Pharo: a Live Programming Environment

Pharo comes with an integrated development environment that allows you to browse not only your source code, but also the whole system. Pharo is a *live programming environment*: you can modify your objects and your code while your program is executing. All Pharo tools are implemented in Pharo:

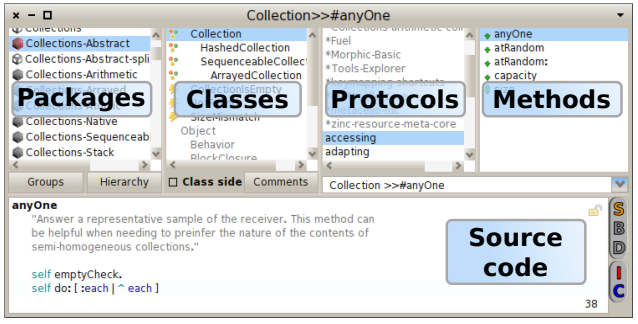
- a code browser with refactorings;
- a debugger, a workspace, and object inspectors;
- and much, much more!

Code can be inspected and evaluated directly in the image, using simple key combinations and menus (open the contextual menu on any selected text to see available options).

The Pharo Code Browser

The Pharo code browser is composed of 5 panes:

- The *packages* pane shows all the packages of the system;
- The *classes* pane shows a hierarchy of the classes in the selected package; the *class side* checkbox allows for getting the methods of the metaclass; the *comments* button toggles the display of the class's comment;
- The *protocols* pane groups the methods in the selected class to facilitate searching for a method when its name is not known; when a protocol name starts with a *, methods of this protocol belong to a different package (*e.g.*, the **Fuel* protocol groups methods that belong to the *Fuel* package);
- The *methods* pane lists the methods of the selected protocol; a down-arrow (resp. up-arrow) icon in front of a method indicates a method overridden in at least one subclass (resp. superclass); icons are clickable and trigger special actions;
- The *source code* pane shows the source code of the selected method. If the source code gets too long, the background color starts changing indicating it is time for refactoring!



A simple, uniform and powerful model

Pharo has a simple dynamically-typed object model:

- everything is an object — instance of a class;
- classes are objects too;
- there is single inheritance between classes;
- traits are groups of methods that can be reused orthogonally to inheritance;
- instance variables are protected;
- methods are public;
- blocks are lexical closure a.k.a. anonymous methods;
- computation happens only via message sends (and variable assignment).

Less is more: There is no type declaration, no primitive objects, no generic types, no modifiers, no operators, no inner classes, no constructor, and no static methods. You will never need them in Pharo!

Books

Pharo By Example (also available through amazon.com)
http://pharobyexample.org

Deep into Pharo (also available through amazon.com)
http://deepintopharo.com

Enterprise Pharo (draft of future book)
https://ci.inria.fr/pharo-contribution/job/PharoForTheEnterprise/ws/

More books
http://stephane.ducasse.free.fr/FreeBooks

Links

Main website http://www.pharo.org
Code hosting http://smalltalkhub.com
Questions http://stackoverflow.com/questions/tagged/pharo
Contributors http://contributors.pharo.org
Consultants http://consultants.pharo.org
Consortium http://consortium.pharo.org
Association http://association.pharo.org

Pharo

a clean, innovative, open-source Smalltalk-inspired environment

http://www.pharo.org

Pharo: the Elevator Pitch

Pharo is both an *object-oriented*, *dynamically-typed* general-purpose language and its own programming environment. The language has a simple and expressive syntax which can be learned in a few minutes. Concepts in Pharo are *very consistent*:

- Everything is an object: buttons, colors, arrays, numbers, classes, methods... *Everything!*
- A small number of rules, no exceptions!

Pharo runs in a *virtual machine*. Development takes place in an *image* in which all objects live and can be modified, eliminating the edit/compile/run cycle. Developers share and publish their source code using a dedicated version control system and services such as http://smalltalkhub.com. For deployment and debugging, the state of a running image can be saved at any point, then restored.

Minimal Syntax

Six reserved words only	
`nil`	the undefined object
`true`, `false`	boolean objects
`self`	the receiver of the current message
`super`	the receiver, in the superclass context
`thisContext`	the current invocation on the call stack

Reserved punctuation characters	
`"comment"`	
`'string'`	
`#symbol`	unique string
`$a`	the character a
`12 2r1100 16rC`	twelve (decimal, binary, hexadecimal)
`3.14 1.2e3`	floating-point numbers
`.`	expression separator (period)
`;`	message cascade (semicolon)
`:=`	assignment
`^`	return a result from a method (caret)
`[ :p | expr ]`	code block with a parameter
`| foo bar |`	declaration of two temporary variables
`#(abc 123)`	literal array with the symbol #abc and the number 123
`{foo . 3+2}`	dynamic array built from 2 expressions

Message Sending

A method is called by sending a message to an object, the message *receiver*; the message returns an object. Messages are modeled from natural languages, with a subject, a verb, and complements. There are three types of messages: unary, binary, and keyword.

A **unary message** is one with no arguments.

```
Array new.
#(1 2 3) size.
```

The first example creates and returns a new instance of the `Array` class, by sending the message `new` to the class `Array` (that is an object). The second message returns the size of the literal array which is 3.

A **binary message** takes only one argument and is named by one or more symbol characters.

```
3 + 4.
'Hello', ' World'.
```

The + message is sent to the object 3 with 4 as the argument. In the second case, the string `'Hello'` receives the message `,` (comma) with `' World'` as the argument.

A **keyword message** can take one or more arguments that are inserted in the message name.

```
'Smalltalk' allButFirst: 5.
3 to: 10 by: 2.
```

The first example sends the message `allButFirst:` to a string, with the argument 5. This returns the string `'talk'`. The second example sends `to:by:` to 3, with arguments 10 and 2; this returns a collection containing 3, 5, 7, and 9.

Precedence

Parentheses > unary > binary > keyword, and finally from left to right.

```
(10 between: 1 and: 2+4*3) not
```

Here, the messages + and * are sent first, then `between:and:` is sent, and finally `not`. The rule suffers no exception: operators are just binary messages with *no notion of mathematical precedence*, so 2 + 4 * 3 reads left-to-right and gives 18, not 14!

Cascading Messages

Multiple messages can be sent to the same receiver with `;`.

```
OrderedCollection new
  add: #abc;
  add: #def;
  add: #ghi.
```

The message `new` is sent to `OrderedCollection` which results in a new collection to which 3 `add:` messages are sent. The value of the whole message cascade is the value of the last message sent (here, the symbol `#ghi`). To return the receiver of the message cascade instead (*i.e.,* the collection), make sure to send `yourself` as the last message of the cascade.

Blocks

Blocks are objects containing code that is executed on demand, (anonymous functions). They are the basis for control structures like conditionals and loops.

```
2 = 2
  ifTrue: [Error signal: 'Help'].
#('Hello World' $!)
  do: [:e | Transcript show: e]
```

The first example sends the message `ifTrue:` to the boolean `true` (computed from `2 = 2`) with a block as argument. Because the boolean is `true`, the block is executed and an exception is signaled. The next example sends the message `do:` to an array. This evaluates the block once for each element, passing it via the e parameter. As a result, `Hello World!` is printed.

Unit testing

Pharo has a minimal still powerful framework that supports the creation and deployment of tests. A test must be implemented in a method whose name has a `test` prefix and in a class that subclasses `TestCase`.

```
OrderedCollectionTest >> testAdd
  | added |
  added := collection add: 'foo'.
  self assert: added == 'foo'.
  self assert: (collection includes: 'foo').
```

The `OrderedCollectionTest >> testAdd` notation indicates that the following text is the content of the method `testAdd` in the class `OrderedCollectionTest`. The second line declares the variable `added`. To execute unit-tests, open the `Test Runner` tool.