A TASTE OF

SMALLTALK

SMALLTALK IS AN OBJECT-ORIENTED, DYNAMICALLY TYPED, REFLECTIVE PROGRAMMING LANGUAGE.

Wikipedia

Smalltalk

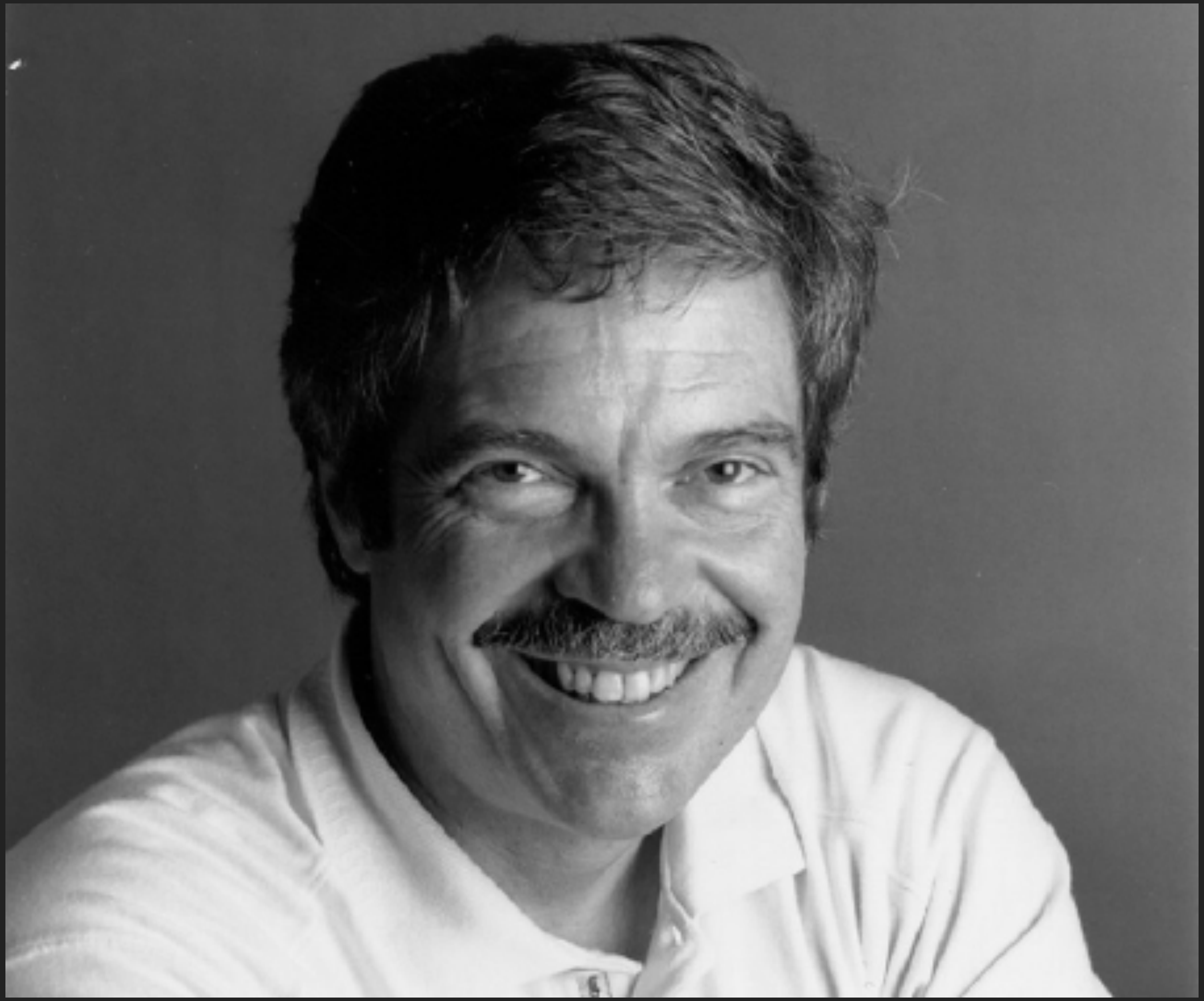
is an

Interactive Programming Environment

Designed by Alan Kay

in the 1970s

at Xerox PARC



Turing Award in 2003

For pioneering many of the ideas at the root of contemporary object-oriented programming languages, leading the team that developed Smalltalk, and for fundamental contributions to personal computing.

Smalltalk is responsible for numerous innovations

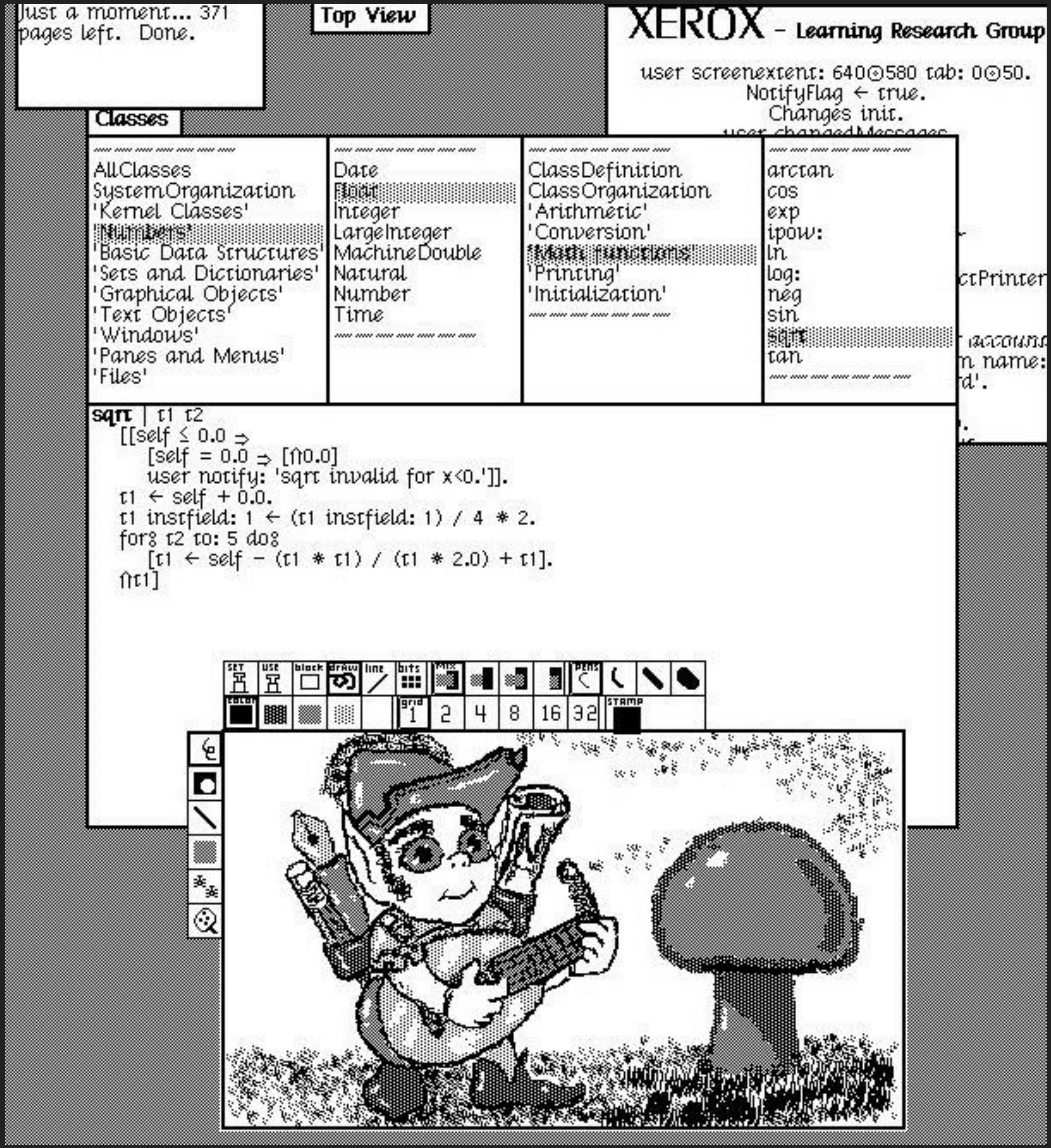
Line Oriented Graphics

Bitmapped Graphics!

DEC VT100 Terminal (1978)

Xerox Alto running Smalltalk (1976)

Early Smalltalk Environment (1970s)



Just a moment... 371
pages left. Done.

Top View

XEROX - Learning Research Group

user screenextent: 640⊙580 tab: 0⊙50.
NotifyFlag ← true.
Changes init.

Classes

AllClasses	Date	ClassDefinition	arctan
SystemOrganization	Float	ClassOrganization	cos
'Kernel Classes'	Integer	'Arithmetic'	exp
'Numbers'	LargeInteger	'Conversion'	ipow:
'Basic Data Structures'	MachineDouble	'Math functions'	ln
'Sets and Dictionaries'	Natural	'Printing'	log:
'Graphical Objects'	Number	'Initialization'	neg
'Text Objects'	Time		sin
'Windows'			sqrt
'Panes and Menus'			tan
'Files'			

```
sqrt | t1 t2
   [[self ≤ 0.0 ⇒
      [self = 0.0 ⇒ [⇑0.0]
      user notify: 'sqrt invalid for x<0.']].
   t1 ← self + 0.0.
   t1 instfield: 1 ← (t1 instfield: 1) / 4 * 2.
   for§ t2 to: 5 do§
      [t1 ← self - (t1 * t1) / (t1 * 2.0) + t1].
   ⇑t1]
```

Apple Macintosh (1984)

Xerox Alto (1970s)

Pure OOP

GUIs

JIT

TDD

MVC

Mouse

INTRODUCTION

SMALLTALK IS

- Object Oriented
- Dynamically Typed, Reflective
- Compiled (to Smalltalk ByteCode)
- Interpreted (ByteCode is interpreted by a Smalltalk VM)
- Cross-Platform
- Image Based
- Interactive

WHAT DOES IMAGE BASED MEAN?

WHAT DOES IMAGE BASED MEAN?

Virtual Machine + **Linux OS Image**

WHAT DOES IMAGE BASED MEAN?

Virtual Machine + **Linux OS Image**

OS Image contains a *snapshot* of CPU, RAM, and the state of virtual devices like hard disks

WHAT DOES IMAGE BASED MEAN?

Virtual Machine + **Linux OS Image**

OS Image contains a *snapshot* of CPU, RAM, and the state of virtual devices like hard disks

Allowing you to Pause & Resume

SMALLTALK IS IMAGE BASED

- Smalltalk VM (Smalltalk ByteCode Interpreter)
- Smalltalk Image (Persistent State of all Smalltalk Objects)
- On resuming a Smalltalk Image, its objects come alive

MULTIPLE IMPLEMENTATIONS

- Squeak
- Pharo
- GNU Smalltalk
- Amber
- … and many others …

MULTIPLE IMPLEMENTATIONS

- Squeak
- Pharo
- GNU Smalltalk
- Amber
- … and many others …

The Smalltalk Environment

THE SMALLTALK ENVIRONMENT

- Entirely made up of objects
- Working in Smalltalk essentially means modifying the environment (by introducing new objects, modifying existing objects)
- You modify the environment by interacting with the objects (tools) available in the environment

THE SMALLTALK ENVIRONMENT

- Code is NOT written or stored in files
- You develop code in tools offered by the environment (IDE like tools such as Class Browsers, Debuggers, etc)
- Your code becomes part of the environment
- You incrementally modify the environment until your Software is fully built
- Code, Class Browsers, Compilers, Debuggers are all *objects that co-exist in the environment!*

(lets dive in)

The Smalltalk Language

A VERY SMALL LANGUAGE

- 6 keywords
- Syntax for numbers, strings, arrays, other literals
- Syntax for code blocks (anonymous functions)
- 3 forms of sending messages to objects (method calls)
- Miscellaneous syntax (variable declaration, assignment, comments, etc)

THE SIX KEYWORDS (PSEUDO VARIABLES)

true	false	nil
self	super	thisContext

THE SIX KEYWORDS (PSEUDO VARIABLES)

true	false	nil (like null in Java)
self	super	thisContext

THE SIX KEYWORDS (PSEUDO VARIABLES)

true	false	nil (like null in Java)
self (like this in Java)	super	thisContext

LITERAL SYNTAX

Numbers	123, 2.5434e10, 2r1010
Character Literals	prefixed with a $ e.g. $% is same as '%' in Java
String Literals	'single quoted, multi-line'
Symbols (String Constants)	#xyz
Constant Arrays	#(15 $y 'abc' #xyz)
Dynamic Arrays	{ 12 + 3 . 5 squared . 'abc' length }

MESSAGE SYNTAX (METHOD INNOVATION SYNTAX)

	Smalltalk	Javascript(*ish*)
Unary	123 squared 'abc' size	123.squared() 'abc'.size()
Binary	5 + 10 (operators are messages)	5.plus(10)

Keyword	text indexOf: $a
	text findString: 'abc' startingAt: 10
		text.indexOf('a')
		text.findStringStartingAt('abc', 10)

MESSAGE PRECEDENCE

Unary > Binary > Keyword

MESSAGE PRECEDENCE

Unary > Binary > Keyword

```
calculator display: ((2 squared) + (3 squared))
```

MESSAGE PRECEDENCE

Unary > Binary > Keyword

```
calculator display: 2 squared + 3 squared
```

LANGUAGE SYNTAX

Comments	"double quoted, multi-line"
Variable Declaration	| x y z |
Expression Grouping	(5 + 4) * (12 / 4)
Blocks (Anonymous Functions)	[:x :y | (x + y) / 2]
Assignment	x := 123
Statement Separator	fullstop . (like ; in Java)
Method Return	^ answer
Message Cascade	car makeRight; drive; makeLeft

that covers **90%** of the smalltalk language...

SYNTAX IN A POSTCARD

```
exampleWithNumber: x

"A method that illustrates every part of Smalltalk method syntax
except primitives. It has unary, binary, and keyword messages,
declares arguments and temporaries, accesses a global variable
(but not an instance variable), uses literals (array, character,
symbol, string, integer, float), uses the pseudo variables
true false, nil, self, and super, and has sequence, assignment,
return and cascade. It has both zero argument and one argument blocks."

    |y|
    true & false not & (nil isNil) ifFalse: [self halt].
    y := self size + super size.
    #($a #a "a" 1 1.0)
        do: [:each | Transcript show: (each class name);
                                show: ' '].
    ^ x < y
```

KEYWORD MESSAGES

```
text findString: ‘abc’ startingAt: 10
```

vs

```
text.findStringStartingAt(‘abc’, 10)
```

KEYWORD MESSAGES

```
text findString: ‘abc’ startingAt: 10
```

Reads like a sentence

No need to guess the order of parameters

KEYWORD MESSAGES

```
text findString: 'abc' startingAt: 10
```

Send text the message #findString:startingAt: with arguments 'abc' and 10

KEYWORD MESSAGES

```
text findString: 'abc' startingAt: 10
```

Send text the message #findString:startingAt: with arguments 'abc' and 10

called a selector

BLOCKS

```
[ Transcript show: 'Hello World!' ]
```

BLOCKS

```
[ Transcript show: ‘Hello World!’ ] value
```

Sending the #value message
to a block causes it to evaluate

BLOCKS

```
[ :a : b | ( a + b ) / 2 ]
```

BLOCKS

```
[ :a : b | ( a + b ) / 2 ] value: 10 value: 20
```

Send the #value:value: message along with the parameters to the block

DID WE COVER ALL SYNTAX?

- How do we define classes?
- How do we create new object instances?
- What about syntax for if / else statements?
- What about loops?
- What about throwing and handling exceptions?
- What about mathematical operators like + - * / ?

DID WE COVER ALL SYNTAX?

- How do we define classes?
- How do we create new object instances?
- What about syntax for if / else statements?
- What about loops?
- What about throwing and handling exceptions?
- What about mathematical operators like + - * / ?

Smalltalk does not define syntax for these things because…

EVERYTHING ELSE IS IMPLEMENTED AS MESSAGES

- All operators (+ - * / & | …)
- Conditionals
- Loops
- Exception Handling
- No special class keyword (to create classes)
- No special new keyword (to create object instances)

A LANGUAGE SHOULD BE DESIGNED AROUND A POWERFUL METAPHOR THAT CAN BE UNIFORMLY APPLIED IN ALL AREAS.

Dan Ingalls

THE SMALLTALK WAY

- Everything is an *object*
- Everything happens through objects exchanging *messages*

IF / ELSE

```
aNumber even ifTrue:  [ Transcript show: 'x is even' ]
             ifFalse: [ Transcript show: 'x is odd' ].
```

IF / ELSE

```
aNumber even ifTrue:  [ Transcript show: 'x is even' ]
             ifFalse: [ Transcript show: 'x is odd' ].
```

IF / ELSE

```
aNumber even ifTrue:  [ Transcript show: 'x is even' ]
             ifFalse: [ Transcript show: 'x is odd' ].
```

#ifTrue:ifFalse: is defined in the Boolean class

LOOPS

```
x := 1.
[ x <= 10 ] whileTrue: [
  Transcript showln: x.
  x := x + 1
].
```

LOOPS

```
x := 1.
[ x <= 10 ] whileTrue: [
  Transcript showln: x.
  x := x + 1
].
```

LOOPS

```
x := 1.
[ x <= 10 ] whileTrue: [
  Transcript showln: x.
  x := x + 1
].
```

#whileTrue: is defined in the BlockClosure class

EXCEPTION HANDLING

```
[ 100 / 0 ]
  on: ZeroDivide
  do: [
    Transcript showln: 'Error: divide by 0'
  ]
```

EXCEPTION HANDLING

```
[ 100 / 0 ]
  on: ZeroDivide
  do: [
    Transcript showln: 'Error: divide by 0'
  ]
```

EXCEPTION HANDLING

```
[ 100 / 0 ]
  on: ZeroDivide
  do: [
    Transcript showln: 'Error: divide by 0'
  ]
```

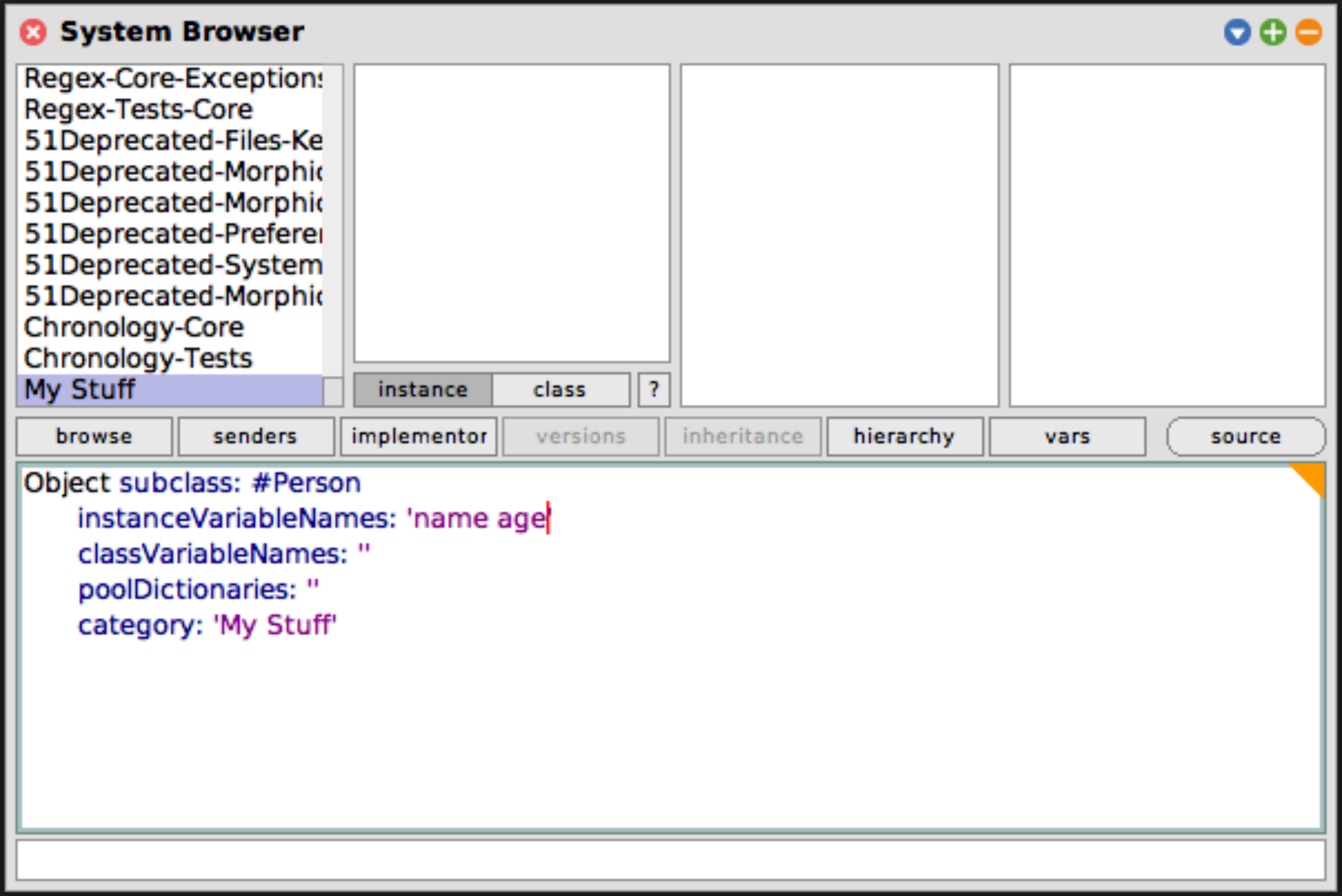
#on:do: is defined in the BlockClosure class

CREATING A CLASS

CREATING A CLASS

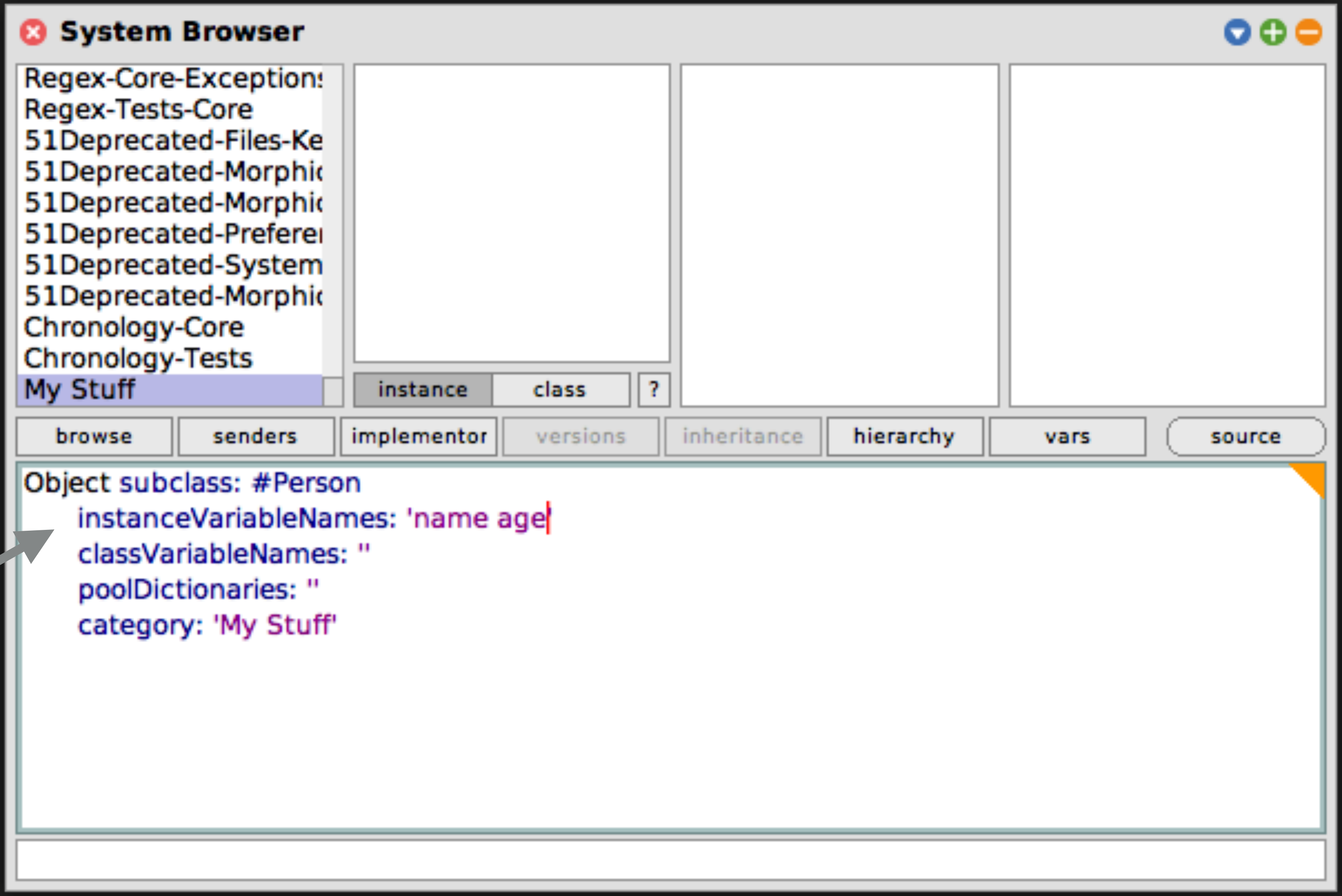
(DEMO)

CREATING A CLASS



```
Object subclass: #Person
    instanceVariableNames: 'name age
    classVariableNames: ''
    poolDictionaries: ''
    category: 'My Stuff'
```

CREATING A CLASS



```
Object subclass: #Person
	instanceVariableNames: 'name age'
	classVariableNames: ''
	poolDictionaries: ''
	category: 'My Stuff'
```

Class Template

CREATING A CLASS

```
Object subclass: #Person
  instanceVariableNames: 'name age'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'My Stuff'
```

Whats really happening here?

CREATING A CLASS

```
Object subclass: #Person
  instanceVariableNames: 'name age'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'My Stuff'
```

Does this look like sending a message?

CREATING A CLASS

Send **Object** a message with a really long name

#subclass:instanceVariableNames:classVariable**Names:poolDictionaries:category:**

We just ask a class to create a subclass of itself!

INSTANTIATING AN OBJECT

```
Person new
```

INSTANTIATING AN OBJECT

```
Person new
```

INSTANTIATING AN OBJECT

```
Person new
```

Sending the #new message to a class object causes an instance to be created

ALIVENESS

IS EVERYTHING REALLY AN OBJECT?

- Is the Class Browser an object?
- Is the maximize button on that window an object?
- Is the Scrollbar on that window an object?
- Can I make that window spin a 360?
- Is the Debugger an object?
- Can I really *interact* with them?

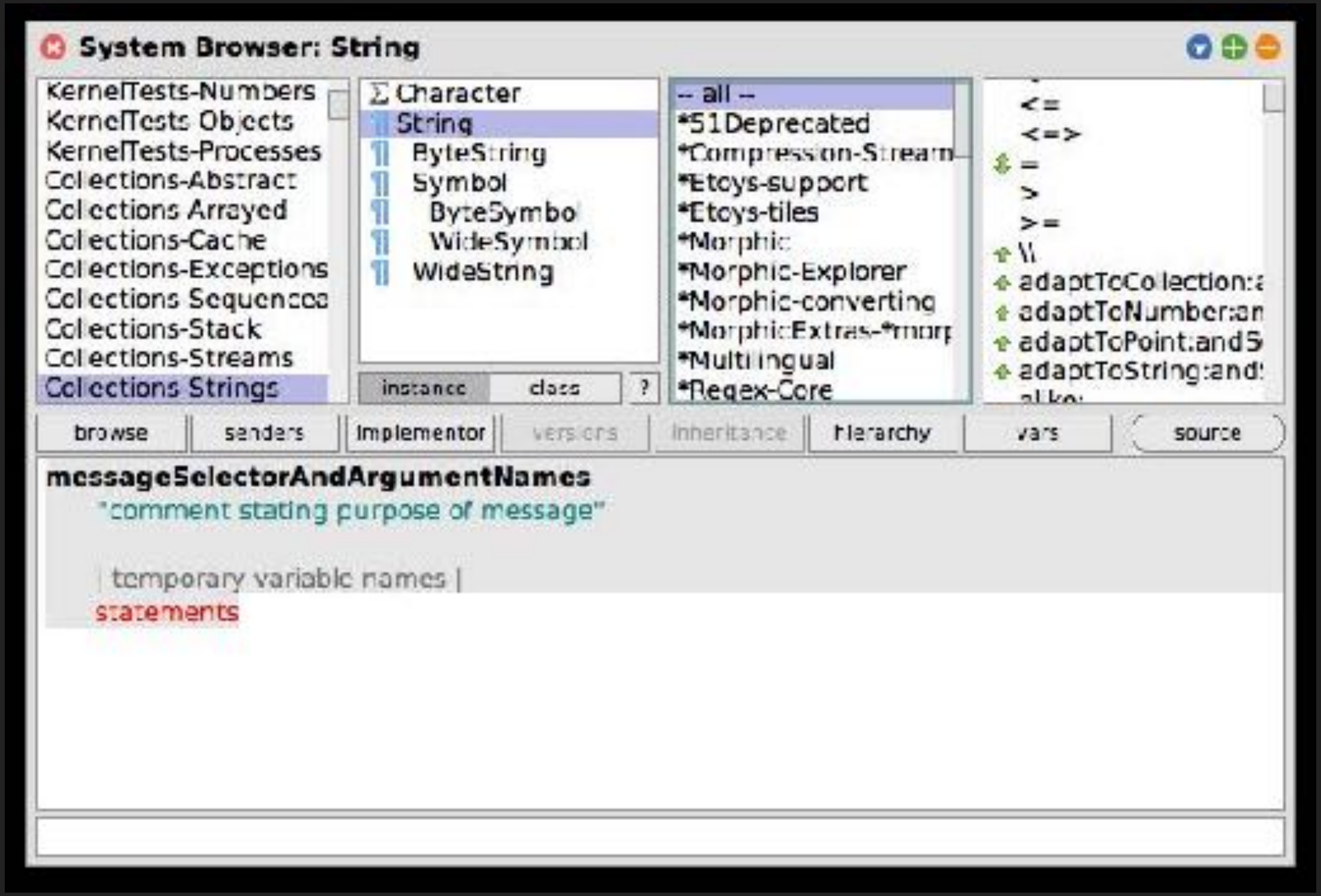
(lets find out)

HOW IS THE USER INTERFACE SO MALLEABLE?

- Every pixel rendered as color values into a screen sized buffer
- Buffer rendered to screen via host platform APIs
- Every pixel of every graphic shape (including fonts, windows, lines, all shapes) rendered using code written in Smalltalk
- Entire GUI + Event System written in Smalltalk
- All tools such as Class Browser, Debugger, etc render their appearance into that buffer (via the Morphic graphics framework)
- Everything is open to modification!

Can I modify the IDE?

CLASS BROWSER

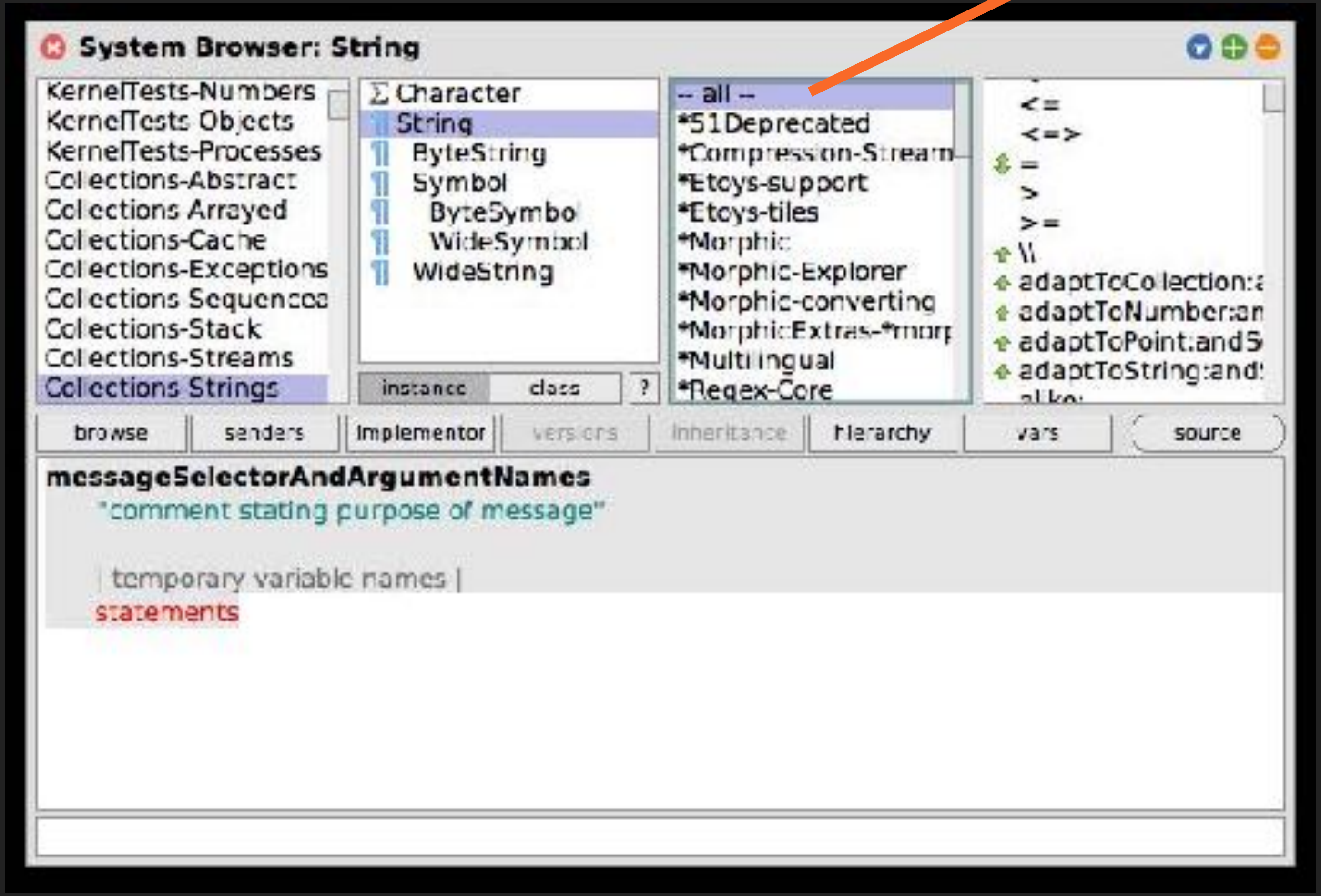


```
messageSelectorAndArgumentNames
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

CLASS BROWSER

–all--
pseudo protocol

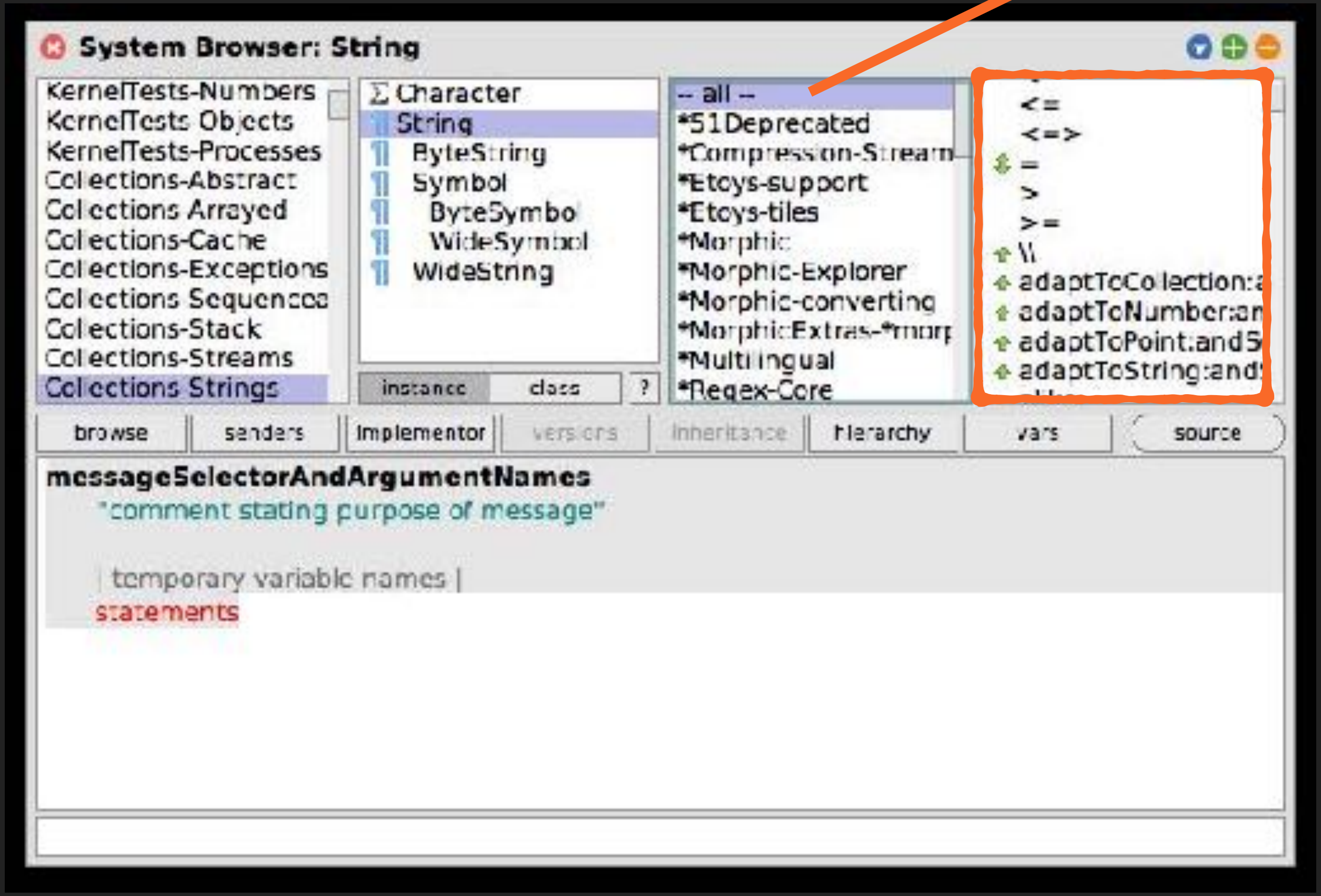


```
messageSelectorAndArgumentNames
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

CLASS BROWSER

—all--
pseudo protocol



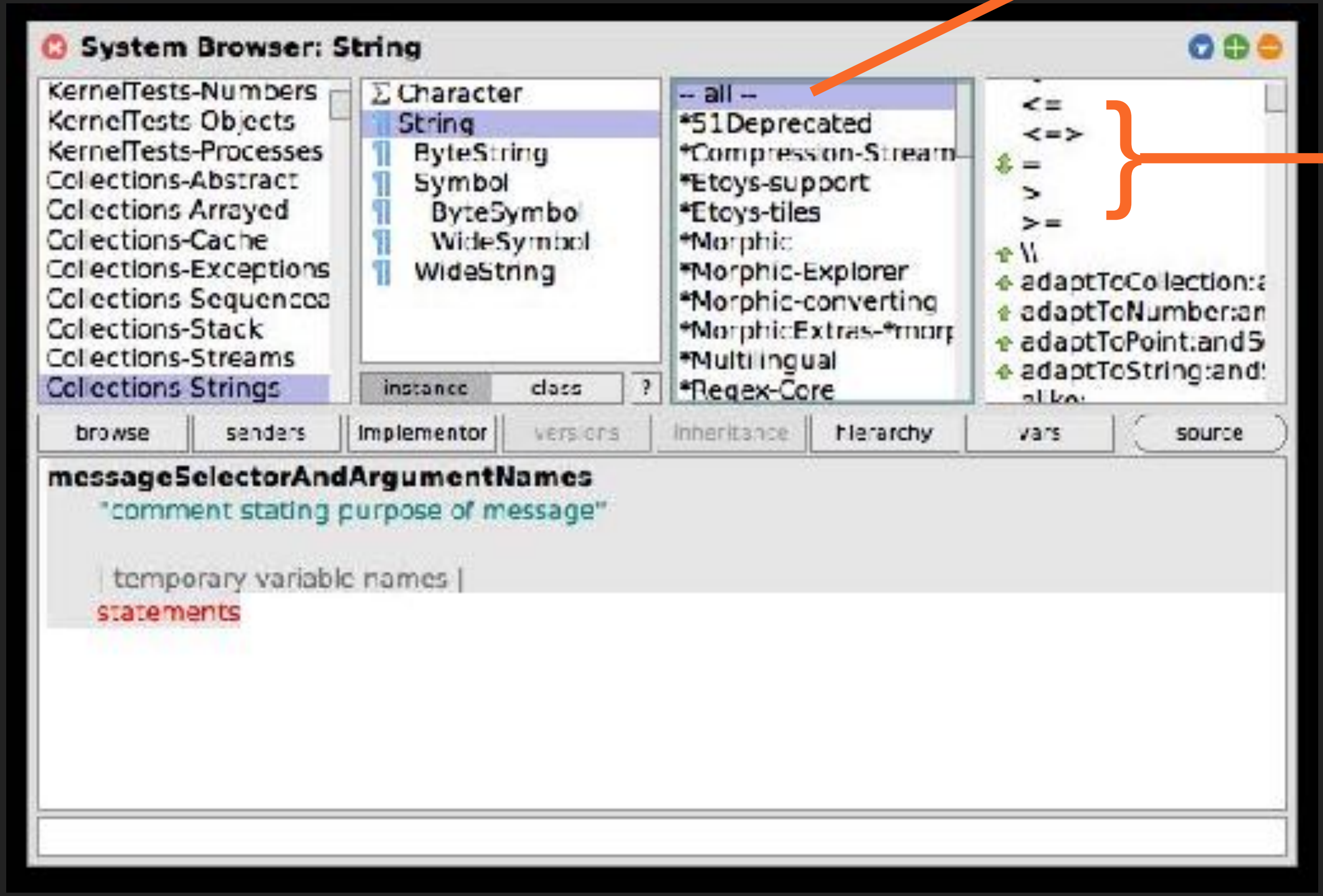
```
messageSelectorAndArgumentNames
	"comment stating purpose of message"

	| temporary variable names |
	statements
```

clicking it shows **all** the methods
defined on the Class across all protocols

CLASS BROWSER

—all--
pseudo protocol



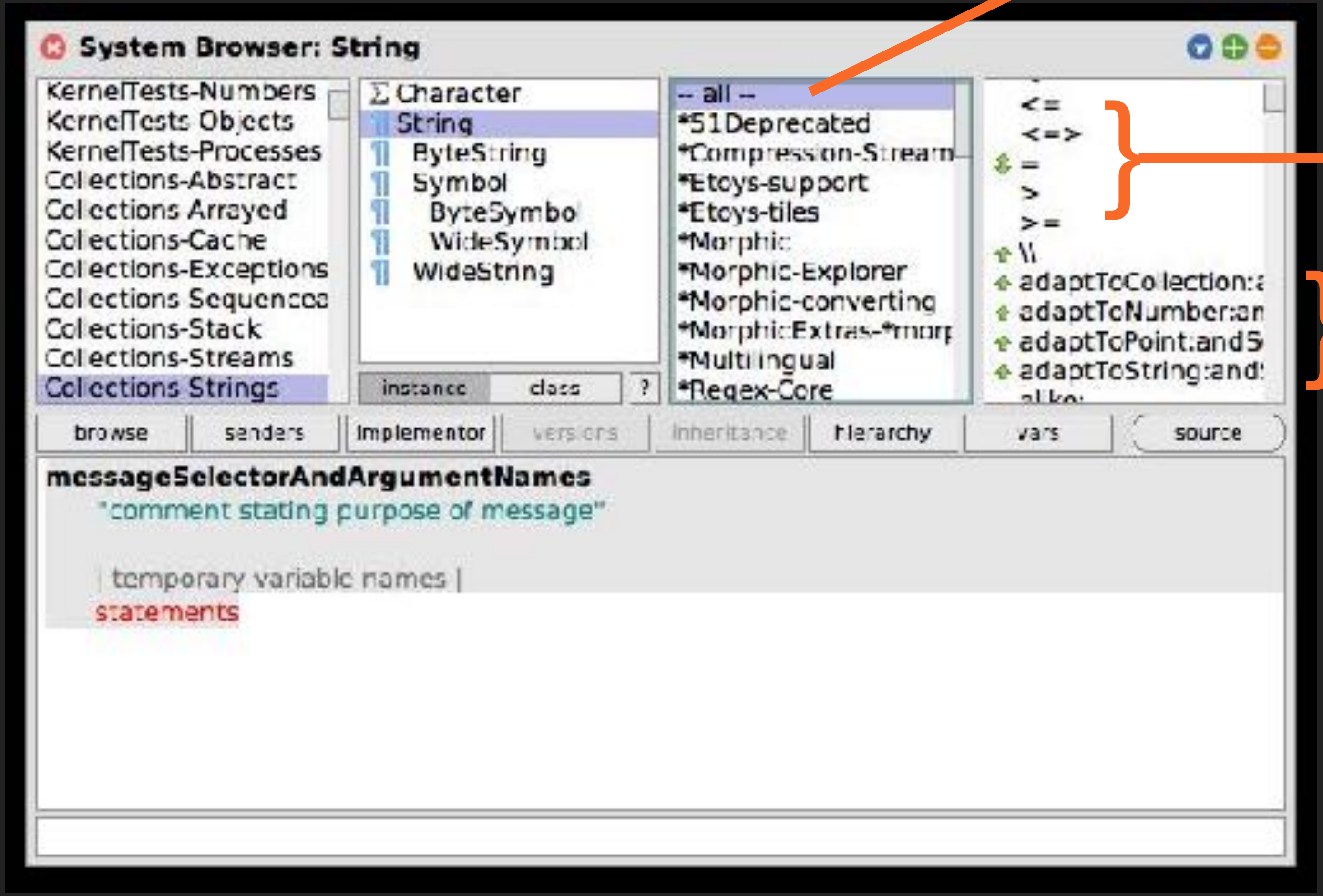
```
messageSelectorAndArgumentNames
	"comment stating purpose of message"

	| temporary variable names |
	statements
```

binary
messages

clicking it shows **all** the methods
defined on the Class across all protocols

CLASS BROWSER



```
messageSelectorAndArgumentNames
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

–all--
pseudo protocol

binary messages

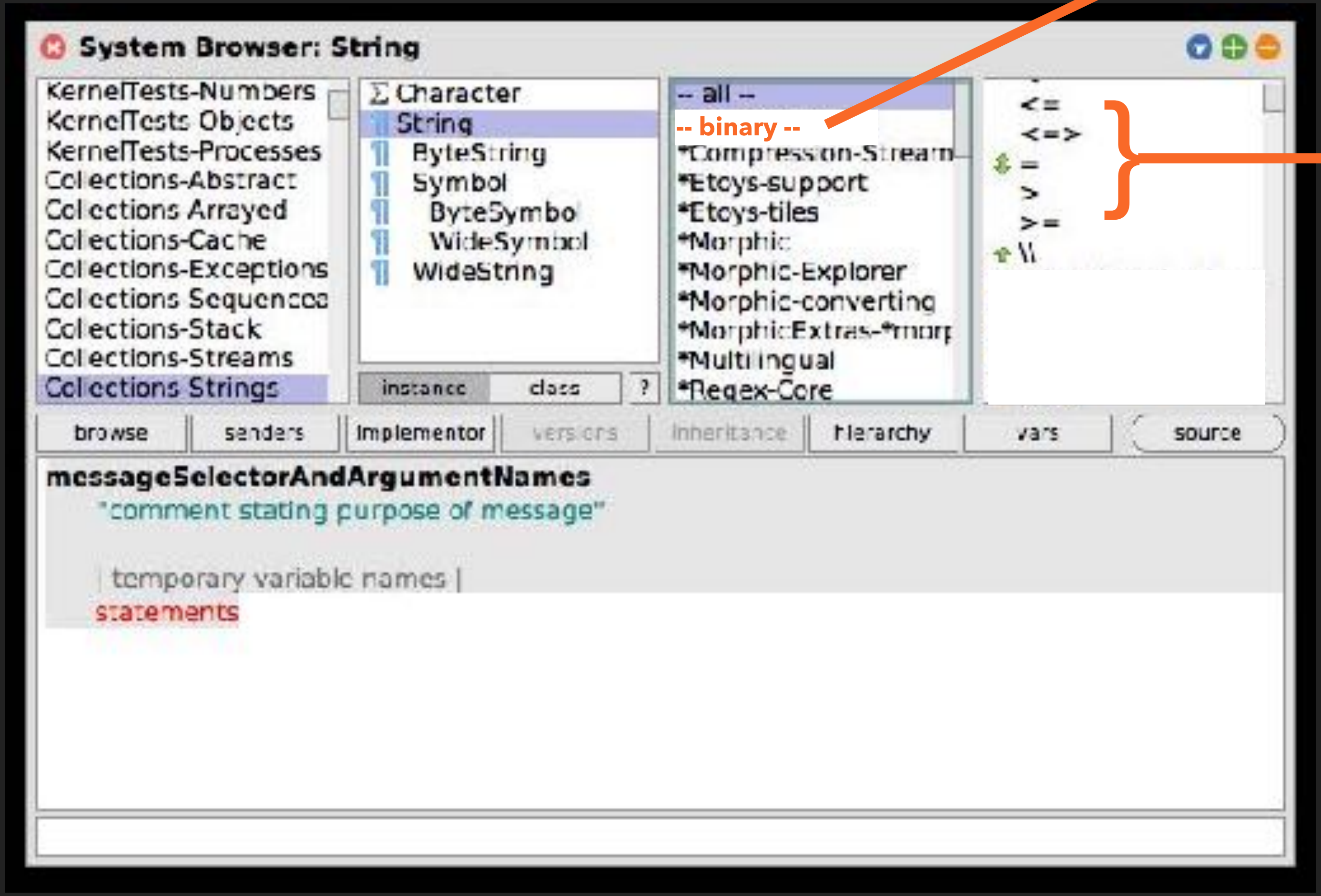
keyword messages

clicking it shows **all** the methods defined on the Class across all protocols

LETS MAKE A NEW CLASS BROWSER

- lets subclass the Browser class
- then, modify the protocol list
- and add a new pseudo protocol called –binary–
- clicking on –binary– should should list only binary messages (i.e. only symbolic messages like + * & <=)
- we'll use the method finder to ask Smalltalk how to do things

WHAT WE WANT



System Browser: String

KernelTests-Numbers
KernelTests-Objects
KernelTests-Processes
Collections-Abstract
Collections-Arrayed
Collections-Cache
Collections-Exceptions
Collections-Sequences
Collections-Stack
Collections-Streams
Collections-Strings

Σ Character
String
ByteString
Symbol
ByteSymbol
WideSymbol
WideString

-- all --
-- binary --
*Compression-Stream
*Etoys-support
*Etoys-tiles
*Morphic
*Morphic-Explorer
*Morphic-converting
*MorphicExtras-*morp
*Multilingual
*Regex-Core

<=
<=>
=
>
>=
\\

```
messageSelectorAndArgumentNames
	"comment stating purpose of message"

	| temporary variable names |
	statements
```

new –binary-- pseudo protocol

only binary messages should be listed

(lets do this)

DDD
(Debugger Driven Development)

DEBUGGER DRIVEN DEVELOPMENT

- You can evaluate any piece of code
- During evaluation, Smalltalk will ask you what you meant and will pop open a debugger if necessary!
- Allowing you to create classes, methods, instance variables, etc on the fly
- Once you complete all editing, execution simply proceeds as if everything was already in place!

(lets try this out)

WHY DID SMALLTALK POP OPEN A DEBUGGER?

- When an object does not understand a message, the VM sends it the #doesNotUnderstand message along with the original message, arguments
- Classes are free to implement this message in any manner
- The Object class supplies a default implementation that pops open a debugger!

SMALLTALK IS A RECURSION ON THE NOTION OF COMPUTER ITSELF. *INSTEAD OF DIVIDING "COMPUTER STUFF" INTO THINGS EACH LESS STRONG THAN THE WHOLE – LIKE DATA STRUCTURES, PROCEDURES, AND FUNCTIONS WHICH ARE THE USUAL PARAPHERNALIA OF PROGRAMMING LANGUAGES –* ***EACH SMALLTALK OBJECT IS A RECURSION ON THE ENTIRE POSSIBILITIES OF THE COMPUTER.***

Alan Kay

The End

Questions