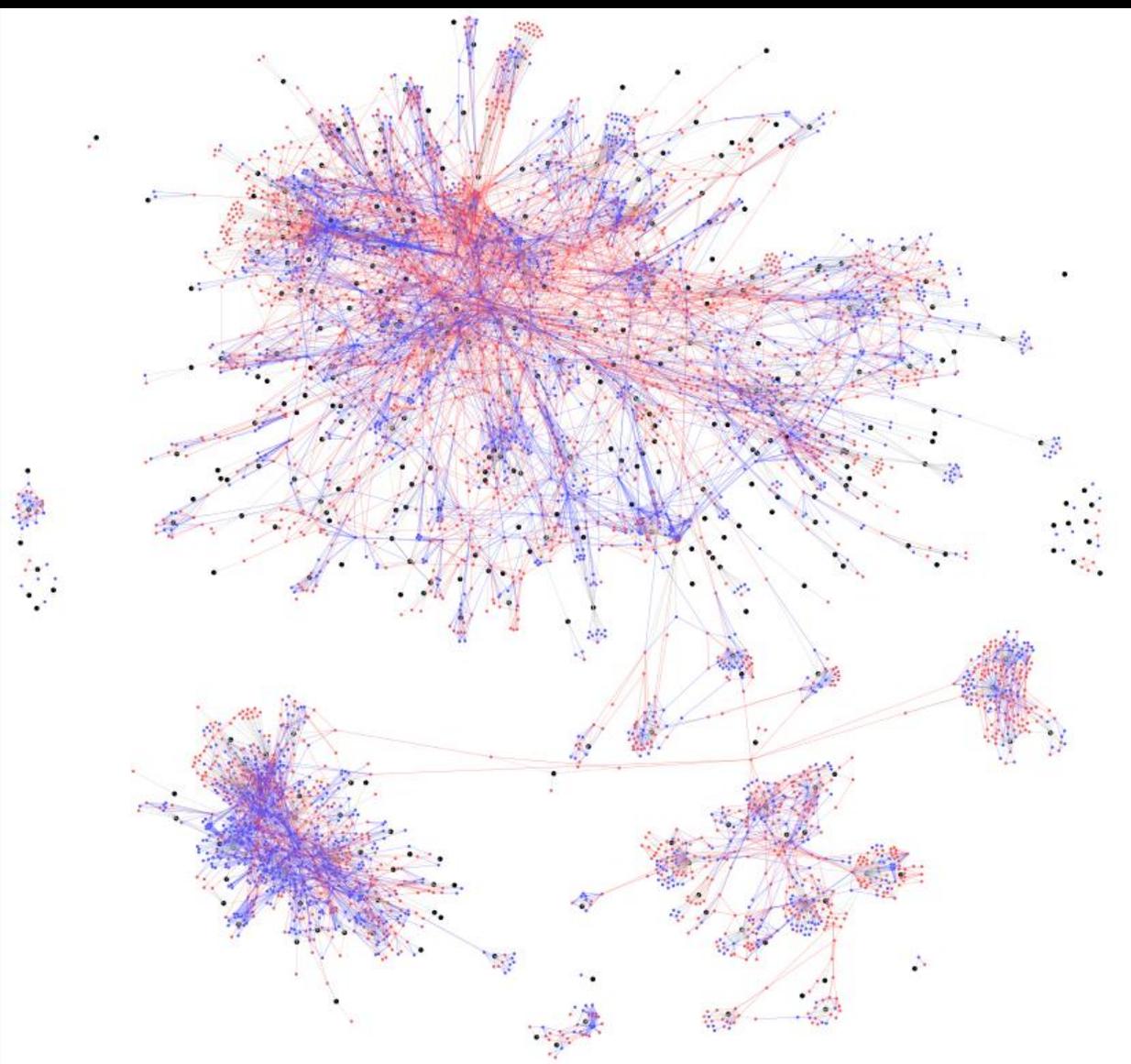
**There are many tips to  
construct **well****

**There is no simple way  
to construct a good  
machine**

**A machine has to fulfill  
many requirements  
that contradict each  
other**

**Low weight  $\leftrightarrow$  robust**

**Coding** becomes  
**complex** quite soon





Robert C. Martin Series



**WORKING  
EFFECTIVELY  
WITH  
LEGACY CODE**

Michael C. Feathers

*legacy code*

is

*code without tests*





# **UNIT Tests**

## **Automatic tests**



Do not use much time  
with finding errors

But there are dependencies?

SELECT

INSERT

Calls to Code that changes ...

**Common advice**

**Use **interfaces****

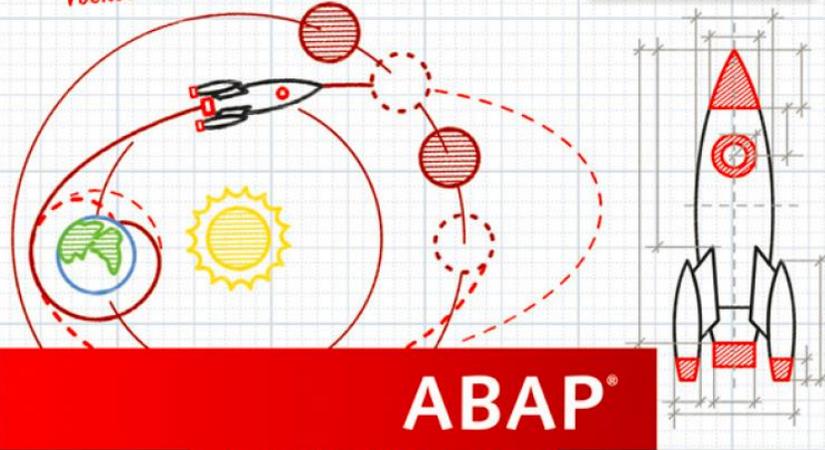
But using interfaces is  
often  
not easy



Paul Hardy

```
DATA(rocket) = zcl_rocket_builder=>build( ).  
rocket->launch( abap_true ).
```

SAP PRESS



# ABAP<sup>®</sup>

## to the Future

- ▶ Discover the latest and greatest features in the ABAP universe
- ▶ Explore the new worlds of SAP HANA, BRFplus, BOPF, and more
- ▶ Propel your code and your career into the future

Use instance methods  
and  
inherit mock classes



**Simpler**

Difficult and  
dangerous to add  
to existing code  
that has no tests

Even simpler

# TEST-SEAMS in ABAP 7.50

Works on static methods

Needs no additional  
classes or methods

Add to existing code  
without risk

# Read scn blogs

## ABAP News for Release 7.50 – Test Seams and Test Injections

October 23, 2015 | 1,011 Views |



Horst Keller 

more by this author

ABAP Development

## Working effectively with ABAP legacy code – ABAP Test Seams are your friend

February 6, 2016 | 290 Views |



Klaus Ziegler

more by this author

ABAP Testing and Analysis

**Replace  
accesses to  
persistent data**

## Test Seam

```
TEST-SEAM read_content_seam.  
  SELECT *  
    FROM sflight  
   WHERE carrid IN @carrid_range AND  
          fldate EQ @sy-datum  
 INTO TABLE @flights.  
END-TEST-SEAM.
```

## Injection

```
TEST-INJECTION read_content_seam.  
  flights =  
    VALUE #( ( carrid = 'LHA'  
              connid = 100 )  
            ( carrid = 'AFR'  
              connid = 900 ) ).  
END-TEST-INJECTION.
```

**Test Seam**

```
TEST-SEAM store_content_seam.  
  MODIFY sflight  
    FROM TABLE @new_flights.  
END-TEST-SEAM.
```

**Injection**

```
TEST-INJECTION store_content_seam.  
  cl_abap_unit_assert=>assert_equals(  
    act = new_flights  
    exp = global_buffer=>exp_flights ).  
END-TEST-INJECTION.
```

**Replace**  
**authority checks**

## Test Seam

```
TEST-SEAM authorization_seam.  
  AUTHORITY-CHECK OBJECT 'S_CTS_ADMI'  
    ID 'CTS_ADMFCT' FIELD 'TABL'.  
END-TEST-SEAM.  
  
IF sy-subrc = 0.  
  is_authorized = abap_true.  
ENDIF.
```

## Injection

```
TEST-INJECTION authorization_seam.  
  sy-subrc = 0.  
END-TEST-INJECTION.
```

**Inject**  
**test doubles**

### Test Seam

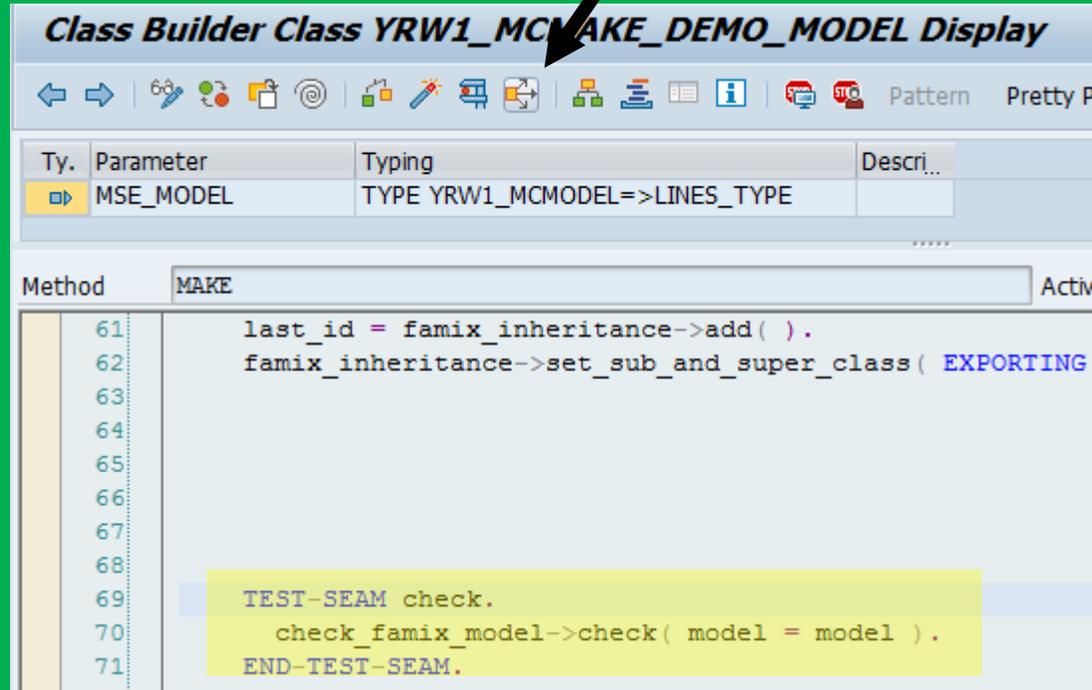
```
TEST-SEAM instantiation_seam.  
  me->oref = NEW #( ).  
END-TEST-SEAM.
```

### Injection

```
TEST-INJECTION instantiation_seam.  
  me->oref = NEW dummy_class( ).  
END-TEST-INJECTION.
```

Sometimes small changes needed so that TEST-SEAMS can be added

Where Used is working



The screenshot shows the 'Class Builder' interface for the class 'YRW1\_MCMODEL'. The 'MAKE' method is selected, and the 'Where Used' tool is active. The tool displays a table of parameters used by the method.

Ty.	Parameter	Typing	Descri...
	MSE_MODEL	TYPE YRW1_MCMODEL=>LINES_TYPE	

Below the table, the source code for the 'MAKE' method is displayed. A yellow highlight is placed over the 'TEST-SEAM check' block, which is highlighted in blue in the original image.

```
61 last_id = famix_inheritance->add( ).
62 famix_inheritance->set_sub_and_super_class( EXPORTING
63
64
65
66
67
68
69 TEST-SEAM check.
70     check_famix_model->check( model = model ).
71 END-TEST-SEAM.
```



TEST-SEAMS for new coding?

Only way to add Unit tests  
to code that accesses the  
outside world

Simple

Better than **no** Unit Tests

I use TEST-SEAMS in  
new coding  
when it is the best  
option

**Less coding – short! 😊**

Ask colleagues for help –  
Starting with Unit Tests is  
not easy

Feeling more save

Understanding how  
a method is to be  
used

# **Tools – Automatic diagram generation**

What depends on this?

What is it influencing?

Where am I?

What is the big picture?

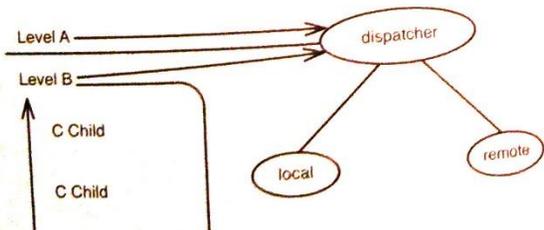
## Notes/Sketching

When reading through code gets confusing, it pays to start drawing pictures and making notes. Write down the name of the last important thing that you saw, and then write down the name of the next one. If you see a relationship diagrams or function call graphs using some special notation—although, if things get more confusing, you might want to get more formal or neater to organize your thoughts. Sketching things out often helps us see things in a different way. It's also a great way of maintaining our mental state when we are trying to understand something particularly complex.

Figure 16.1 is a re-creation of a sketch that I drew with another programmer the other day as we were browsing code. We drew it on the back of a memo (the names in the sketch have been changed to protect the innocent).

The sketch is not very intelligible now, but it was fine for our conversation. We learned a bit and established an approach for our work.

Doesn't everyone do this? Well, yes and no. Few people do it frequently. I suspect that the reason is because there really isn't any guidance for this sort of thing, and it's tempting to think that every time we put pen to paper, we should be writing a snippet of code or using UML syntax. UML is fine, but so are blobs and lines and shapes that would be indecipherable to anyone who wasn't there when we drew them. The precision doesn't have to be on paper. The paper is just a tool to make conversation go easier and help us remember the concepts we're discussing and learning.



The really great thing about sketching parts of a design as you are trying to understand them is that it is informal and infectious. If you find this technique useful, you don't have to push for your team to make it part of its process. All you have to do is this: Wait until you are working with someone trying to understand some code, and then make a little sketch of what you are looking at as you try to explain it. If your partner is really engaged in learning that part of the system too, he or she will look at the sketch and go back and forth with you as you figure out the code.

When you start to do local sketches of a system, often you are tempted to take some time to understand the big picture. Take a look at Chapter 17, *My Application Has No Structure*, for a set of techniques that make it easier to understand and tend a large code base.

## Listing Markup

Sketching isn't the only thing that aids understanding. Another technique that I often use is *listing markup*. It is particularly useful with very long methods. The idea is simple and nearly everyone has done it at some time or another, but, frankly, I think it is underused.

The way to mark up a listing depends on what you want to understand. The first step is to print the code that you want to work with. After you have, you can use *listing markup* as you try to do any of the following activities.

### Separating Responsibilities

If you want to separate responsibilities, use a marker to group things. If several things belong together, put a special symbol next to each of them so that you can identify them. Use several colors, if you can.

### Understanding Method Structure

If you want to understand a large method, line up blocks. Often indentation in long methods can make them impossible to read. You can line them up by drawing lines from the beginnings of blocks to the ends, or by commenting the



---

<https://youtu.be/0jLN-2AVIvo>

---

Open Source

## Part 1 - Extractor

<https://github.com/RainerWinkler/Moose-FAMIX-SAP-Extractor>



This repository

Search



RainerWinkler / **Moose-FAMIX-SAP-Extractor**

Open Source

## Part 2 – Improved visualization



Smalltalk  
Hub



[Home](#) [Explore](#)

[RainerWinkler](#) / **RW-Moose-Diagram**  PUBLIC



**Why Pharo?**

**Why Smalltalk?**

<http://modeling-languages.com/openponk-metamodeling-platform/>



## OpenPonk (meta)modeling platform

By **Peter Uhnak** 16/02/2017 | 8:08

Posted in **(meta)modeling, tools**

8

**Peter Uhnak**

**Pharo** makes for a great platform for **rapid prototyping** thanks to the combination of a **dead-simple language** (the grandparent of OO languages), **powerful metaprogramming** capabilities and **short feedback loops**;  
hey, you can even **write** your entire application **in a debugger**

## What about other tools?

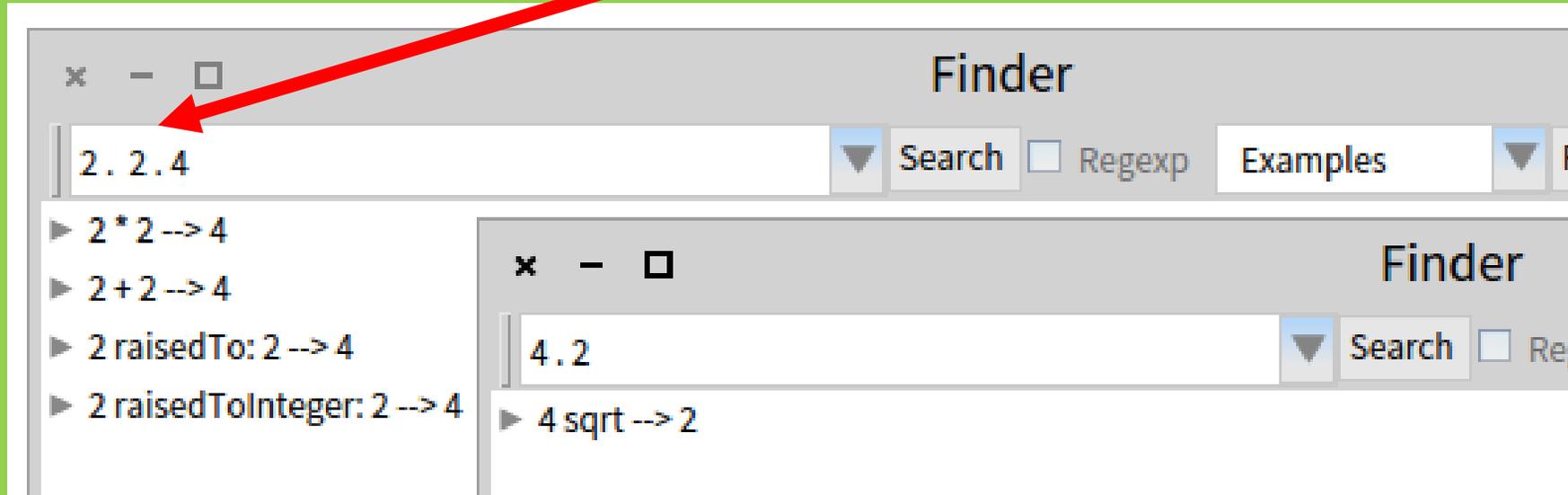
... **layers of abstractions** to facilitate all the generic tools and needs, and overall architecture we simply didn't see it as a good fit for our long-term needs.

**Information hiding, user hand-holding**, and over-engineering are certainly valuable, if you **seek a robust solution for end-users**.

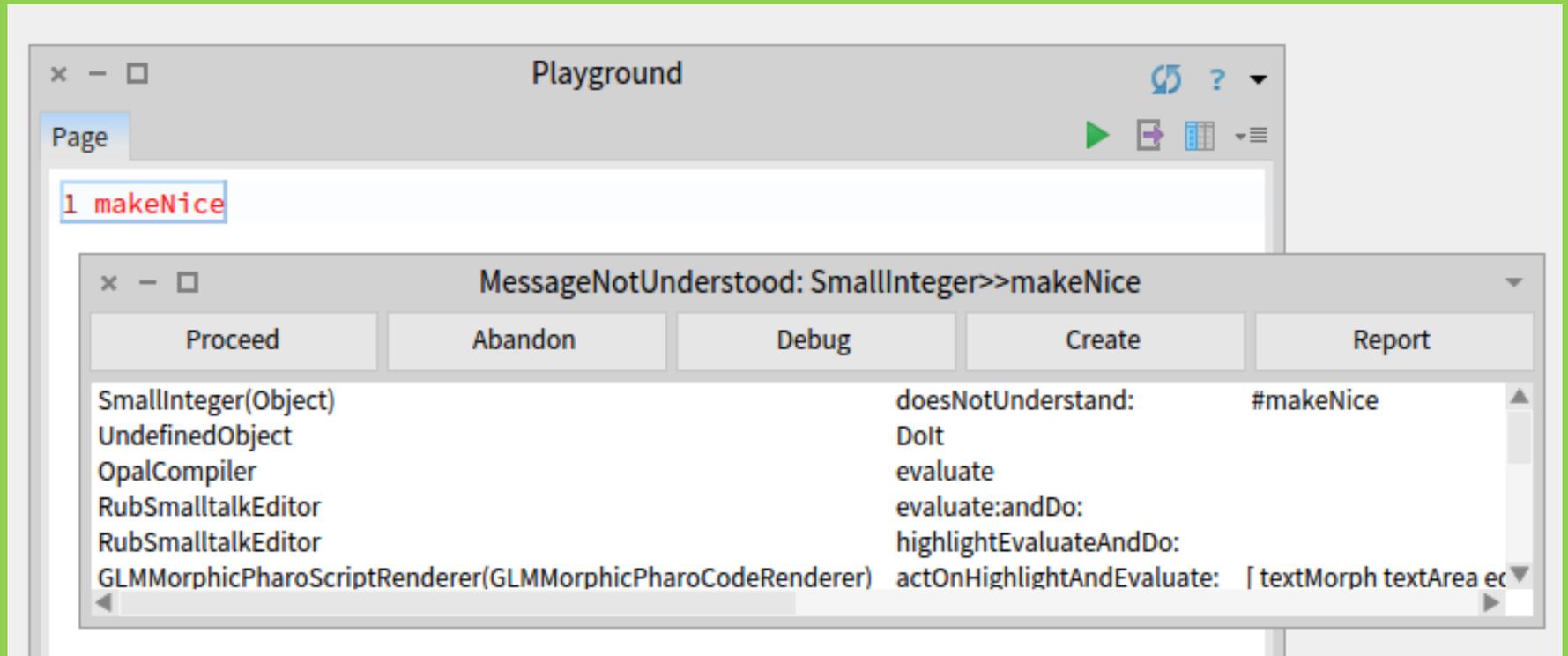
But for a poweruser **any non-essential complexity** is problematic, or **prohibits different use-cases** (in part this also relates to Java).



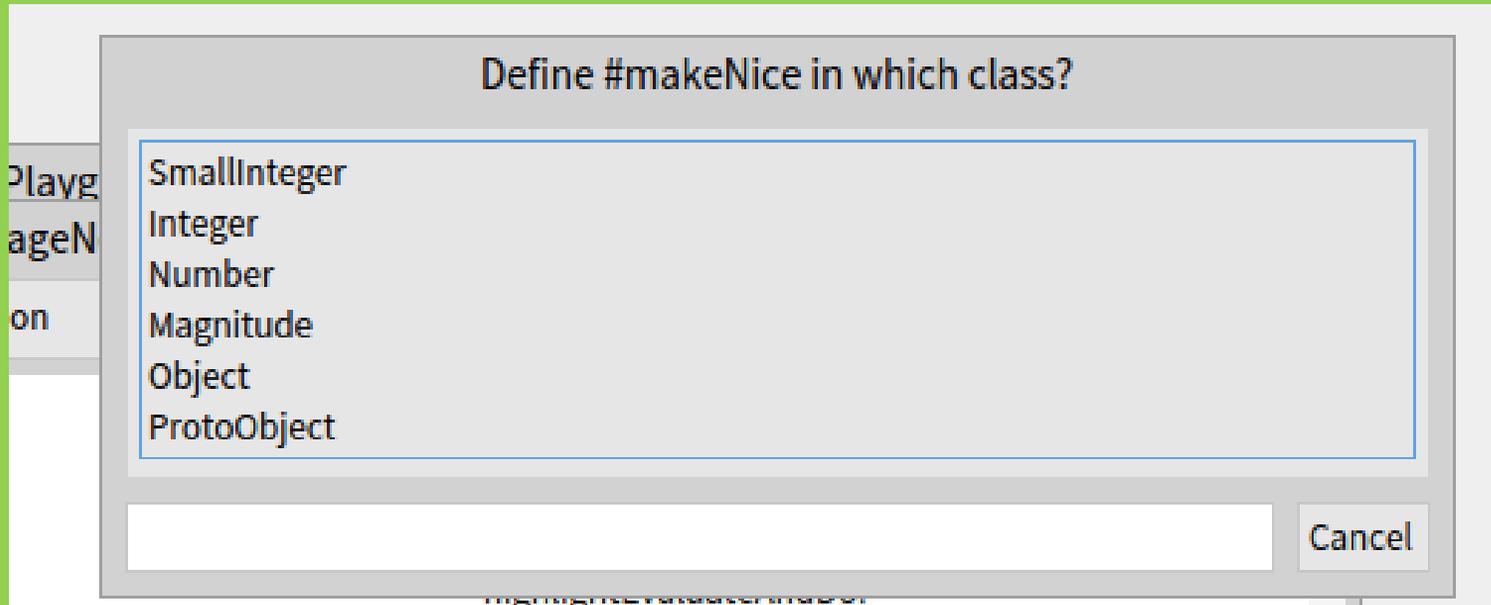
# Search for statements by example



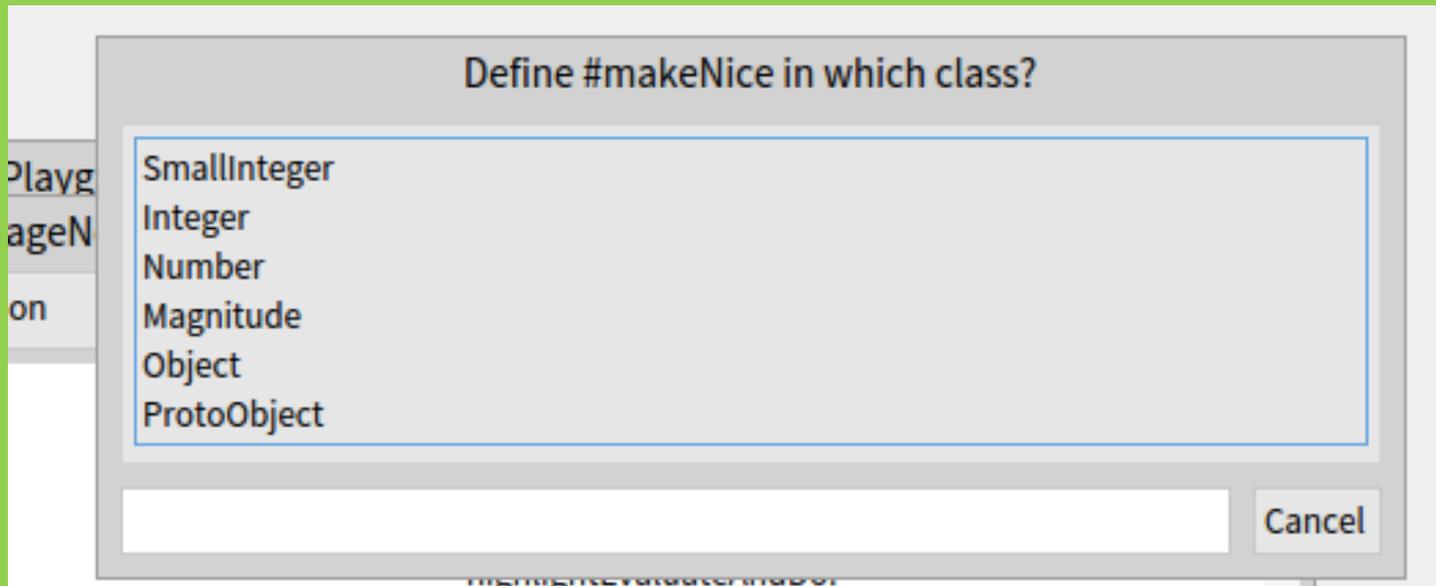
# Write code in debugger



# Write code in debugger



# Write code in debugger



# Write new code while inspecting variables

The screenshot shows a debugger window titled "MessageNotUnderstood: SmallInteger>>makeNice". The stack trace shows the following frames:

- SmallInteger(Integer) makeNice
- UndefinedObject DoIt
- OnalCompiler evaluate

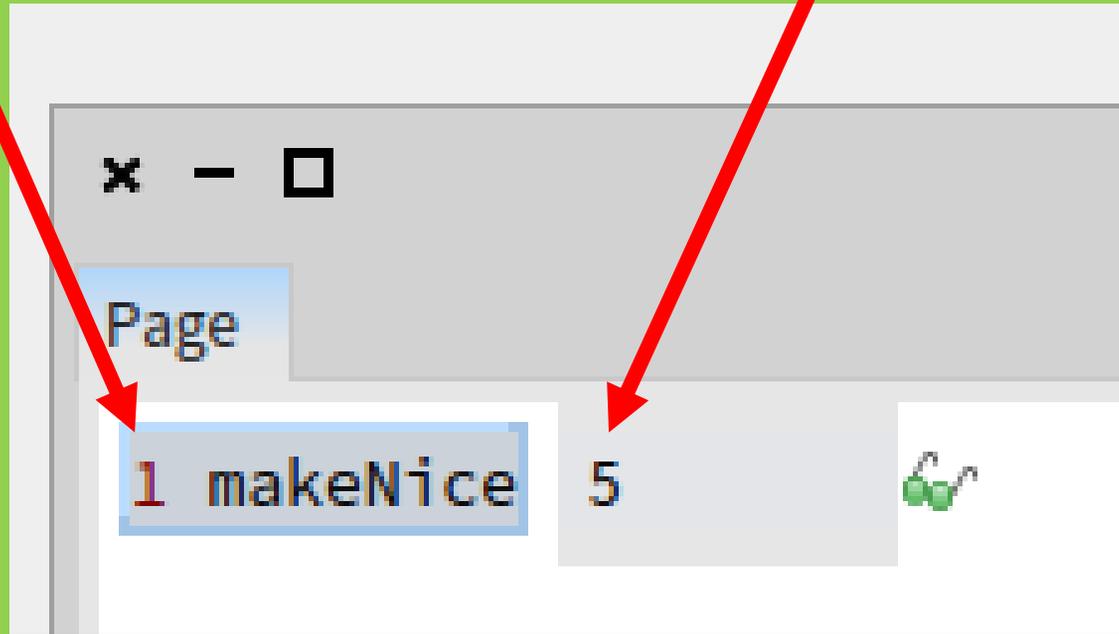
The source code editor shows the following code:

```
makeNice  
  ^5
```

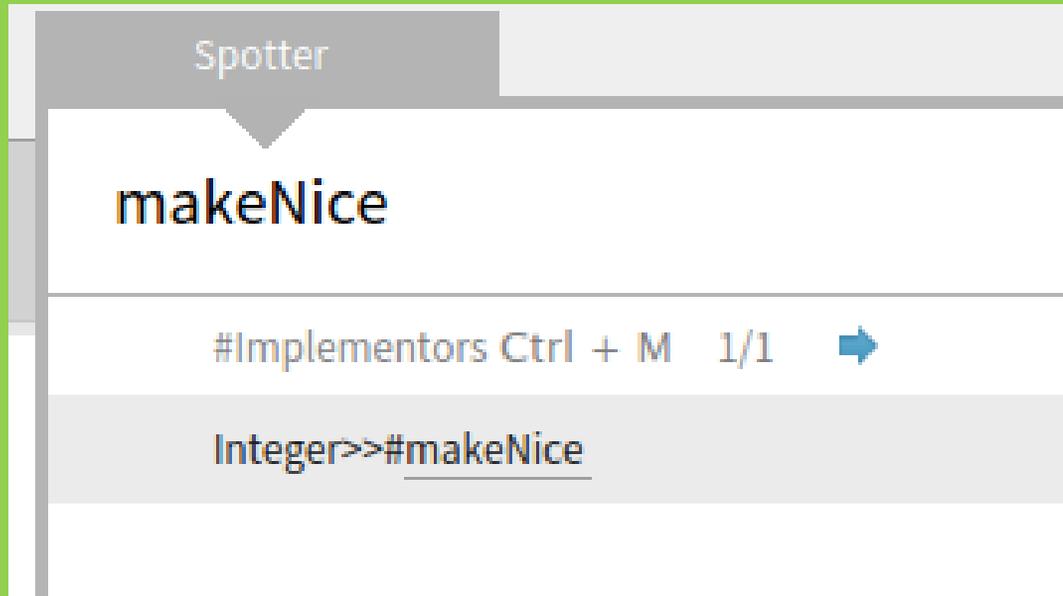
The Variables panel shows the following table:

Type	Variable	Value
implicit	self	1
implicit	thisContext	SmallInteger(Integer)>>makeNice
implicit	stack top	nil

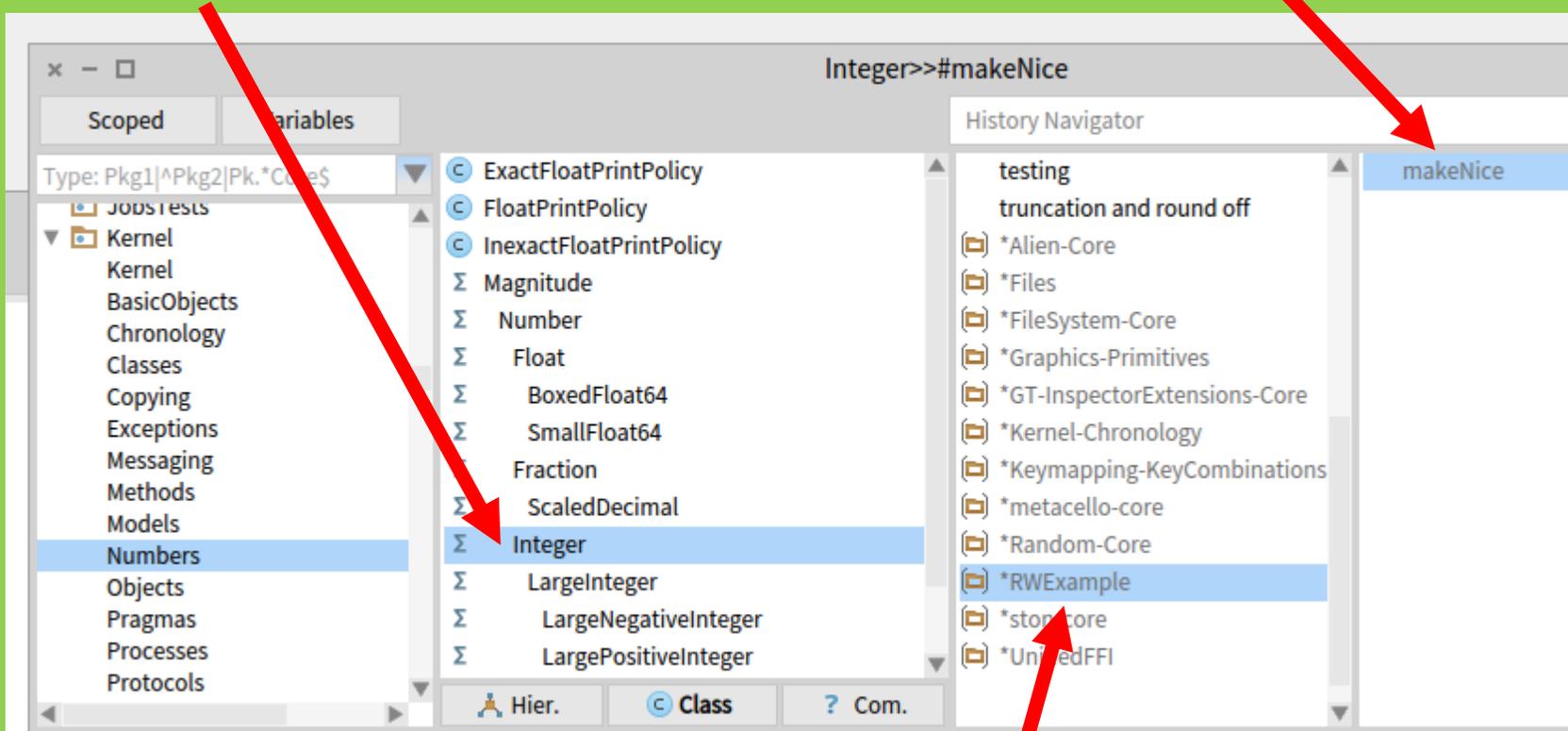
Send message „makeNice“ to 1  
1 replies now with 5



# Press Shift Enter : „Spotter“ Find things fast

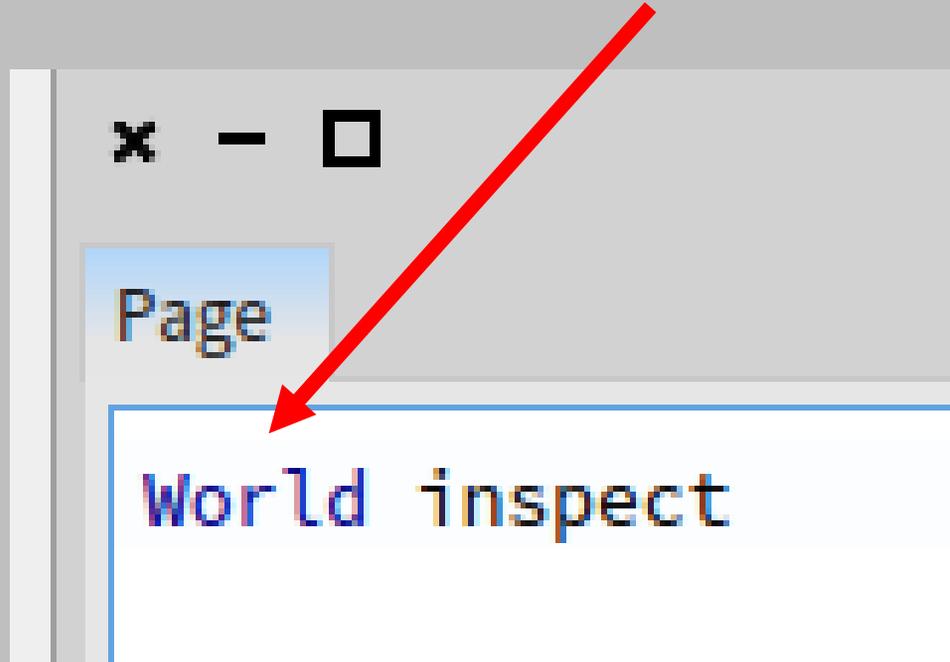


# Integer has now a new method

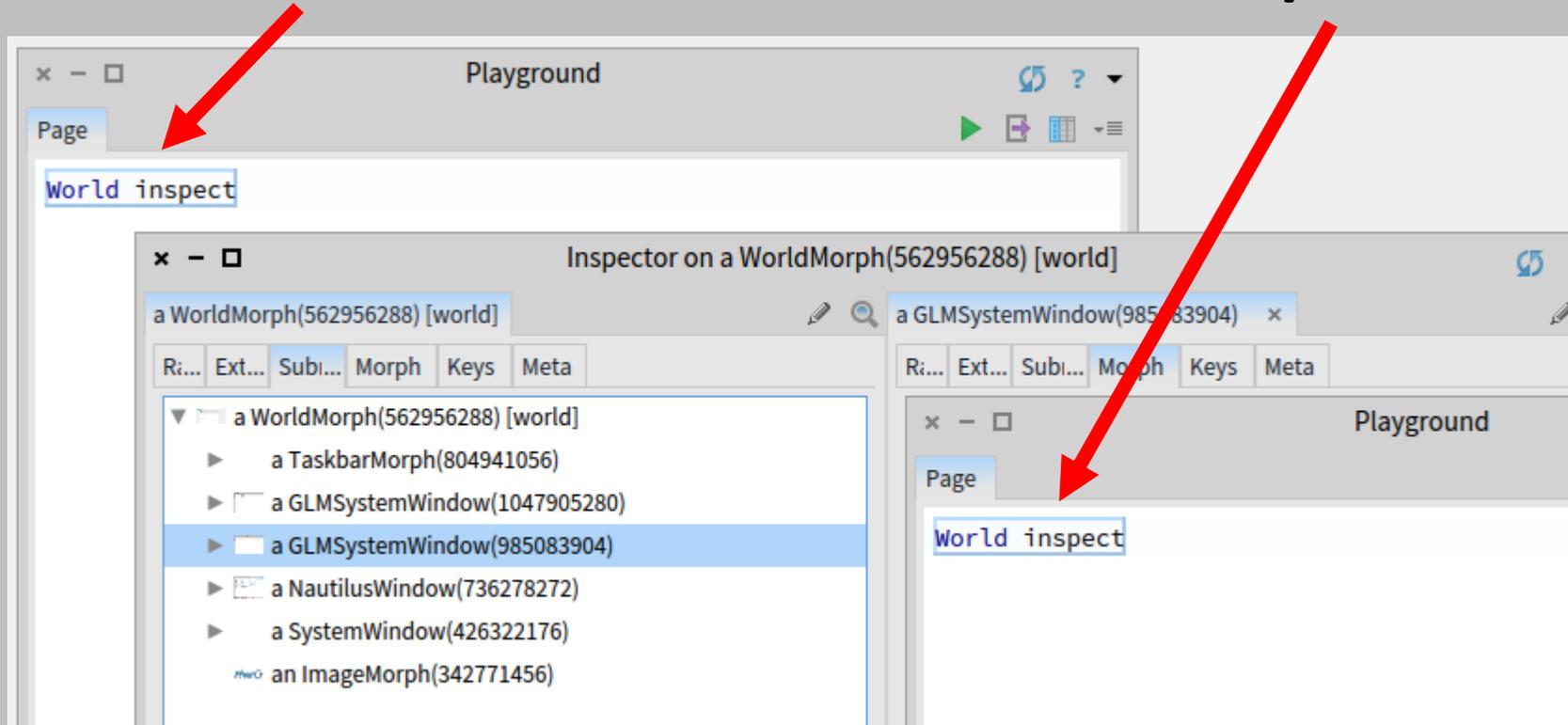


Will be delivered with package RWExample

# Inspect (near to) everything



# The window is shown in inspector



# Internal variables on Raw Tab

The screenshot shows a Playground window with a "Page" tab containing the text "World inspect". Below it is an "Inspector on a WorldMorph(562956288) [world]" window. The inspector shows a tree view of objects under "a WorldMorph(562956288) [world]":

- ▶ a TaskbarMorph(804941056)
- ▶ a GLMSystemWindow(1047905280)
- ▶ a GLMSystemWindow(985083904) (selected)
- ▶ a NautilusWindow(736278272)
- ▶ a SystemWindow(426322176)
- an ImageMorph(342771456)

The "Raw" tab is selected for the selected object, showing a table of internal variables:

Variable	Value
self	a GLMSystemWindow(98...
borderColor	Color lightGray
borderWidth	1
bounds	(254.0@216.0) corner: (85...
closeBox	a MultistateButtonMorph...
collapseBox	a MultistateButtonMorph...
collapsedFrame	nil
color	(Color r: 0.82300000000000...

A red arrow points to the "Raw" tab in the inspector.

# Send to the window a message

The image shows a development environment with two object inspectors. The left inspector, titled "Inspector on a WorldMorph(562956288) [world]", shows a tree view of objects. The right inspector, titled "Inspector on a GLMSystemWindow(985083904)", shows a table of variables and their values. Two red arrows originate from the main title: one points to the selected object in the left inspector, and the other points to the message sending area in the right inspector.

**Inspector on a WorldMorph(562956288) [world]**

- a WorldMorph(562956288) [world]
  - a TaskbarMorph(804941056)
  - a GLMSystemWindow(1047905280)
  - a GLMSystemWindow(985083904)**
  - a NautilusWindow(736278272)
  - a SystemWindow(426322176)
  - an ImageMorph(342771456)

**Inspector on a GLMSystemWindow(985083904)**

Variable	Value
self	a GLMSystemWindow(985083904)
borderColor	Color lightGray
borderWidth	1
bounds	(254.0@216.0) corner
closeBox	a MultistateButton
collapseBox	a MultistateButton
collapsedFrame	nil
<b>color</b>	(Color r: 0.8230000
embeddable	nil
expandBox	a MultistateButton
extension	a MorphExtension

Message sending area:  
a GLMSystemWindow(985083904)"  
self color: Color green

# The window is now green

The image shows a software development environment with two windows. The top window, titled "Playground", has a bright green border and a text input field containing "World inspect". Below it is an "Inspector on a WorldMorph(562956288) [world]" window. This inspector has two tabs: "a WorldMorph(562956288) [world]" and "a GLMSystemWindow(985083904)". The left pane of the inspector shows a tree view of objects, with "a GLMSystemWindow(985083904)" selected. The right pane shows a table of variables for the selected object.

Variable	Value
self	a GLMSystemWindow(985083904)
borderColor	Color lightGray
borderWidth	1
bounds	(254.0@216.0)
closeBox	a MultistateButton
collapseBox	a MultistateButton
collapsedFrame	nil
color	(Color r: 0.823)
embeddable	nil
expandBox	a MultistateButton
extension	a MorphExtension

At the bottom of the inspector, the following code is visible:

```
"a GLMSystemWindow(985083904)"  
self color: Color green
```

# Write a test for a not existing method

RWExampleTest>>#testMySignatures

Scoped Variables History Navigator

Type: Pkg1|^Pkg2|Pkg.\*Co

- ! RWExample
- RWExampleTest
- Σ Integer

Hier. Class Com.

-- all -- testMySignatures

as yet unclassified

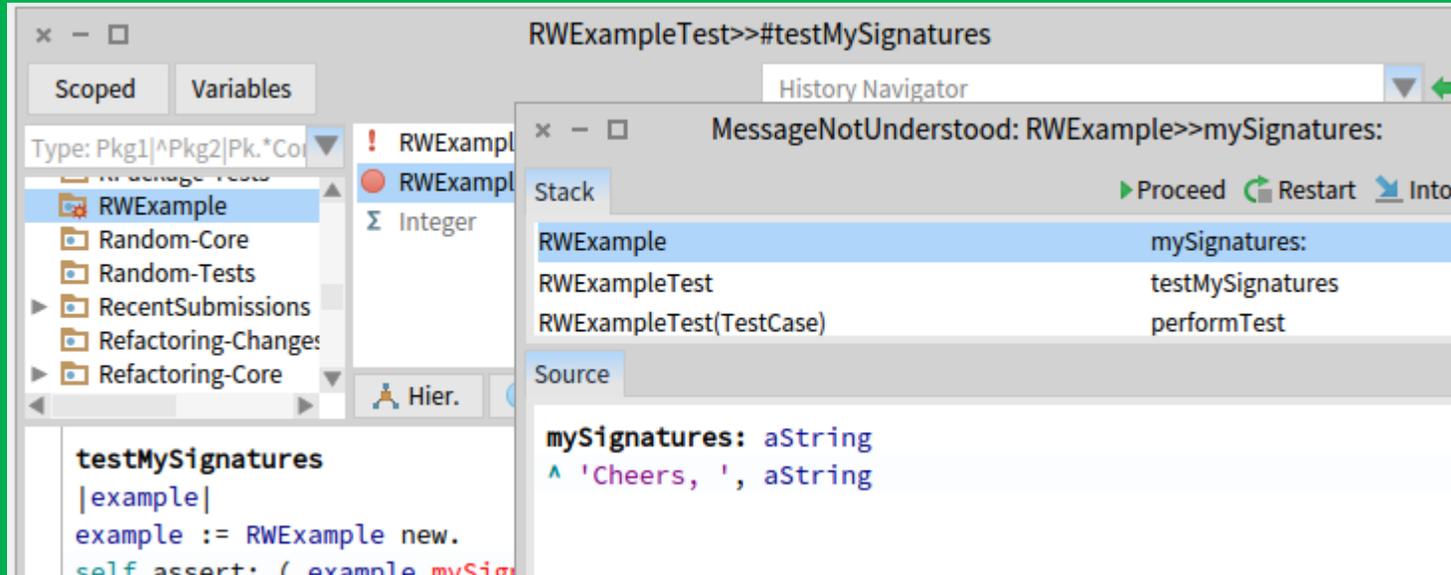
```
testMySignatures
|example|
example := RWExample new.
self assert: ( example mySignatures: 'Rainer' ) equals: 'Cheers, Rainer'
```

4/4 [36]  Format as you read W +L

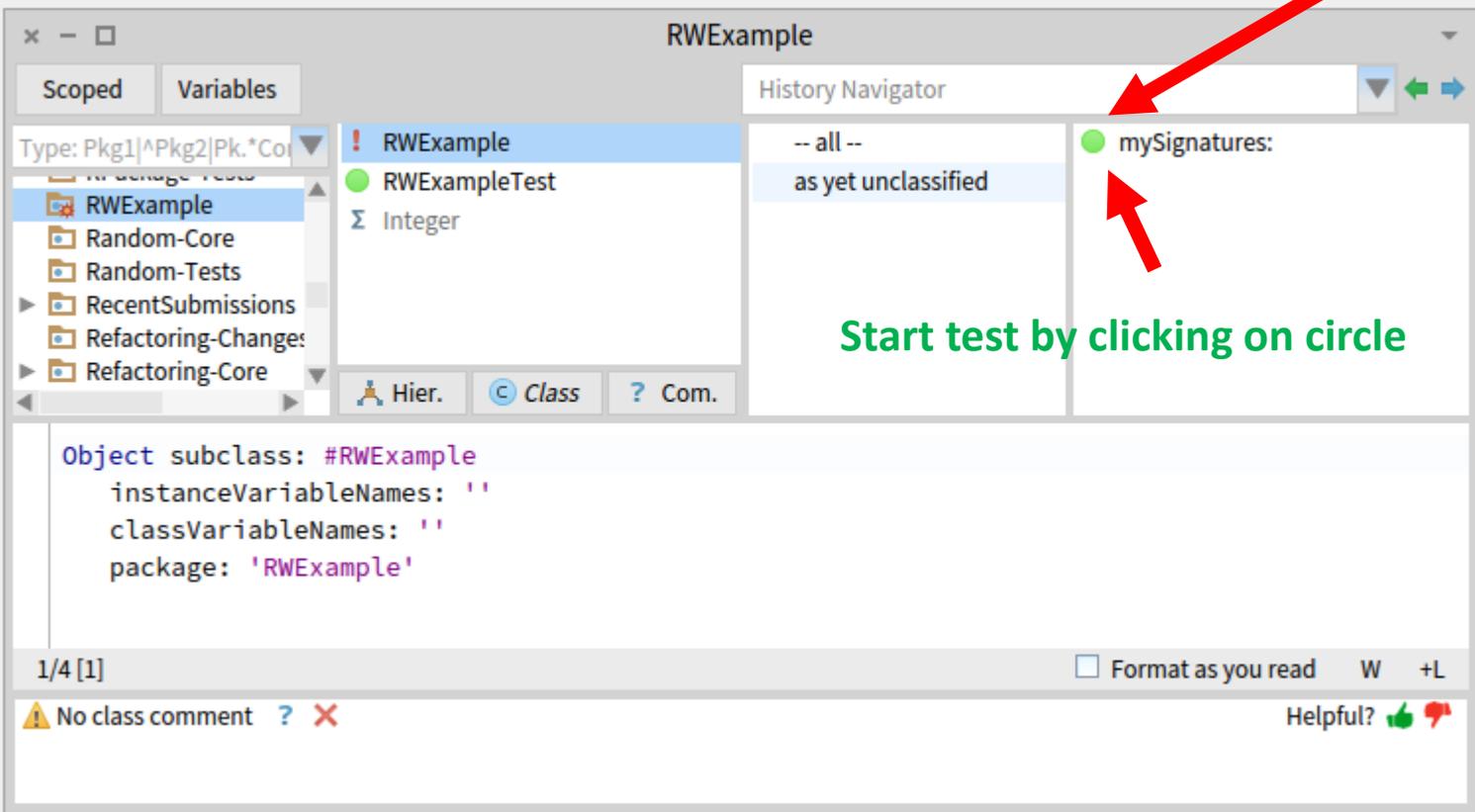
⚠ Unclassified methods ? ✖ Helpful? 👍 🇸🇬

❗ [mySignatures:] Messages sent but not implemented ? ✖ Helpful? 👍 🇸🇬

# Write program in debugger



# Method has now a passing test



The screenshot shows an IDE window titled "RWExample". The left sidebar displays a project tree with "RWExample" selected. The main editor area shows the "History Navigator" with a list of items: "-- all --" and "as yet unclassified". A green circle next to "mySignatures:" is highlighted, with a red arrow pointing to it. Below the history navigator, the code for the "mySignatures:" method is displayed, showing it is a subclass of "#RWExample" with instance and class variable names and a package name of "RWExample". The status bar at the bottom indicates "1/4 [1]" and "Format as you read" is checked. A warning icon and "No class comment" message are visible in the bottom left, and "Helpful?" with thumbs up/down icons are in the bottom right.

Start test by clicking on circle

Ask me:

Rainer Winkler

Have fun 😊