

# Pharo by Example

Andrew P. Black Stéphane Ducasse  
Oscar Nierstrasz Damien Pollet

with Damien Cassou and Marcus Denker

*Version of 2013-01-02*

This book is available as a free download from <http://PharoByExample.org>.

Copyright © 2007, 2008, 2009 by Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz and Damien Pollet.

The contents of this book are protected under Creative Commons Attribution-ShareAlike 3.0 Unported license.

*You are free:*

**to Share** — to copy, distribute and transmit the work

**to Remix** — to adapt the work

*Under the following conditions:*

**Attribution.** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page: [creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/)
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author’s moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license): [creativecommons.org/licenses/by-sa/3.0/legalcode](http://creativecommons.org/licenses/by-sa/3.0/legalcode)

Published by Square Bracket Associates, Switzerland. <http://SquareBracketAssociates.org>

ISBN 978-3-9523341-4-0

First Edition, October, 2009. Cover art by Samuel Morello.

# Contents

	<b>Preface</b>	<b>ix</b>
<b>I</b>	<b>Getting Started</b>	
<b>1</b>	<b>Pharo 早巡り</b>	<b>3</b>
1.1	入門 . . . . .	3
1.2	ワールドメニュー . . . . .	8
1.3	メッセージを送る . . . . .	8
1.4	Pharo のセッションを保存、終了そして再開する . . . . .	10
1.5	ワークスペースとトランスクリプト . . . . .	11
1.6	キーボードショートカット . . . . .	13
1.7	クラスブラウザ . . . . .	15
1.8	クラスを見つける . . . . .	16
1.9	メソッドを見つける . . . . .	19
1.10	メソッドを新しく定義する . . . . .	20
1.11	まとめ . . . . .	25
<b>2</b>	<b>最初のアプリケーション</b>	<b>27</b>
2.1	Lights Out ゲーム . . . . .	27
2.2	パッケージを作成する . . . . .	28
2.3	LOCell クラスを定義する . . . . .	29
2.4	クラスにメソッドを追加する . . . . .	31
2.5	オブジェクトをインスペクトする . . . . .	33

2.6	LOGame クラスを定義する . . . . .	34
2.7	メソッドをプロトコルにまとめる . . . . .	37
2.8	コードを実行してみましょう . . . . .	40
2.9	Smalltalk コードの保存と共有 . . . . .	43
2.10	章のまとめ . . . . .	48
<b>3</b>	<b>文法早わかり</b>	<b>49</b>
3.1	文法要素 . . . . .	49
3.2	擬似変数 . . . . .	52
3.3	メッセージ送信 . . . . .	53
3.4	メソッドのシンタックス . . . . .	54
3.5	ブロックのシンタックス . . . . .	55
3.6	条件式とループの要点 . . . . .	56
3.7	プリミティブとプラグマ . . . . .	58
3.8	まとめ . . . . .	58
<b>4</b>	<b>メッセージ構文を理解しよう</b>	<b>61</b>
4.1	メッセージを読み取る . . . . .	61
4.2	3種類のメッセージ . . . . .	63
4.3	メッセージ式の組み合わせ . . . . .	65
4.4	キーワードメッセージの切れ目を見つけるためのヒント . . . . .	72
4.5	式の並び . . . . .	73
4.6	カスケードメッセージ . . . . .	74
4.7	まとめ . . . . .	74
<b>II</b>	<b>Developing in Pharo</b>	
<b>5</b>	<b>Smalltalk オブジェクトモデル</b>	<b>79</b>
5.1	モデルの規則 . . . . .	79
5.2	すべてのものはオブジェクトである . . . . .	79
5.3	すべてのオブジェクトはクラスのインスタンスである . . . . .	80
5.4	すべてのクラスにはスーパークラスがある . . . . .	88
5.5	すべてはメッセージ送信で起こる . . . . .	91



5.6	メソッド探索は継承の連鎖をたどっていく . . . . .	92
5.7	共有変数 . . . . .	99
5.8	章のまとめ . . . . .	104
<b>6</b>	<b>Pharo のプログラミング環境</b>	<b>105</b>
6.1	概要 . . . . .	106
6.2	ブラウザ . . . . .	107
6.3	Monticello . . . . .	119
6.4	インスペクタと [Explore] . . . . .	127
6.5	デバッガ . . . . .	129
6.6	プロセスブラウザ . . . . .	139
6.7	メソッド検索 . . . . .	140
6.8	チェンジセットとチェンジソーター . . . . .	141
6.9	ファイルリストブラウザ . . . . .	143
6.10	Smalltalk では、コードを失うことはありません . . . . .	145
6.11	まとめ . . . . .	146
<b>7</b>	<b>SUnit</b>	<b>149</b>
7.1	はじめに . . . . .	149
7.2	なぜテストは重要か . . . . .	150
7.3	良いテストを書くには? . . . . .	151
7.4	SUnit by example . . . . .	152
7.5	SUnit クックブック . . . . .	156
7.6	SUnit フレームワーク . . . . .	157
7.7	SUnit のより高度な機能 . . . . .	160
7.8	SUnit の内部実装 . . . . .	161
7.9	テストのポイント . . . . .	164
7.10	まとめ . . . . .	165
<b>8</b>	<b>基本的なクラス</b>	<b>167</b>
8.1	Object . . . . .	167
8.2	Number . . . . .	177
8.3	Character . . . . .	180

8.4	String . . . . .	181
8.5	Boolean . . . . .	183
8.6	まとめ . . . . .	184
<b>9</b>	<b>コレクション</b>	<b>187</b>
9.1	はじめに . . . . .	187
9.2	コレクションの種類 . . . . .	188
9.3	コレクションの実装 . . . . .	191
9.4	主要なクラスの例 . . . . .	192
9.5	コレクションイテレータ . . . . .	202
9.6	コレクションを使うときのヒント . . . . .	206
9.7	まとめ . . . . .	208
<b>10</b>	<b>ストリーム</b>	<b>209</b>
10.1	二つの「要素の並び」 . . . . .	209
10.2	ストリーム対コレクション . . . . .	210
10.3	コレクションに対するストリーム処理 . . . . .	211
10.4	ストリームをファイルアクセスに使う . . . . .	219
10.5	まとめ . . . . .	222
<b>11</b>	<b>Morphic</b>	<b>223</b>
11.1	Morphic の歴史 . . . . .	223
11.2	モーフを操作する . . . . .	225
11.3	モーフの組み込み . . . . .	226
11.4	自由にモーフを作り、描く . . . . .	226
11.5	対話とアニメーション . . . . .	230
11.6	対話機能 . . . . .	234
11.7	ドラッグ&ドロップ . . . . .	234
11.8	例題 . . . . .	237
11.9	キャンバスに関する追加情報 . . . . .	241
11.10	まとめ . . . . .	242

<b>12</b>	<b>Seaside by Example</b>	<b>243</b>
12.1	なぜ Seaside? . . . . .	243
12.2	はじめに . . . . .	244
12.3	Seaside のコンポーネント . . . . .	249
12.4	XHTML のレンダリング . . . . .	252
12.5	CSS: カスケーディング・スタイル・シート . . . . .	259
12.6	制御フローの管理 . . . . .	261
12.7	サンプルアプリ作成によるチュートリアル . . . . .	268
12.8	AJAX の利用 . . . . .	274
12.9	この章のまとめ . . . . .	277
<b>III</b>	<b>Advanced Pharo</b>	
<b>13</b>	<b>クラスとメタクラス</b>	<b>283</b>
13.1	クラスとメタクラスのルール . . . . .	283
13.2	Smalltalk オブジェクトモデルの復習 . . . . .	284
13.3	すべてのクラスはメタクラスのインスタンスである . . . . .	286
13.4	メタクラスの階層はクラスの階層と並列に存在する . . . . .	287
13.5	すべてのメタクラスは Class と Behavior を継承している . . . . .	290
13.6	すべてのメタクラスは Metaclass のインスタンスである . . . . .	292
13.7	Metaclass のメタクラスは Metaclass のインスタンスである . . . . .	292
13.8	この章のまとめ . . . . .	294
<b>14</b>	<b>リフレクション</b>	<b>297</b>
14.1	イントロスペクション . . . . .	298
14.2	コードをブラウズする . . . . .	302
14.3	クラス、メソッド辞書、メソッド . . . . .	305
14.4	ブラウザ環境 . . . . .	307
14.5	実行時コンテキストにアクセスする . . . . .	309
14.6	理解されないメッセージをインターセプトする . . . . .	312
14.7	メソッドラッパーとしてのオブジェクト . . . . .	316
14.8	プラグマ . . . . .	319

14.9	章のまとめ . . . . .	320
------	-----------------	-----

## **IV Appendices**

<b>A</b>	<b>よくある質問</b>	<b>325</b>
A.1	はじめに . . . . .	325
A.2	コレクション . . . . .	325
A.3	システムブラウザ . . . . .	326
A.4	Monticello と SqueakSource を使う . . . . .	328
A.5	ツール . . . . .	329
A.6	正規表現と構文解析 . . . . .	329
	<b>Bibliography</b>	<b>331</b>

# Preface

## Pharo とは?

Pharo は Smalltalk プログラミング言語・環境をフルに実装した、オープンソースによる新しい処理系です。Pharo は古典的な Smalltalk-80 システムの再実装である、Squeak<sup>1</sup>から派生しました。Squeak が主に実験的な教育向けソフトウェアを開発するためのプラットフォームとして作られたのに対して、Pharo はプロフェッショナルなソフトウェアを開発するための、凝縮されたオープンソース・プラットフォームを目指しています。また、動的な言語・環境の研究開発のための、頑丈で安定したプラットフォームの提供にも努めます。Pharo はウェブアプリ開発フレームワーク Seaside のリファレンス実装用にも使われています。

Pharo は Squeak にあったライセンスの問題を解決しています。Squeak の以前のバージョンとは異なり、Pharo のコア部分は MIT ライセンスで寄贈されたコードのみからできています。Pharo プロジェクトは 2008 年 3 月に Squeak 3.9 からのフォークとして始まり、最初の 1.0 ベータバージョンは 2009 年 7 月 31 日にリリースされました。

Pharo は Squeak からパッケージをいくつも取り去った一方で、Squeak ではオプションだった機能をたくさん取り入れています。例えば、Pharo には True Type フォントが入っています。Pharo は本物のブロッククロージャもサポートしています (訳注:Squeak では BlockContext という再帰できないクロージャ実装でした)。ユーザインターフェースもシンプルで洗練されたものになっています。

Pharo は極めて高い移植性を持っています。仮想マシンでさえもすべて Smalltalk で書かれていて、デバッグ・解析・変更をしやすくなっています。Pharo はマルチメディア・アプリケーションから、教育向けプラットフォーム、商用ウェブ開発環境まで、広い範囲で革新的なプロジェクトの媒体となります。

---

<sup>1</sup>Dan Ingalls et al., Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97). ACM Press, November 1997 (URL: <http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html>).

Pharo は重要な視点を背景として持っています。「Pharo は過去の単なる複製であってはならない。Smalltalk を再発明すべきだ」というものです。ビッグバンアプローチは滅多に成功しません。Pharo では進化的でインクリメンタルな変化をさせる方法を取ります。重要な新機能やライブラリ開発を実験的に進められるようにしたいのです。進化とは Pharo が誤りを受け入れていくことを意味します。Pharo では一度で完璧に解決することを目指しません。そうしたくなる気持ちをこらえてでも、です。小さな変化をいくつも積み上げていくやり方が、Pharo ではうまくいくでしょう。Pharo の成功はコミュニティからの貢献にかかっているのです。

## この本が想定している読者

この本はオープンソースで公開されている Squeak の入門書 *Squeak by Example*<sup>2</sup> を元にしてしています。その名の通り、Pharo と Squeak の違いを反映して修正しました。Pharo の基本からはじまり、先進的な話題まで、内容は多岐にわたります。

この本はどうやってプログラムするのかを教える本ではありません。この本は既にプログラミング言語をいくつか知っている人に向けて書かれています。オブジェクト指向プログラミングの経験も役に立つでしょう。

この本では Pharo のプログラミング環境 (言語と関連ツール) を紹介します。一般的なイディオムや実例に触れてもらいますが、オブジェクト指向設計よりも技術の方に焦点を当てています。可能限り例をたくさん示そうと思います (Alec Sharp による素晴らしい Smalltalk の本<sup>3</sup> にインスパイアされています)。

ウェブ上には Smalltalk についての無料の本が他にもたくさんあります。しかし Pharo に特化したものはありません。参照: <http://stephane.ducasse.free.fr/FreeBooks.html>

## 読者に向けてのアドバイス

個々の Smalltalk の記述がすぐにわからないからといっていらだってははいけません。すべてを知る必要はないのです! Alan Knight は、以下のようにこの原則を表現しています。<sup>4</sup>:

---

<sup>2</sup><http://SqueakByExample.org>

<sup>3</sup>Alec Sharp, *Smalltalk by Example*. McGraw-Hill, 1997 (URL: <http://stephane.ducasse.free.fr/FreeBooks/ByExample/>).

<sup>4</sup><http://www.surfskranton.com/architecture/KnightsPrinciples.htm>

気にしないようにする Smalltalk を学び始めたプログラマは、どうやってそれが動くのかを詳細まですべて理解しなければならないと考え、困ってしまうことがよくあります。つまり、Transcript show: 'Hello World' をマスターするまでにかなり時間がかかるということです。オブジェクト指向の優れた点は、「どうやって動くのだろう?」という問いに対して「気にしない」と答えられることなのです。

## オープンな本

この本は、以下の意味でオープンな本です。

- この本の内容はクリエイティブ・コモンズの表示—継承 (by-sa) ライセンスで公開されています。つまり、以下の URL にあるライセンス条件を尊重する限り、この本を自由に配布したり改変したりできます。URL: <http://creativecommons.org/licenses/by-sa/3.0/deed.ja>
- この本は Pharo のコア部分のみを解説しています。理想を言うと、私たちが書けなかった部分も他の人が寄稿できるようにしたいと思っています。この取り組みに参加したい人は連絡してください。私たちはこの本が育つのを見たいのです!

詳しくは、<http://PharoByExample.org> を見てください。

## Pharo のコミュニティ

Pharo のコミュニティは親切で活動的です。役に立ちそうなリソースをいくつかここに挙げておきます。

- <http://www.pharo-project.org> Pharo のメインとなるウェブサイト。
- <http://www.squeaksource.com> Pharo プロジェクトにとっての SourceForge 的なもの。Pharo 用の追加パッケージがここでたくさん生まれています。


## 例と練習問題


この本では特殊な記法を二つ決めました。

できる限りたくさん例を示すようにしています。特に、実行可能な短いコードで示した例がたくさんあります。式を選択して `print it` した結果を示すために、`→` 記号を使っています。

3 + 4 → 7 "3+4を選択して'print it'を選ぶと7が得られる"

Pharo で実際に試すときのために、この本のサイト (<http://PharoByExample.org>) からすべてのコード例がテキストファイルでダウンロードできるようになっています。

二つ目の決まりとして、みなさんにしてほしいことを示すため、アイコンを使います。

 次の章へと読み進めましょう!

## 謝辞

まず、Squeak というこの驚くべき Smalltalk 開発環境を、オープンソース・プロジェクトとして公開したオリジナルの開発者たちに感謝します。

Smalltalk についてのコラムを翻訳することを許可してくれた Hilaire Fernandes と Serge Stinckwich、ストリームの章を寄稿してくれた Damien Cassou に感謝します。

査読してくれた Alexandre Bergel、Orla Greevy、Fabrizio Perin、Lukas Renggli、Jorge Ressia、Erwann Wernli に特に感謝します。

このオープンソース・プロジェクトを快く援助し、この本のウェブサイトを設置してくれたベルン大学 (スイス) に感謝します。

そして、Squeak のコミュニティに感謝します。この本のプロジェクトを熱意を持って助け、また、この本の初版の間違いを教えてくださいました。



Part I


# Getting Started



# Chapter 1

## Pharo 早巡り

この章では、Pharo 環境に親しんでもらうための、概要を解説します。Pharo を実際に試す場面がたくさんあるので、この章はコンピュータを手元に置いて読むとよいでしょう。

Pharo を動かして試してもらいたい箇所にはアイコン:  で印を付けておきます。Pharo の起動方法、システムとやりとりする様々なやり方、基本的なツールについて知ることになるでしょう。また、新しいメソッドの定義の仕方や、オブジェクトを作成してメッセージを送る方法についても学びます。

### 1.1 入門

Pharo は <http://pharo-project.org> から自由にダウンロードできます。三つのパーツをダウンロードしなければなりません。それらは実際には4個のファイルからなります (図 1.1)。

1. バーチャルマシン (VM)。Pharo のシステムで唯一、オペレーティングシステムとプロセッサごとに異なっているものです。主要なプラットフォーム用にコンパイル済みのバーチャルマシンが用意されています。図 1.1 にあるのは、某プラットフォーム用のバーチャルマシンで、*Pharo.exe* という名前です。
2. ソースファイル。ソースファイルには Pharo の中で、あまり頻繁には変更されない部分のソースコードが入っています。図 1.1 では *SqueakV39.sources* がソースファイルです。<sup>1</sup>
3. 仮想イメージ。システム仮想イメージは、実行中の Pharo システムのスナップショットです。このスナップショットは二つのファイルからなり

---

<sup>1</sup>Pharo は Squeak 3.9 をベースに作られています。今のところバーチャルマシンは Squeak と同じものを使っています。

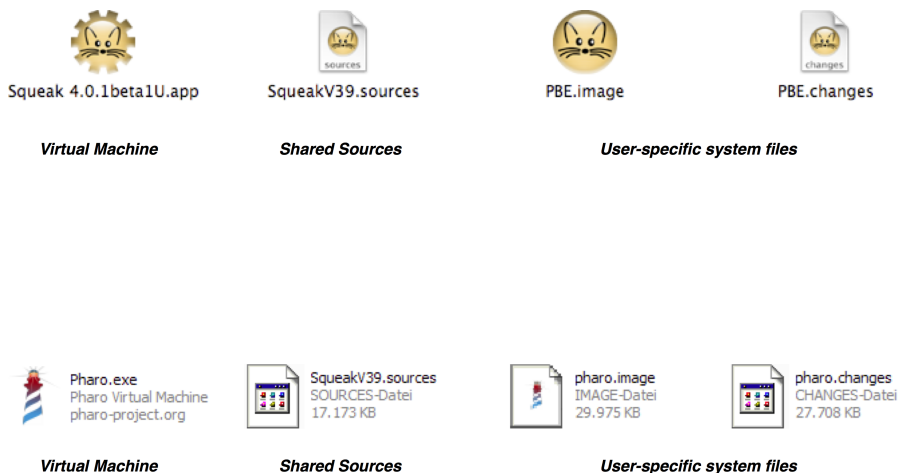


Figure 1.1: Pharo がサポートするプラットフォーム用のダウンロードファイル。

ます。イメージファイル イメージファイル (.image) にはシステムのすべてのオブジェクト (クラスやメソッドもオブジェクトです) が入っています。チェンジファイル チェンジファイル (.changes) にはシステムに加えたソースコードの変更ログが入ります。図 1.1 では、これらのファイルは *pharo.image* および *pharo.changes* です。

📥 *Pharo* をダウンロードしてインストールしましょう。

Pharo by Example のウェブページにある仮想イメージを使うことをお勧めします。<sup>2</sup>

本書のサンプルのほとんどは *Pharo* のどのバージョンでも動作するので、もし *Pharo* を既にインストールしているのならそれを使って構いません。ただし、ここで説明することと、見た目や振る舞いが違ったからといって驚かないでください。

*Pharo* で作業しているとき、イメージファイルとチェンジファイルは常に内容が更新されるため、これらは書き込み可能にしておいてください。そして、二つのファイルは常に同じ場所に置いてください。テキストエディタで直接変更してはいけません。*Pharo* は、オブジェクトを格納したりソースコードの変更のログを書き出したりするのにこの二つのファイルを使います。ダウンロードしたイメージファイルとチェンジファイルのバックアップを取っておくとよいでしょう。こうしておけば、いつでもまっさらな仮想イメージから *Pharo* を開始し、自分で書いたコードを後からロードできます。

ソースファイルとバーチャルマシンは書き込み不可にして、ユーザ間で共有できます。バーチャルマシン、ソースファイル、イメージファイル (チェン

<sup>2</sup><http://PharoByExample.org>


ジファイル) は同じ場所に置いてかまいませんし、バーチャルマシンとソースファイルだけ別の共有ディレクトリに置いておくこともできます。オペレーティングシステムや作業スタイルに合わせて一番良い方法を選んでください。



Figure 1.2: <http://PharoByExample.org> からダウンロードしたままのまっさらな仮想イメージ。

**起動する。** Pharo の起動は普通のアプリケーションと同じです: バーチャルマシンのアイコンにイメージファイルをドラッグ&ドロップしたり、イメージファイルをダブルクリックしたり、あるいはコマンドラインでバーチャルマシンの名前の後にイメージファイルへのパスを入力することになるでしょう。使っているオペレーティングシステムに合わせてください (異なるバージョンのバーチャルマシンがあると、オペレーティングシステムは正しいバーチャルマシンを自動で選んでくれないかもしれません。そうしたときはドラッグ&ドロップ、あるいはコマンドラインから起動した方が安全でしょう)。

Pharo を起動すると、1 個の大きなウィンドウが現れます。ウィンドウの中にはワークスペースウィンドウがいくつか開いているかもしれません (図 1.2)。しかしどうすればよいのか、これだけではわかりません。メニューバーが出ることもあります、Pharo ではもっぱらコンテキスト依存のポップアップメニューを使います。

 **Pharo** を始めましょう。ウィンドウの左上にある赤いボタンをクリックするとワークスペースを閉じることができます。

ウィンドウを最小化するにはオレンジ色のボタン、最大化するには緑色の

ボタンをクリックします。

最初のやりとり。 図 1.3(a) のワールドメニュー から始めるのがいいでしょう。

🔊 メインウィンドウの背景でマウスのボタンをクリックしてワールドメニューを表示しましょう。そして **Workspace** を選択して新しいワークスペースを開きましょう。

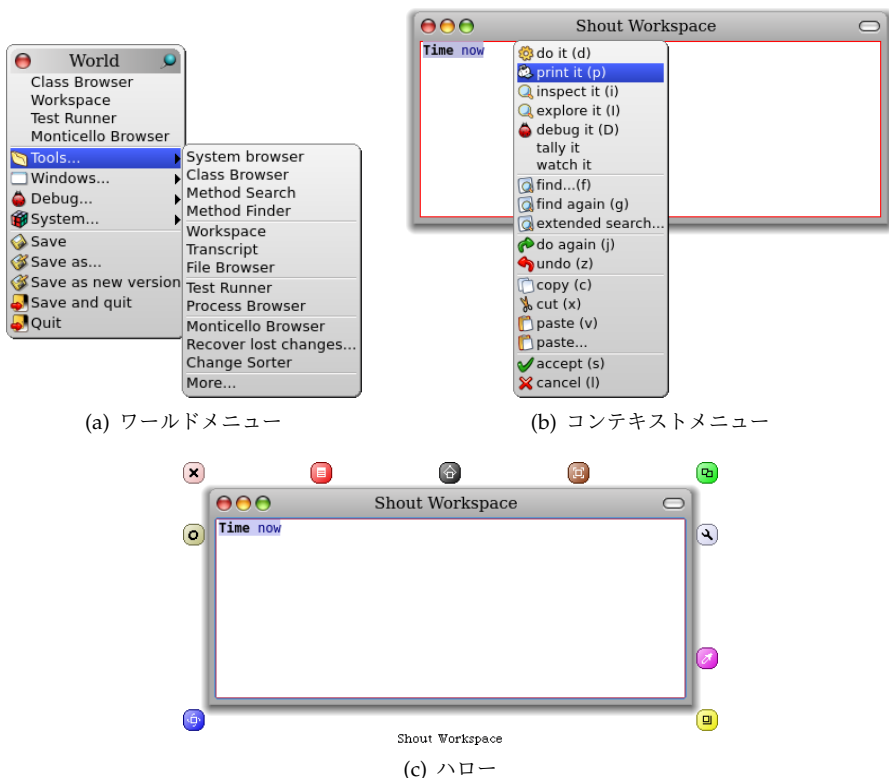


Figure 1.3: ワールドメニュー (クリックすれば出る)、コンテキストメニュー (アクションクリック)、ハロー (メタクリック)。


Smalltalk は元々 3 ボタンマウスのコンピュータを想定して設計されています。マウスのボタンが足りないときは、修飾キーを押しながらマウスをクリックしなければなりません。2 ボタンマウスでも十分 Pharo を使うことはできますが、もし 1 ボタンマウスしか持っていないなら、クリックできるスクロールホイールが付いた 2 ボタンマウスを買うことを考えた方がいいでしょう。そうすれば Pharo をいっそう楽しく使えるようになります。


様々なコンピュータ、マウス、キーボードそして個人設定があるため、Pharoでは「マウスの左ボタンをクリック」という表現は避けています。元々 Smalltalk ではそれぞれのマウスボタンを色で表していました。<sup>3</sup>あることをするのにユーザが様々な修飾キー (*control*、*ALT*、*meta* など) を使うので、この本では代わりに以下の用語を用います:

**クリック:** 一番よく使われるマウスボタンで、通常は修飾キーを使わずに1ボタンマウスをクリックするのと同じです。背景をクリックして「ワールド」メニュー (図 1.3 (a)) を表示してみましょう。

**アクションクリック:** 次によく使われるボタンです。このボタンでコンテキストメニューを表示します。コンテキストメニューが表示するアクションの一覧は、マウスが指している場所によって異なります; 図 1.3 (b) を見てください。マウスに該当するボタンがない場合、普通は *control* 修飾キーを使ってアクションクリックするように設定します。

**メタクリック:** 最後に、画面上の任意のオブジェクトをメタクリックすると、そのオブジェクトの「ハロー」を表示させることができます。ハローは画面上のオブジェクトを回転させたりリサイズしたりするためのハンドルの集まりです。図 1.3 (c) を見てください。<sup>4</sup>ハンドルにマウスをかざしておけば、バルーンヘルプが出てその機能について教えてくれます。Pharo では、メタクリックする方法はオペレーティングシステムに依存します。*SHIFT ctrl* あるいは *SHIFT option* を押しながらクリックすることになるでしょう。

 **Time now** とワークスペースに入力しましょう。そしてワークスペースでアクションクリックして、**print it** を選択しましょう。

右利きの人なら クリック は左ボタンに、アクションクリック は右ボタンに、メタクリック はクリックできるスクロールホイール (あれば) に、それぞれ設定することをお勧めします。Macintosh で1ボタンのマウスを使っているのなら、 を押しながらマウスをクリックすることでアクションクリックかメタクリックをシミュレートすることもできます。とはいえ、これから Pharo を頻繁に使うのなら、最低でも二つ以上のボタンの付いたマウスに投資することをお勧めします。

オペレーティングシステムやマウスドライバの設定を変えることで、マウスの動作を望み通りにできます。Pharo にはマウスやキーボードのメタキーの設定を変更できる機能もあります。プリファレンス・ブラウザ (System ...▷ Preferences ...▷ Preference Browser...) では、**keyboard** カテゴリの **swapControlAndAltKeys** オプションを使うことでアクションクリックとメタクリックを入れ替えることができます。プリファレンス・ブラウザには他にも様々なキーボードショートカットを設定するオプションがあります。

<sup>3</sup>マウスボタンの色は赤、黄そして青です。著者はどの色がどのボタンを指していたのかさっぱり思い出せません。

<sup>4</sup>Pharo はデフォルトでハンドルを無効にしていますが、これらはプリファレンス・ブラウザを使って有効にできます。プリファレンス・ブラウザについてはすぐ後で述べます。

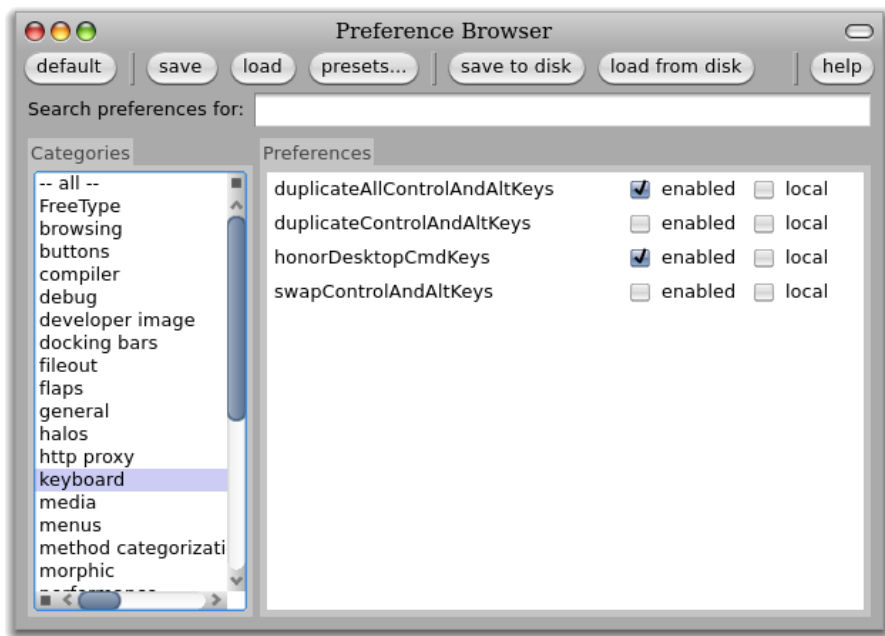


Figure 1.4: プリファレンス・ブラウザ。

## 1.2 ワールドメニュー

🕒 *Pharo* の背景でもう一度クリックしてみましょう。

再び **World** メニューが表示されるはずです。ほとんどの *Pharo* のメニューにはモードがありません。メニューは、右上のピンアイコンをクリックして、画面上に残しておくことができます。やってみましょう。

ワールドメニューを使えば多くの *Pharo* のツールに簡単にアクセスすることができます。

🕒 **World** メニューと **Tools ...** メニューを詳しく見てみましょう (図 1.3(a))。

ブラウザやワークスペースといった *Pharo* の主要なツールがリストされています。以降の章ではこれらのツールのほとんどを使うことになります。

## 1.3 メッセージを送る

🕒 ワークスペースを開けましょう。そして以下のテキストを入力しましょう。



BouncingAtomsMorph new openInWorld

🕒 アクションをクリックしましょう。メニューが現れるはずですが、`do it (d)` を選択しましょう (図 1.5)。

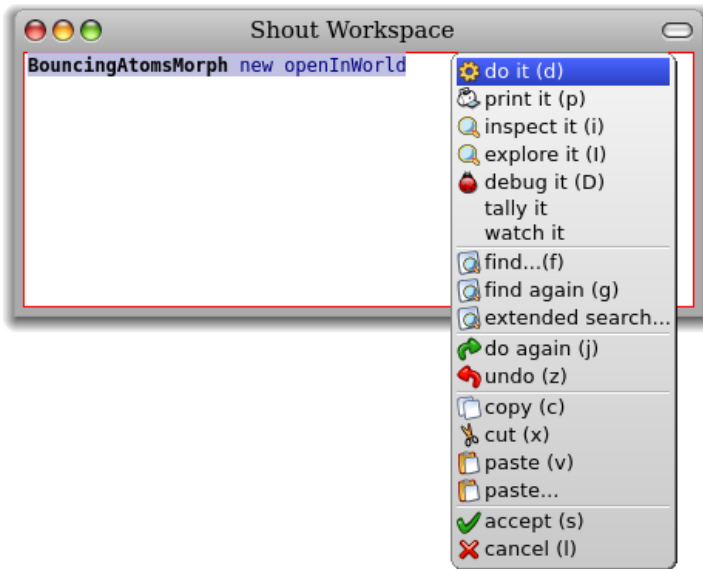


Figure 1.5: 式を“do it”する。

Pharo の画面左上に、たくさんの原子が弾んでいるウィンドウが表示されただけです。

あなたはたった今、最初の Smalltalk の式を実行しました! BouncingAtomsMorph クラスへ new メッセージを送り、結果として BouncingAtomsMorph のインスタンスが生成され、さらに openInWorld メッセージがこのインスタンスに送られました。BouncingAtomsMorph クラスは new メッセージを受け取った際、自分がすることを決めました。つまり、bam クラスは new メッセージに対するメソッドを探して適切に反応しました。同様に BouncingAtomsMorph のインスタンスも openInWorld に対するメソッドを探して適切なアクションを取りました。

Smalltalker としばらく話せば、すぐに、彼らが「手続きをコールする」または「メソッドを呼び出す」といった表現を使わず、代わりに「メッセージを送る」と言っていることに気づくでしょう。これは、オブジェクトが自身のアクションに責任を持つという思想の現れです。決して、オブジェクトに何々をしろと命令することはありません。代わりにメッセージを送って、何かしてほしいと礼儀正しく頼みます。あなたではなくオブジェクトが、メッセージに反応するための適切なメソッドを選ぶのです。

## 1.4 Pharo のセッションを保存、終了そして再開する

🌀 弾む原子のウィンドウをクリックして好きなところへドラッグしましょう。ウィンドウは「意のまま」です。クリックして適切な位置に置きましょう。

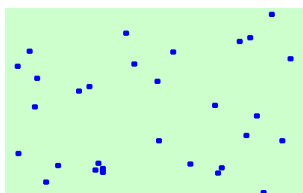


Figure 1.6: BouncingAtomsMorph。

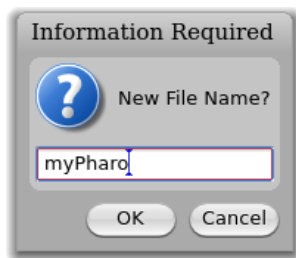


Figure 1.7: save as ... ダイアログ。

🌀 **World**▷**Save as ...** を選択して「myPharo」と入力し、**OK** ボタンをクリックしましょう。そして **World**▷**Save and quit** を選びましょう。

オリジナルのイメージファイルとチェンジファイルがあった場所に「myPharo.image」と「myPharo.changes」というファイルができています。これらのファイルには **Save and quit** する直前の Pharo 仮想イメージの動作中の状態が入っています。この二つのファイルはディスク上の好きな場所に移動してかまいませんが、(オペレーティングシステムによっては)バーチャルマシンとソースファイルも同じ場所へ移動、コピー(あるいはリンク)する必要がありますかもしれません。

🌀 今作った「myPharo.image」ファイルを使って Pharo を起動しましょう。

先ほど Pharo を終了したときとそっくりそのままの状態に戻ったことに気づきでしょう。BouncingAtomsMorph も同じところにあり、原子も弾み続けています。

Pharo を起動すると、Pharo バーチャルマシンは指定されたイメージファイルを読み込みます。このファイルにはたくさんのオブジェクトのスナップショットが入っています。これらのオブジェクトには、既に書かれた大量のコードやたくさんのプログラミングツール(これらはすべてオブジェクトです)が含まれます。Pharo を使っていると、これらのオブジェクトへメッセージを送ったり、新しいオブジェクトを作ったりすることになります。また、いくつかのオブジェクトは命を終え、オブジェクトに割り当てられていたメモリは回収(すなわちガベージコレクション)されます。


Pharo を終了するとき、普通はすべてのオブジェクトのスナップショットを保存することになるでしょう。ここで通常、古いイメージファイルは上書きさ

れることとなりますが、先ほど行ったように新しく名前を付けてイメージファイルを保存することもできます。

イメージファイルの他にチェンジファイルもあります。このファイルは、標準のツールを使っていったソースコードの変更をすべて記録しています。通常このファイルはまったく意識する必要はありません。しかし後で見るように、チェンジファイルはエラーから回復したり保存しそこなった変更を再現したりするのに重宝します。これについては後ほど!

みなさんがここまで使ってきた仮想イメージは、1970年代後半に作られたオリジナルの Smalltalk-80 の仮想イメージの子孫です。この仮想イメージの中には何十年と生き続けているオブジェクトもあります!

プロジェクトを保存し管理するのにも仮想イメージが基本的なメカニズムとして使われていると思うかもしれませんが、それは違います。すぐ後で見るように、ソースコードを管理し、ソフトウェアをチームで共有するにはもっと良いツールがあります。仮想イメージは非常に便利ですが、Monticello などのツールを使えばバージョン管理やコードの共有がもっとうまくできるので、仮想イメージに執着せず、無造作に作ったり捨てたりすることに慣れた方がよいでしょう。

 マウス (と必要ならば修飾キー) を使って `BouncingAtomsMorph` をメタクリックしましょう。<sup>5</sup>

色とりどりの円が表示されたはずです。これらをまとめて `BouncingAtomsMorph` のハローと呼びます。一つ一つの円をハンドルと呼びます。十字のピンクのハンドルをクリックしましょう; `BouncingAtomsMorph` は消えるはずです。

## 1.5 ワークスペースとトランスクリプト

 すべてのウィンドウを閉じ、トランスクリプト とワークスペース を開きましょう (トランスクリプトは `World ▸ Tools ...` サブメニューから開けます)。

 トランスクリプト とワークスペースの位置やサイズを変えて、ワークスペースをトランスクリプトにオーバーラップさせましょう。

ウィンドウをリサイズするにはウィンドウの角をドラッグするか、メタクリックしてハローを出して黄色の (右下の) ハンドルをドラッグします。

常にアクティブなウィンドウは一つだけです。そのウィンドウは前面にあり枠がハイライトされています。

トランスクリプトは、システムメッセージのログを取るのにしばしば使われるオブジェクトです。トランスクリプトは「システムコンソール」の一種です。

<sup>5</sup> うまくいかない場合はプリファレンス・ブラウザの `halosEnabled` オプションをチェックしてみてください。

ワークスペースは、試してみたい Smalltalk コードの断片を入力するのに役立ちます。ワークスペースにはまた、任意のテキスト (TODO リストや仮想イメージを使う人への手引きなど) を書き残しておくこともできます。ワークスペースはしばしば、保存した仮想イメージについてドキュメントを記すのに使われます。先ほどダウンロードした標準仮想イメージがその例です (図 1.2)。

🕒 ワークスペースに次のテキストを入力しましょう：

```
Transcript show: 'hello world'; cr.
```

ワークスペースで、今入力したテキストの様々な箇所をダブルクリックしてみましょう。単語の上、文字列の終わりあるいは式全体の終わりという具合にクリックしてみて、単語全体、文字列全体あるいはテキスト全体が選択されるのを確認しましょう。

🕒 入力したテキスト全体を選択してアクションクリックし、`do it (d)` を選択しましょう。

トランスクリプトウィンドウに「hello world」と表示されます。(図 1.8)。もう一度やってみましょう。(メニュー項目 `do it (d)` の中の `(d)` は `do it` へのキーボードショートカットが `CMD-d` であることを示しています。これについて詳しくは次の節で!)

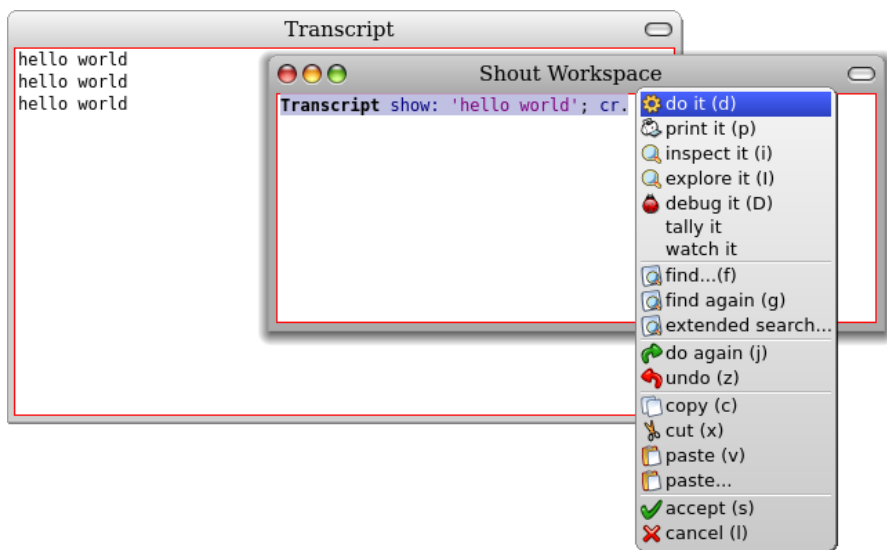


Figure 1.8: オーバーラップしたウィンドウ。ワークスペースがアクティブ。

## 1.6 キーボードショートカット

式を評価したいとき、いつもアクションクリックする必要はありません。代わりにキーボードショートカットを使うこともできます。メニューの括弧書きの部分が該当します。プラットフォームに応じて、いずれかの修飾キー (control、alt、command、あるいは meta) を押すことになるでしょう (このようなキーボードショートカットを以後、CMD-キーと表記することにします)。

🕒 もう一度ワークスペースの式を評価してみましょう。ただしキーボードショートカットを使ってください (CMD-d)。

do it の他に print it、inspect it そして explore it にも気づいたでしょう。手短かに説明します。

🕒 3 + 4 とワークスペースに入力しましょう。そしてキーボードショートカットを使って do it します。

何も起きなかったとしても驚かないでください! ここでは 3 という数に + というメッセージを 4 という引数付きで送ったことになります。普通に 7 が計算されて返されたのですが、ワークスペースはこの答えをどうすべきかわからないので答えを捨ててしまいます。結果を見たければ、代わりに print it を使う必要があります。print it は式をコンパイル、実行して、その結果に printString を送ってできた文字列を表示します。

🕒 3+4 を選択して print it しましょう (CMD-p)。

今度は期待通りの結果になります (図 1.9)。

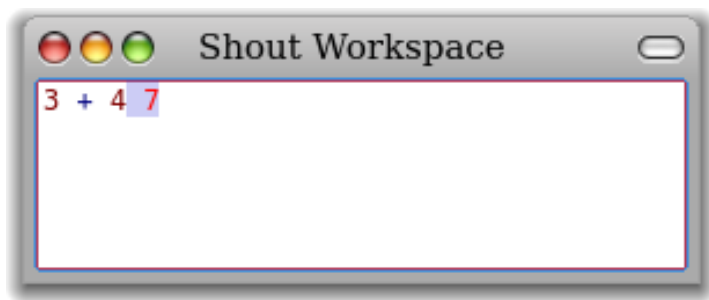


Figure 1.9: “do it” ではなく “print it”。

3 + 4 → 7

この本の約束事として、特定の Pharo の式を print it したらどうなるかを示すのに → の表記を用います。

👁️ 選択されている「7」を削除し (Pharo は既に「7」を選択しているはずですが。だから `delete` キーを押すだけです)、`3+4` をもう一度選択して今度は `inspect it` (`CMD-i`) しましょう。

タイトルに `SmallInteger: 7` と書かれたウィンドウが表示されるはずですが。これをインスペクタと呼びます (図 1.10)。インスペクタは極めて便利なツールで、これを用いればシステムのどんなオブジェクトもブラウズできますし、どんなオブジェクトともやりとりすることができます。タイトルの意味は、7 はクラス `SmallInteger` のインスタンスであるということです。左のペインを使ってオブジェクトのインスタンス変数をブラウズすることができます。インスタンス変数の値は右のペインに表示されます。下のペインにはオブジェクトへメッセージを送るための式を書くことができます。

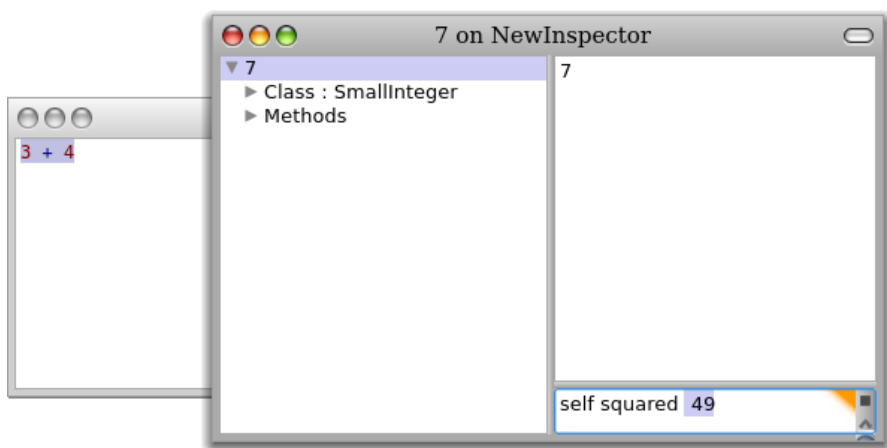


Figure 1.10: オブジェクトをインスペクトする。

👁️ 7 のインスペクタの下のペインに `self squared` と入力して `print it` しましょう。

👁️ インスペクタを閉じましょう。ワークスペースに `Object` と入力して今度は `explore it` (`CMD-I`、大文字の `i`) します。

今度は `Object` のタイトルが付いたウィンドウ (オブジェクト・エクスプローラ) が現れるはずですが。このウィンドウの中には `root: Object` というテキストがあります。三角をクリックして中身を開いてください (図 1.11)。

オブジェクト・エクスプローラはインスペクタと似ていますが、複雑なオブジェクトのツリービューを提供する点で異なります。この例では、見ているオブジェクトは `Object` クラスです。ここではこのクラスに格納されている要素を直に見ることができますし、さらに要素の内部構造をたどっていくことも容易にできます。

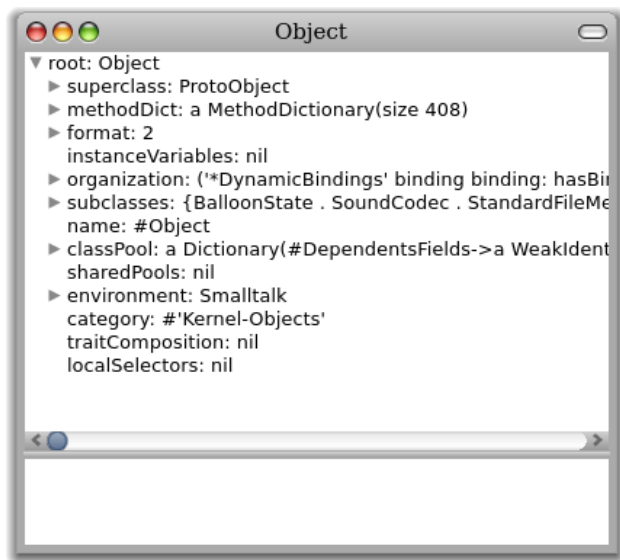


Figure 1.11: Object をエクスプロアする。

## 1.7 クラスブラウザ

クラスブラウザ<sup>6</sup>はプログラミングにとって重要なツールです。後で見るとように Pharo にはいくつかの興味深いブラウザがありますが、どの仮想イメージを使う場合でも、クラスブラウザは最も基本的なブラウザです。

🔍 **World** ▷ **Class browser** を選択してブラウザを開きましょう。<sup>7</sup>

図 1.12 がブラウザです。タイトルバーは **Object** クラスをブラウズしていることを示しています。

ブラウザを開くと、一番左のペイン以外、すべて空欄となっています。このペインで全パッケージを一覧できます。各パッケージには関連するクラスがグループ化されて入っています。

🔍 **Kernel** パッケージをクリックしましょう。

選択されたパッケージに含まれるクラスが 2 番目のペインにリストされます。

<sup>6</sup>紛らわしいことに「システムブラウザ」や「コードブラウザ」などと呼ばれることがあります。Pharo では **OmniBrowser** というブラウザの実装が使われています。OmniBrowser は「OB」または「パッケージブラウザ」としても知られています。この本では単に「ブラウザ」としますが、曖昧さを避けるために「クラスブラウザ」も用います。

<sup>7</sup>もしブラウザが図 1.12 のようでなければ、デフォルトのブラウザを変える必要があるでしょう。FAQ 5, p. 326 をご覧ください。

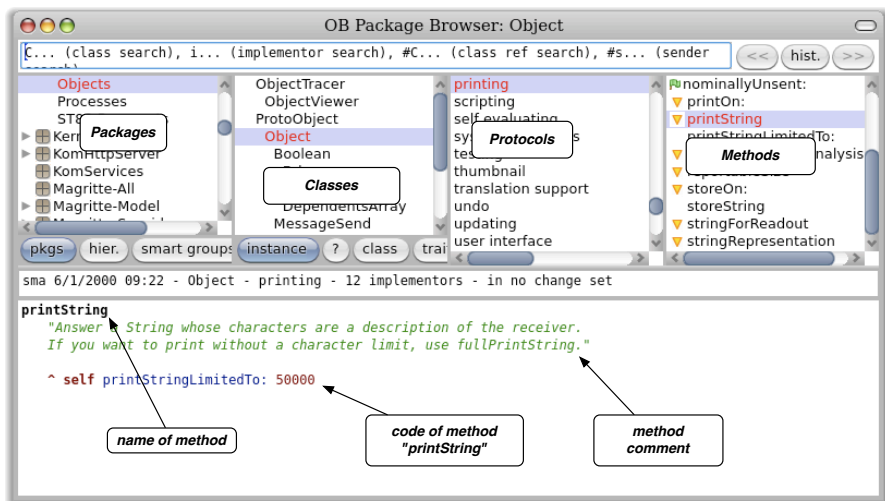


Figure 1.12: Object クラスの printString メソッドを表示しているブラウザ。

🕒 Object クラスを選択しましょう。

今度は残りの二つのペインにテキストが表示されます。3 番目のペインには選択されたクラスのプロトコルが表示されます。プロトコルは、関連するメソッドを扱いやすいようにグループ分けしたものです。プロトコルを何も選択していなければ、すべてのメソッドが4 番目のペインに表示されます。

🕒 printing プロトコルを選択しましょう。

printing プロトコルを見つけるには下にスクロールしなければならないかもしれません。今回は文字列表示に属するメソッドだけが4 番目のペインに表示されます。

🕒 printString メソッドを選択しましょう。

今度は下のペインに printString メソッドのソースコードが表示されます。このメソッドはすべてのオブジェクトで共有されます (メソッドをオーバーライドするものを除く)。

## 1.8 クラスを見つける

Pharo でクラスを見つける方法は複数あります。一つ目は、今見たようにブラウザを使ってパッケージからたどっていく方法です。この場合パッケージ名は最初からわかっているか推測することになります。



二つ目はクラスに `browse` メッセージを送って自身のブラウザを開いてもらう方法です。例えば `Boolean` クラスをブラウズしたいとしましょう。

🕒 ワークスペースに `Boolean browse` と入力し、`do it` しましょう。

`Boolean` クラスのブラウザが開きます (図 1.13)。キーボードショートカット `CMD-b` (`browse`) もあります。このキーボードショートカットはクラス名が現れる場所ならどのツールの中でも使えます。クラス名を選択して `CMD-b` をタイプすればよいのです。

🕒 キーボードショートカットを使って `Boolean` クラスをブラウズしましょう。

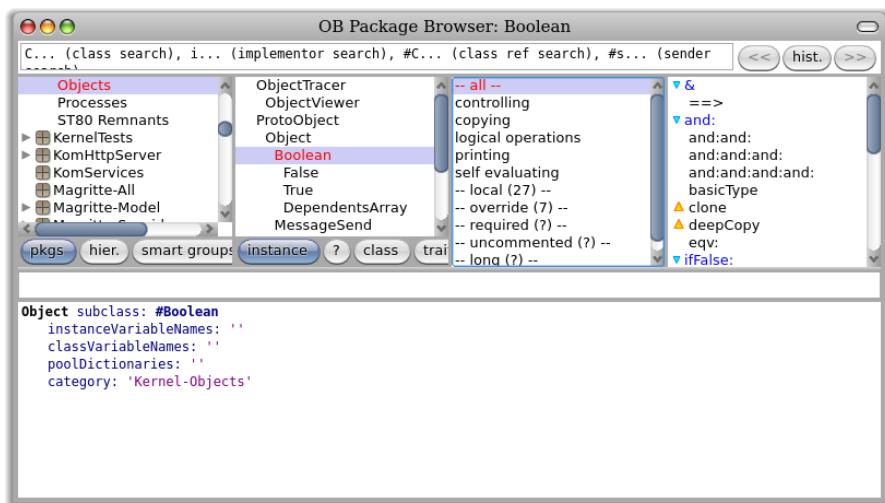


Figure 1.13: `Boolean` クラスの定義を表示しているブラウザ。

`Boolean` クラスが選択されていて、プロトコルもメソッドも選択されていないときは、メソッドのソースコードの代わりにクラス定義が表示されることに注意してください (図 1.13)。クラス定義は、親クラスにサブクラスの作成を依頼するという、Smalltalk のメッセージ送信にすぎません。ここでは `Object` クラスが `Boolean` という名前のサブクラスを作成するよう依頼されているのがわかります。`Boolean` クラスのインスタンス変数、クラス変数、プール辞書は空で、カテゴリは `Kernel-Objects` になっています。クラスペインの下の `?` をクリックすると、専用のペインでクラスコメントを見ることができます (図 1.14)。

クラスを見つけるには、大体は名前を検索するのが一番の早道です。例えば日付と時間を表すクラスを探しているとしましょう。

🕒 ブラウザのパッケージペインにマウスを置き、`CMD-f` とタイプするかアクションクリックして `find class ...(f)` を選択しましょう。ダイアログボックスに「`time`」と打ってください。

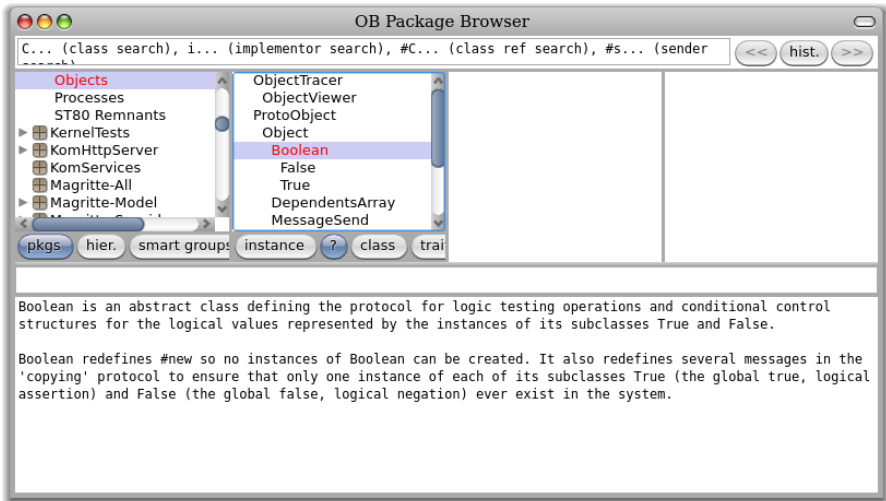


Figure 1.14: Boolean クラスのクラスコメント。

「time」を含んだクラス名のリストが表示されます (図 1.15)。Time を選択しましょう。ブラウザが Time クラスを表示します。Time のクラスコメントには関連するクラスが書いてあります。これらをブラウズしたければ、その名前を選んで `CMD-b` をタイプします (実際はどのテキストペインでもこの方法は使えます)。

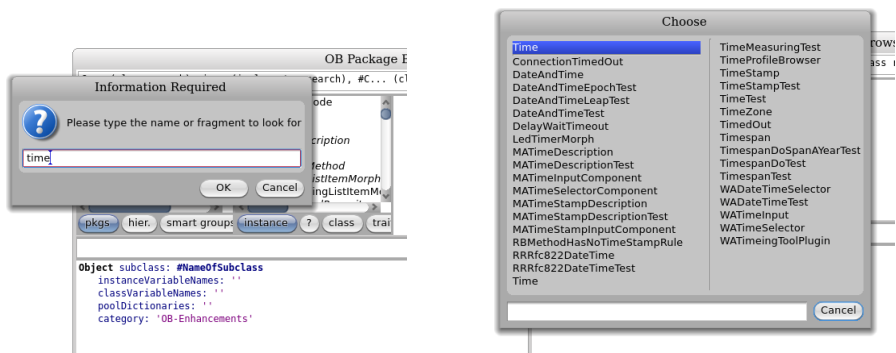


Figure 1.15: 名前でクラスを検索する。

検索ダイアログで完全なクラス名 (最初は大文字) を入力した場合、ブラウザは候補リストを表示することなくそのクラスを直接開きます。

## 1.9 メソッドを見つける

メソッド名やその一部が、クラス名よりも簡単に推測できることもあります。現在の時刻を知りたい場合、「now」というメソッドや「now」を含んだメソッドがあるのではと思うでしょう。しかしどこにあるのでしょうか? そうしたときはメソッド・ファインダが助けてくれます。

👉 **World** ▸ **Tools ...** ▸ **Method finder** を選択しましょう。左上のペインに「now」と入力して **accept** しましょう (または **RETURN** キーを押します)。

メソッド・ファインダは「now」を含むすべてのメソッド名のリストを表示します。now までスクロールするには、リストにカーソルを移動し「n」をタイプします (ちなみにこの方法はすべてのスクロールするリストで使えます)。「now」を選択しましょう。すると右側のペインにこの名前のメソッドを定義しているクラスの一覧が表示されます (図 1.16)。クラス名を選択するとそのクラスのブラウザが開きます。

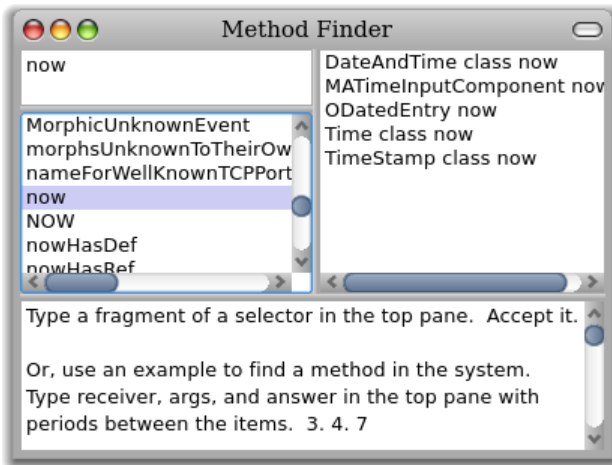


Figure 1.16: メソッド・ファインダ。now の定義を含んだすべてのクラスをリストしている。

時にはメソッドが存在することはわかっている、何という名前なのか見当がつかないこともあるでしょう。メソッド・ファインダはこのようなときにも役立ちます! 例えば文字列を大文字化するメソッドを見つけたかったとします。'eureka' を 'EUREKA' という具合にです。

👉 'eureka'. 'EUREKA' とメソッド・ファインダに入力して **RETURN** キーを押しましょう (図 1.17)。

メソッド・ファインダはお望みのメソッドを勧めてくれます。<sup>8</sup>

<sup>8</sup>もしウィンドウが開いて推奨されないメソッドと警告されても、あわてないでください。メ

メソッド・ファインダの右側ペインの行頭のアスタリスクは、望んだ結果を得るのに、そのメソッドが実際に使われたことを示します。今回 `String asUppercase` の先頭にアスタリスクが付いているので、クラス `String` で定義された `asUppercase` が `'eureka'`、`'EUREKA'` の変換に使われたとわかります。アスタリスクが付いていないメソッドは、単に名前が同じということで挙げられたものです。`Character »asUppercase` は実行されていません。`'eureka'` は `Character` オブジェクトではないからです。

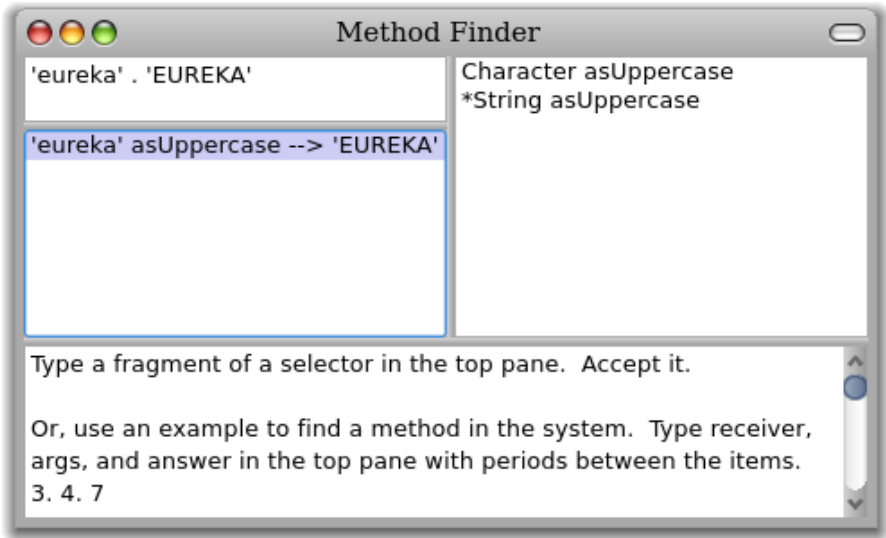


Figure 1.17: 例を使ってメソッドを見つける。

引数のあるメソッドについてもメソッド・ファインダを使うことができます。例えば二つの整数の最大公約数を求めるメソッドを探るときは、`25. 35. 5` と入力します。また複数の例を使って検索の範囲を絞りこむこともできます。下のペインのヘルプテキストに詳しい方法が書いてあります。

## 1.10 メソッドを新しく定義する

テスト駆動開発<sup>9</sup> (TDD) が登場してから、コードを書く方法は一変しました。TDD の背景にある思想は、振る舞いを期待するテストコードを、コード本体より先に書くというものです。TDD ではテストを書いてから、初めてそのテストを満足させるコードを書きます。

メソッド・ファインダは候補すべてを表示しようとするので、推奨されないメソッドが含まれることもあります。落ち着いて `Proceed` をクリックしてください。

<sup>9</sup>Kent Beck, *Test Driven Development: By Example*. Addison-Wesley, 2003, ISBN 0-321-14653-0.

「何かを大声で強調して言う」メソッドを書く、という課題があったとしましょう。これは正確にはどういう意味でしょう？ そうしたメソッドの名前としてふさわしいのは何でしょう？ 将来そのメソッドを保守するプログラマに、このメソッドが何をするのかを確実に伝えるにはどうすればいいでしょう？ 以下の例で、こうした疑問に答えていきます。

文字列「Don't panic」にメッセージ `shout` を送ったときの結果は、「DON'T PANIC!」となるべきである。

この例を、システムが理解できるようにテストメソッドに変換します。

### Method 1.1: `shout` メソッドのテスト

```
testShout
```

```
self assert: ('Don't panic' shout = 'DON'T PANIC!')
```

Pharo で新しくメソッドを作るにはどうすればよいでしょうか？ まずメソッドがどのクラスに属するかを決めなければいけません。今からテストしようとしている `shout` メソッドは `String` クラスのものとするので、テストは習慣として `StringTest` クラス内に作成します。

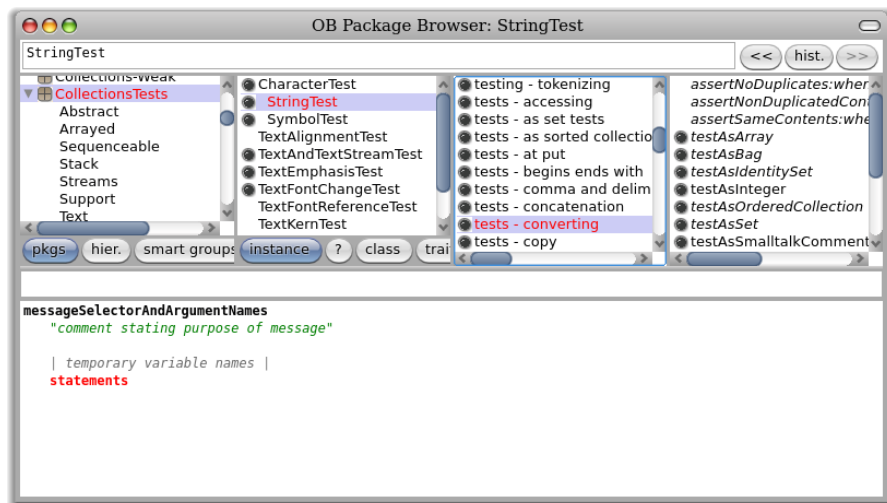


Figure 1.18: `StringTest` の新規メソッドのテンプレート。

🔍 `StringTest` のブラウザを開きましょう。 `testShout` のプロトコルとして `tests - converting` を選択しましょう (図 1.18)。下のペインで選択されているのはメソッドのテンプレートです。これを見れば `Smalltalk` のメソッドの大まかな姿がわかります。これを消して `Method 1.1` のコードを入力しましょう。

ブラウザにテキストを入力すると下のペインが赤く縁取りされることに注意してください。これにより、このペインの変更がまだ保存されていないことがわかります。それでは下のペインでアクションクリックして `accept (s)` を選択するか、もしくは `CMD-S` をタイプし、メソッドをコンパイル・保存してください。

仮想イメージでコードをアクセプトするのが初めてだとしたら、名前を入力するよう促されるはずですが、仮想イメージはたくさんの人が書いたコードからできているので、メソッドを作成・変更した人を記録しておくことは重要です。ファーストネームとラストネームを、空白なし、またはドットで区切って入力します。

`shout` というメソッドはまだないので、ブラウザはこれが本当に意図した名前であるかどうか確認します。そして他の可能性のある名前についても提案してきます (図 1.20)。単にミスタイプしたときにはこの機能は非常に助かりますが、今回は今から作ろうとするメソッドは `shout` で間違いないので、確認用メニューの一番上の項目を選択します (図 1.20)。

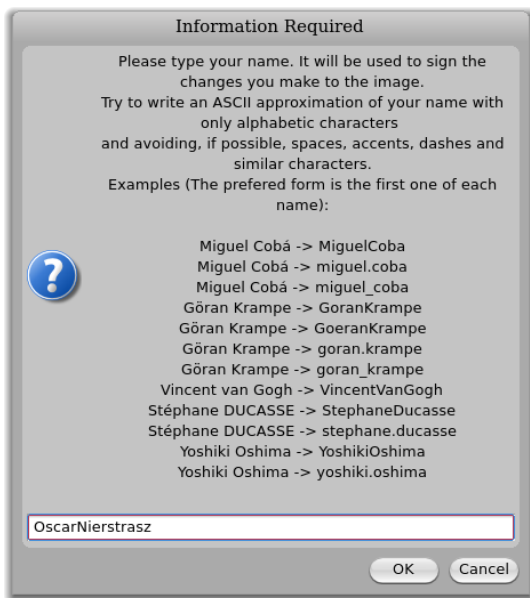


Figure 1.19: 名前を入力する。

🕒 今作ったテストを実行しましょう。 `World` から `SUnit TestRunner` を開いてください。

一番左に二つ並んだペインは、ブラウザの上部のペインに少し似ています。左側のペインにはクラスカテゴリのリストが表示されますが、テストクラスを含んだものに限られます。

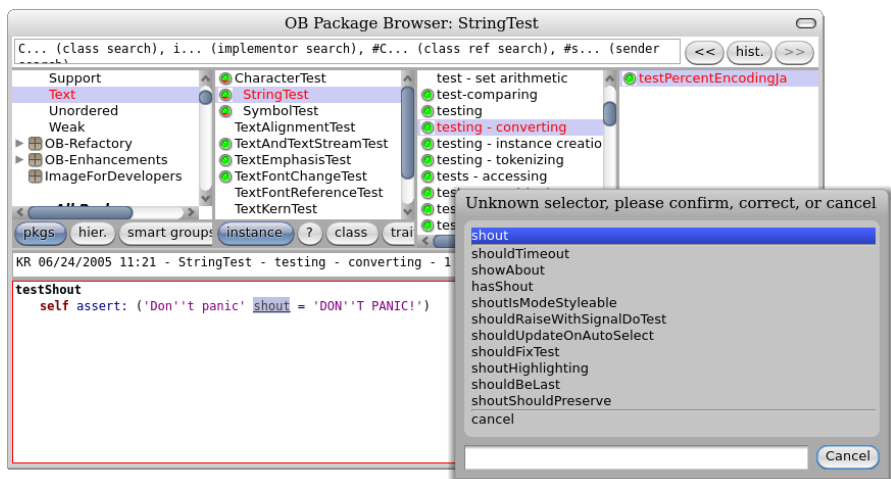


Figure 1.20: StringTest の testShout メソッドをアクセプトする。

🕒 CollectionsTests-Text を選択しましょう。右側のペインにはそのカテゴリのすべてのテストクラスが表示されます。その中に StringTest もあります。テストクラスの名前はもう選択されているので、Run Selected をクリックするとテストが実行されます。

テストを実行した結果が図 1.21 のように表示され、テストの実行時にエラーがあったことがわかります。エラーを起こしたテストのリストは右下のペインに表示されます。見てわかるように、StringTest»#testShout が原因です (Smalltalk では StringTest クラスの testShout メソッドを StringTest»#testShout と表記します)。StringTest»#testShout をクリックすると再度そのテストだけが実行され、「MessageNotUnderstood: ByteString»shout」というウィンドウが開きます。

エラーメッセージとともに開いたこのウィンドウは、Smalltalk のデバッガです (図 1.22)。デバッガとその使い方については第 6 章で解説します。

このエラーは、もちろん期待した通りのものです。文字列が shout するためのメソッドをまだ書いていないので、テストを実行するとエラーが発生するのは当然です。それでもなお、テストが失敗することを確認するのは良い習慣です。それによって、テストの仕組みが正しく設定され新しいテストが実行されたことを確認できるからです。いったんエラーを確認したら、テストの実行をやめて (Abandon) デバッガウィンドウを閉じることができます。ちなみにたいていの Smalltalk では、まだないメソッドを Create を使ってその場で書いてしまうこともできます。デバッガの中で新しく作成されたメソッドを編集し、テストを継続実行 (Proceed) できます。

ではテストが成功するようにメソッドを作成しましょう!

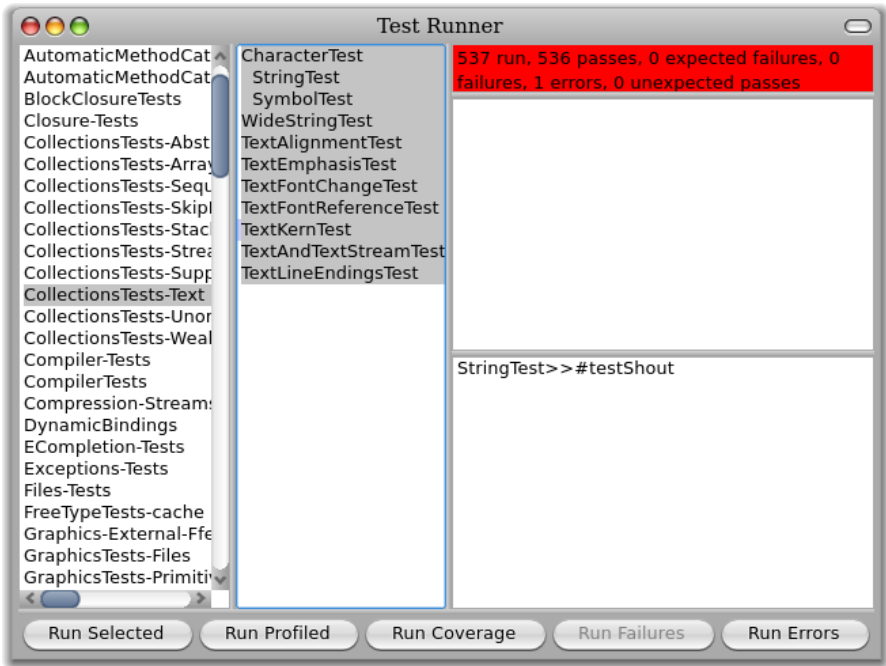


Figure 1.21: StringTest を実行する。

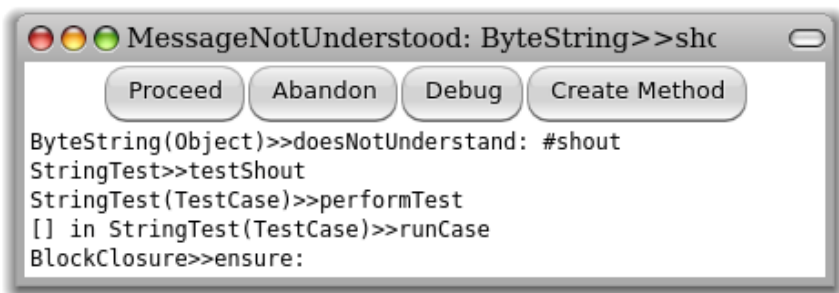


Figure 1.22: (プリ) デバッガ

🕒 ブラウザで `String` クラスを選択し、`converting` プロトコルを選択します。`Method 1.2` のテキストでメソッド作成テンプレートを上書きし `accept` しましょう (注: ↑のところには`^`を入力します)。

Method 1.2: `shout` メソッド



shout

↑self asUppercase, '!'

カンマは、文字列連結を意味します。つまりこのメソッドでは、shout を受け取った String オブジェクトを大文字にし、末尾に感嘆符を付け足します。Pharo では↑以下の式がメソッドの戻り値になります。ここでは戻り値は、大文字化になり感嘆符が付いた新しい文字列です。

このメソッドは思い通り動くでしょうか? もう一度テストを実行してどうなるか確認してみましょう。

🔍 テスト・ランナーで **Run Selected** を再びクリックしましょう。今度は緑のバーが出て、テストがすべて失敗もエラーもなく実行できたことを示すテキストが表示されます。

緑のバーが出たら、成果を保存して一休みするとよいでしょう!

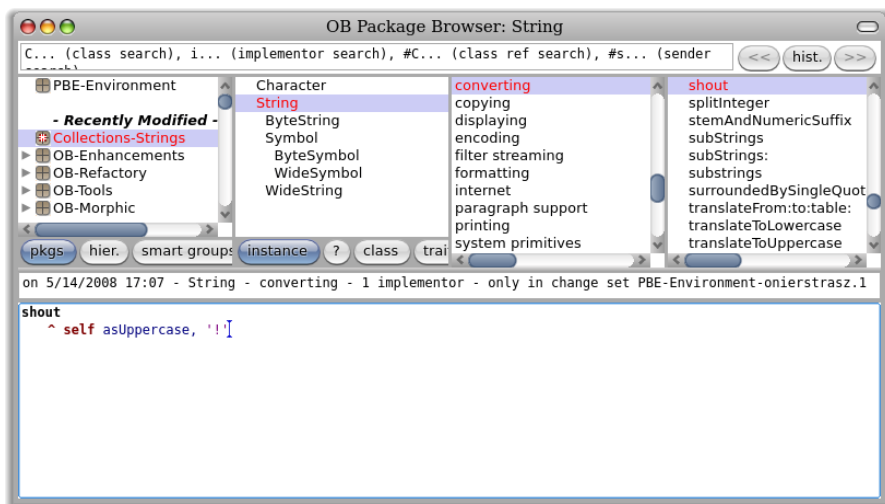


Figure 1.23: String クラスで定義された shout メソッド。

## 1.11 まとめ

この章では Pharo の環境を紹介し、ブラウザ、メソッド・ファインダ、テスト・ランナーと言った主要なツールの使い方を学びました。Pharo の文法についても、全部ではありませんが少しだけ学びました。

- 実行中の Pharo システムは バーチャルマシン、ソースファイル、イメージファイル および チェンジファイルからなります。この中で最後の二つ

だけが更新されます。これらは実行中のシステムのスナップショットを記録します。

- Pharo 仮想イメージをロードすると、実行中のオブジェクトも含め、保存したときとまったく同じ状態が再現されます。
- Pharo は 3 ボタンマウスでクリック、アクションクリック、メタクリックするように設計されています。3 ボタンマウスがなくても、キーボードの修飾キーを使えば同じことができます。
- Pharo の背景部分を クリック するとワールドメニューが表示されます。ワールドメニューからは様々なツールが起動できます。
- ワークスペースは、コードの断片を書いて評価するためのツールです。ワークスペースに、任意のテキストを書いておくこともできます。
- コードを評価するのに、ワークスペースや他のツールのテキストペイン上でキーボードショートカットを使うことができます。最も重要なのは `do it (CMD-d)`、`print it (CMD-p)`、`inspect it (CMD-i)`、`explore it (CMD-I)` そして `browse it (CMD-b)` です。
- ブラウザは、Pharo のコードをブラウズしたり新たにコードを書いたりするための最も重要なツールです。
- テスト・ランナーは、ユニットテストを実行するためのツールです。テスト・ランナーはテスト駆動開発もサポートします。

# Chapter 2

## 最初のアプリケーション

この章では、簡単なゲーム : *Lights Out*<sup>1</sup> を作成します。ゲームの作成を通じて、Pharo プログラマがプログラミングやデバッグに使用するツール群や、他の開発者とプログラムをやりとりする方法について体験していきます。この章で扱う開発ツールは、ブラウザ、オブジェクト・インスペクタ、デバッガ、Monticello パッケージ ブラウザ です。Smalltalk による開発はとても効率的です。Smalltalk はとても簡潔なプログラミング言語です。また、開発ツールは言語と非常によく統合されています。そのため、開発の手順に手間取ることなく、コードを書くことに多くの時間を使うことができるでしょう。

### 2.1 Lights Out ゲーム

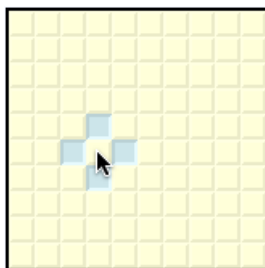


Figure 2.1: *Lights Out* のゲーム盤。ユーザは盤上のセルをマウスでクリックするだけです。


Pharo の開発ツールの使い方を体験するために、これから *Lights Out* という簡単なゲームを作っていきます。ゲーム盤は図 2.1 のように、淡黄色をしたセルの四角形の配列で構成されます。セルをマウスでクリックすると、周囲の

<sup>1</sup>[http://en.wikipedia.org/wiki/Lights\\_Out\\_\(game\)](http://en.wikipedia.org/wiki/Lights_Out_(game))

四つのセルが青色に変わります。もう一度クリックすると、それらはまた淡黄色に戻ります。このゲームの目的は、できるだけ多くのセルを青色に変えることです。図 2.1 で示すように、ゲームはゲーム盤と、100 個のセルからなる 2 種類のオブジェクトで構成されています。Pharo のコードとしては、クラス二つでゲームを実装しています。一つはゲームを表すクラス、もう一つはセルを表すクラスです。それでは、Pharo の開発ツールを使い、どのようにこれらのクラスを定義していくか、見ていくことにしましょう。

## 2.2 パッケージを作成する

第 1 章のブラウザの紹介で、クラスやメソッドのブラウザの仕方や、新たなメソッドを定義する方法を学びました。ここではパッケージ、カテゴリ、クラスの作り方を学びます。

 ブラウザを開きパッケージペインでアクションをクリックし、`create package` を選択します。<sup>2</sup>

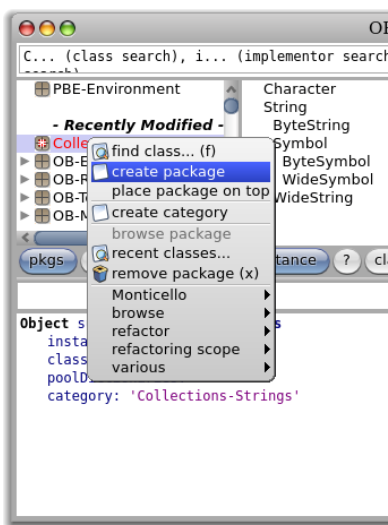


Figure 2.2: パッケージを追加します。

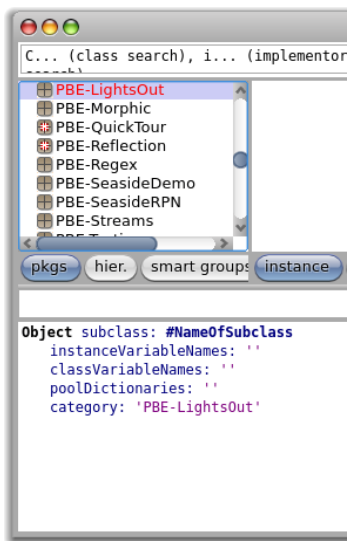


Figure 2.3: クラステンプレート。

ダイアログボックスに新たなパッケージ名 (*PBE-LightsOut* とします) を入力し、`accept` をクリックしてください (または単にリターンキーを押してください)

<sup>2</sup>パッケージ・ブラウザが標準ブラウザとしてインストールされているものとします。もし図 2.2 で示したブラウザでない場合は、デフォルトのブラウザを変更する必要があります。FAQ 5, p. 326 参照。

い)。すると新たなパッケージができ上がり、パッケージ一覧にアルファベット順となって表示されます。

## 2.3 LOCell クラスを定義する

当然ですが、新しいパッケージにはまだ何のクラスもありません。しかし、新たなクラスを作りやすいように、クラスのテンプレートが、メインの編集ペインに自動的に表示されます (図 2.3 参照)。


このテンプレートは、Objectクラスに `NameOfSubClass` というサブクラスを作成するという、Smalltalk のメッセージ式になっています。新しいクラスは何の変数も持たず、*PBE-LightsOut* カテゴリに属するようになっています。

### カテゴリとパッケージについて

従来の Smalltalk ではカテゴリはありましたが、パッケージというものはありませんでした。この二つの違いは何でしょうか。カテゴリとは Smalltalk 関連するクラスをイメージファイル内で集約しているだけのものです。パッケージとは関係するクラスと拡張メソッドを集めたもので、Monticello のようなツールでバージョン管理できるものです。慣習としてパッケージ名とカテゴリ名には同じ名前を使います。通常は二つの違いを気にする必要はありません。しかし、これらの用語の違いが重要なこともあるため、本書では注意深く使い分けています。Monticello を使い始めるところで、より詳しく見ていくことにしましょう。

### 新しいクラスを作成する

テンプレートを修正し、目的に合ったクラスを作ってみましょう。

 クラス作成のテンプレートを修正する手順は以下の通りです:

- Object を SimpleSwitchMorph に書き換えます。
- NameOfSubClass を LOCell に書き換えます。
- インスタンス変数のリストに `mouseAction` を追加します。

結果は Class 2.1 の通りです。

## Class 2.1: LOCellクラスを定義する

```
SimpleSwitchMorph subclass: #LOCell
  instanceVariableNames: 'mouseAction'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE -LightsOut'
```

この新しいクラス定義は、SimpleSwitchMorphという既存のクラスに対し、LOCellというサブクラスを作るようにメッセージを送る Smalltalk の式になっています (実際には LOCellクラスはまだ存在していないため、クラス名を表すシンボル #LOCellを引数で渡しています)。また、このメッセージには新たなクラスのインスタンスが mouseActionインスタンス変数を持つという指定も含まれています。mouseActionは、セル上でマウスがクリックされたときの振る舞いを定義するための、インスタンス変数となります。

この時点ではまだ何も作られてはいません。クラスプレートペインの境界線が赤に変わっていることに注目してください (図 2.4)。これはまだ変更が保存されていないことを意味しています。実際にこのメッセージを送るには、変更を **accept** する必要があります。

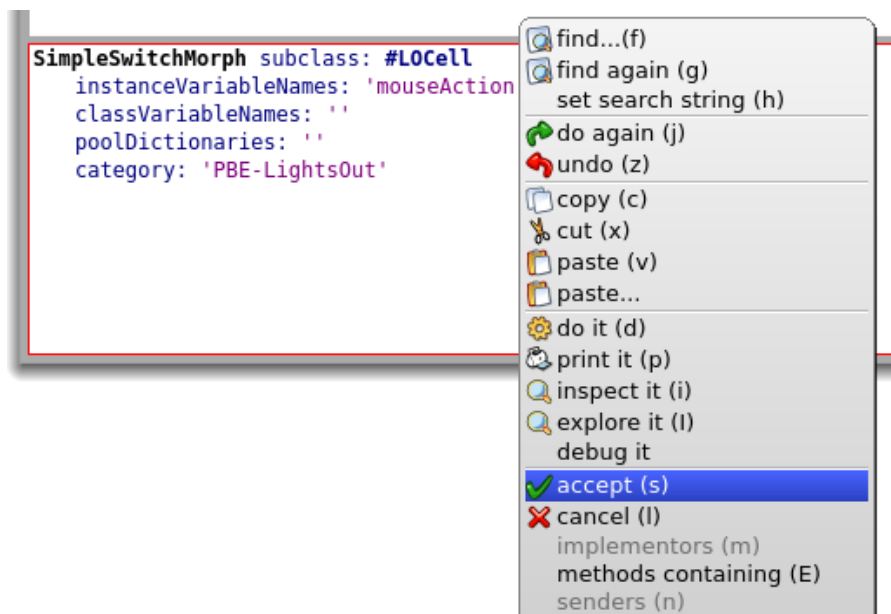


Figure 2.4: クラス作成テンプレート。

🕒 新しいクラス定義を「了解 (accept)」します

action-click して **accept** を選択、もしくはショートカットの **CMD-S** (save) を

実行してください。SimpleSwitchMorphにメッセージが送られ、新しいクラスがコンパイルされます。クラス定義を `accept` すると、クラスができてブラウザのクラスペインに表示されます (図 2.5)。編集ペインにはクラス定義が表示され、その下にある小さなペインでクラスの目的を説明するように促されます<sup>3</sup>。ここへ記述する内容はクラスコメントと呼ばれ、他のプログラマにクラスの大まかな目的を伝える重要な手段となります。Smalltalk プログラマはコードそのものの読みやすさを重視するため、メソッド中に詳細なコメントを書くことはまれです。ここでの哲学は、コードの意図はコード自身に語らせよということです (もしそうっていない場合は、そうなるまでリファクタリングしましょう! )。

コメントには、クラスの詳細すぎる説明ではなく、後でプログラマがこのクラスを見ていくべきかを判断できる、大まかな目的を簡潔に記述するようにしましょう。

🕒 **LOCell** クラスのコメントを入力し、`accept` してください。後に内容を変更することもできます。

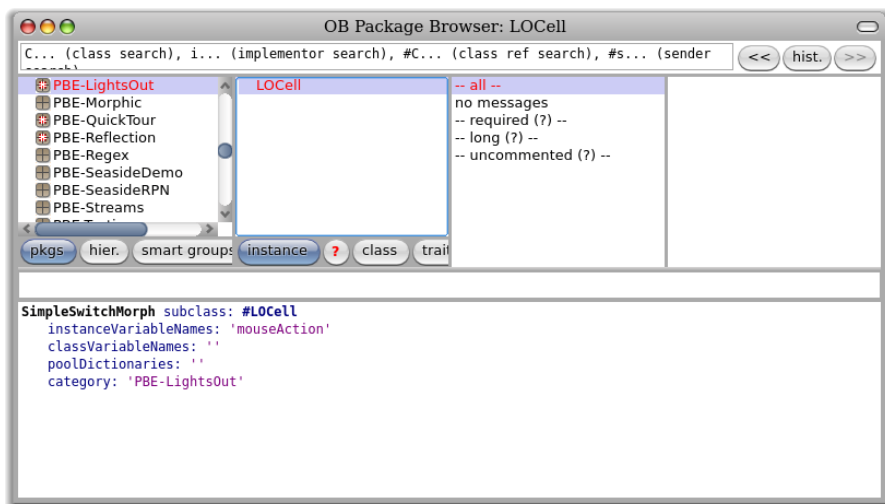



Figure 2.5: 新しくできたLOCellクラス

## 2.4 クラスにメソッドを追加する

それではこのクラスにメソッドをいくつか追加してみましょう。

<sup>3</sup> 訳者が確認した環境 (Pharo by Example Image および 1.3 環境) ではそのようなペインはありません。代わりに、クラスブラウザ中にある「?」ボタンを押すことでクラスの目的を書くペインが表示されます。また、クラスコメントが書かれていない場合には、その「?」の箇所が赤くなりクラスの目的を書くことを促す挙動となっています。

 プロトコルペインにある `-all-` プロトコルを選択します。

編集ペインにメソッド作成用のテンプレートが表示されます。それを選択し、Method 2.2 で示した内容に書き換えてください。

#### Method 2.2: LOCellのインスタンスを初期化する

```

1 initialize
2   super initialize.
3   self label: "".
4   self borderWidth: 2.
5   bounds := 0@0 corner: 16@16.
6   offColor := Color paleYellow.
7   onColor := Color paleBlue darker.
8   self useSquareCorners.
9   self turnOff

```

3行目”の文字は、間に何の文字も挟まない引用符です。二重引用符ではないことに注意してください。”は空の文字列を表します。

 このメソッドの定義を *accept* します。

上記のコードは何をしているのでしょうか？ここでは詳細については触れずに(本の残りの部分はそのためにあるのです！)、簡単に内容を見ていきます。それでは1行ずつ進んでいきましょう。

`initialize` というメソッドに注目してください。この名前はとても重要です。`initialize` というメソッドは、慣習的にオブジェクトが生成された直後に呼び出されます。つまり、`LOCell new`が評価されると `initialize` メッセージが新しく作られたオブジェクトに自動的に送られます。`initialize` メソッドはオブジェクトの状態、主にそのインスタンス変数を設定するために使われます。

このメソッドでは、スーパークラスである `SimpleSwitchMorph` の `initialize` メソッドをまず実行しています(2行目)。スーパークラスの `initialize` メソッドの実行により、継承した状態が適切に初期化されるということを期待しています。何かの処理をする前には、スーパークラスの `initialize` メソッドを呼び出して、継承した状態を初期化しておくといよいでしょう。`SimpleSwitchMorph` の `initialize` メソッドが実際に何をするかは知らなくともかまいません。不明瞭な状態のまま処理を始める危険を冒すよりは、スーパークラスの `initialize` メソッドを呼び出すことで、インスタンス変数に妥当な初期値が設定されるようにした方がよいのです。

メソッドの残りの部分でこのオブジェクトの状態をさらに設定しています。例えば、`self label: ""` をオブジェクトに送ることで、オブジェクトのラベルに空文字列を設定しています。

`0@0 corner: 16@16` については、おそらく説明が必要でしょう。`0@0` は、 $x, y$  座標の両方が `0` に設定された `Point` オブジェクトを表しています。実際には、`0@0` は、数値 `0` に `@` メッセージを引数 `0` で送るといいう、メッセージ送信になっています。その結果、数値 `0` は、`Point` クラスに座標 `(0,0)` のインスタンスを作るよう指示します。新たに作られた `Point` オブジェクトに `corner: 16@16` メッセージ



が送られ、0@0と16@16の角を持つRectangleオブジェクトが作られます。このRectangleオブジェクトは、スーパークラスから継承されたbounds変数に代入されます。Pharoの画面の座標系の原点は左上で、下に向かってy座標の値が増えることに注意してください。

メソッドの残りの部分について説明の必要はないでしょう。Smalltalkをうまく書く秘訣は、コードが英語のように読めるように、良いメソッド名を付けることです。オブジェクトが自分自身に語りかけ、“自分は直角を使う!”、“自分はスイッチを切る!”と言っているイメージを持つことができるはずです。

## 2.5 オブジェクトをインスペクトする

LOCellオブジェクトを作ってインスペクトすることで、先ほど書いたコードの効果を確認できます。

 ワークスペースを開いて、LOCell newと打ち、inspect itしてください。

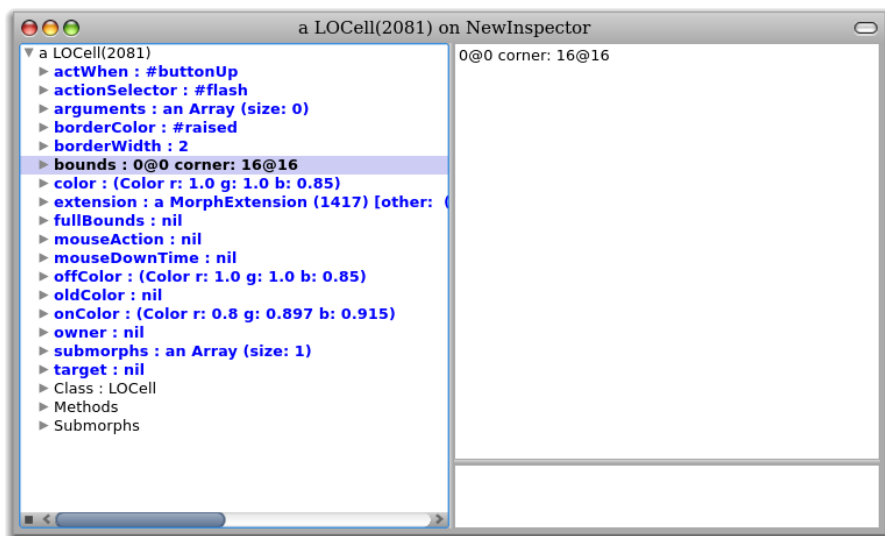


Figure 2.6: LOCell オブジェクトを調べるために使用したインスペクタ。

inspectorの左側のペインには、インスペクトしたオブジェクトのインスタンス変数のリストが表示されます。どれか一つ(例えばbounds)を選んでみてください。右側のペインにそのインスタンス変数の値が表示されます。

インスペクタの下側にあるペインは小さなワークスペースです。このワークスペースでは、選択中のオブジェクトが擬似変数selfで参照できるため、とても便利です。

🔍 インスペクタウィンドウのルートにある `LOCell` を選択してください。下部のペインに、`self bounds: (200@200 corner: 250@250)` と打ち `do it` しましょう。インスペクタ上で `bounds` の値が変化します。次にワークスペース部分に `self openInWorld` と打ち込み `do it` してください。

画面の左上の端、つまり `bounds` が示している位置にセルが表示されます。セル上でメタクリックをすると `halo` が現れます。右上の左隣にある茶色いハンドルでセルを移動したり、右下にある黄色のハンドルでセルをリサイズしてみましょう。`bounds` の値の変化を、インスペクタから確認できます。(新たな `bounds` の値を見るために `refresh` アクションクリックが必要かもしれません。)

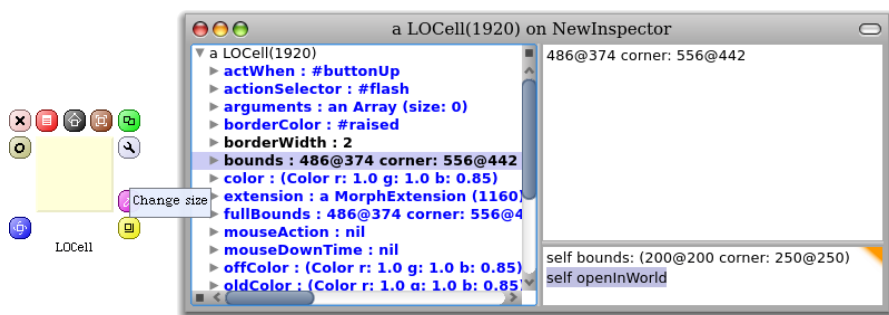


Figure 2.7: セルのリサイズ。

🔍 ピンク色のハンドル `x` をクリックしてセルを消してください。

## 2.6 LOGame クラスを定義する

ゲームに必要な、もう一つのクラスを作っていきます。クラス名は `LOGame` とします。

🔍 ブラウザのメインウィンドウに、クラス定義テンプレートを表示させます。

パッケージ名をクリックすることで、クラス定義テンプレートを表示させることができます。以下のようにコードを書いたら、`accept` してください。

### Class 2.3: `LOGame` クラスの定義

```
BorderedMorph subclass: #LOGame
instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: ""
category: 'PBE-LightsOut'
```

LOGameは BorderedMorph のサブクラスとしました。Morph は Pharo のグラフィック図形すべてのスーパークラスで、BorderedMorphは境界線を扱える Morph になります。2 行目にある引用符の間にはインスタンス変数の名前を入れることができますが、ここでは空のままにしておきましょう。

LOGameの initialize メソッドを定義しましょう。

🔗 以下の内容を LOGameのメソッドとして Browser に書き込み、accept してください。

### Method 2.4: ゲームの初期化

```

1 initialize
2   | sampleCell width height n |
3   super initialize.
4   n := self cellsPerSide.
5   sampleCell := LOCell new.
6   width := sampleCell width.
7   height := sampleCell height.
8   self bounds: (5@5 extent: ((width*n) @(height*n)) + (2 * self borderWidth)).
9   cells := Matrix new: n tabulate: [ :i j | self newCellAt: i at: j].
    
```

Pharo はいくつかの用語が認識できないと警告をしてくるはずですが、まず Pharo は、cellsPerSideメッセージが認識できないため、スペルミスとして修正案をいくつか提示してきます。

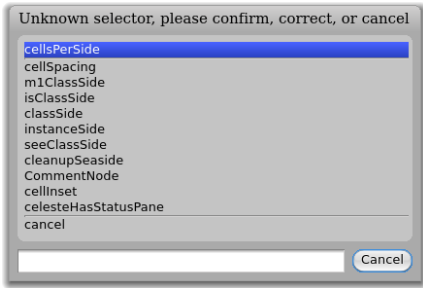


Figure 2.8: Pharo が未知のセレクタを検出。

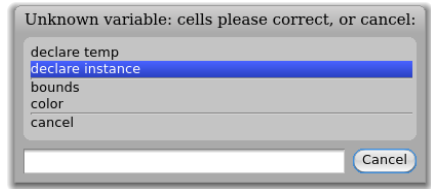



Figure 2.9: 新しいインスタンス変数を宣言。

しかし、cellsPerSideはスペルミスではありません。単にまだメソッドを定義していないだけです。メソッドはこのすぐ後で定義することにしましょう。

🔗 メニューから最初のアイテムを選択し、cellsPerSideがスペルミスではないことを確認します。

次に Pharo は cellsが認識できないと警告してきます。Pharo はこの問題を解消するための案をいくつか提示してきます。

 cellをインスタンス変数として扱いたいので、`declare instance`を選択します。

最後に Pharo は、最終行で使われている `newCellAt:at` メッセージについて警告してきます。もちろんこれもスペルミスではないので、先ほどと同様に流してください。

さて、もう一度 `LOGame` のクラス定義を見てみましょう (`instance` ボタンをクリックします)。クラス定義が更新され、インスタンス変数 `cells` が含まれるようになったことを、ブラウザで確認できるはずです。

それでは、この `initialize` メソッドの中身を見ていきましょう。| `sampleCell width height n` | と書かれた行では、四つの一時変数を宣言しています。ここで宣言された変数は、スコープがこのメソッドに限定されるため、一時変数と呼ばれます。説明的な名前を持った一時変数を使うことで、コードをより読みやすくなります。`Smalltalk` では定数と変数を区別するための特別な構文はありません。また、ここで宣言された四つの“変数”は、実際には定数と変わりありません。4-7 行目で一度だけ値が設定されています。

ゲーム盤には必要な数だけのセルと、それらの境界を表示するための十分な大きさが必要ですが、セルの数がどれぐらいになるかは今はまだ判断できません。そこで、`cellsPerSide` というメソッドを別に定義して、セルの数を決定する責務をそちらに委譲することにします。メソッドを定義する前に `cellsPerSide` をメッセージ送信するようにしたため、`initialize` メソッドを `accept` するとき `Pharo` から “confirm, correct, or cancel” と警告されることになってしまいましたが、ここで避けてはいけません。まだ定義していないメソッドを使いながらメソッドを書いていくというのは、実のところとても良い習慣です。なぜなら、`initialize` メソッドを書くまで私たちはそうしたメソッドが必要となるかわかりませんでしたし、その時点になって初めて、意味のある名前をメソッドに付けることができ、思考を中断せずに進んでいくことができるからです。

4 行目でこのメソッドが使われています。`Smalltalk` 式 `self cellsPerSide` は `cellsPerSide` メッセージを `self`、つまりオブジェクト自身に送ります。この結果、ゲーム盤の一辺に必要なセルの数が `n` に割り当てられました。

次の 3 行では、新たな `LOCell` オブジェクトを作成し、ゲーム盤の幅と高さを、適切な一時変数に代入しています。

8 行目では自身の `bounds` を設定しています。詳細は今は気にせず、括弧内の式で原点 (すなわち左上隅 (5,5)) と、右下隅からなる、指定した数のセルを置くのに十分な長方形が作られると理解してください。

最後の行では、`LOGame` オブジェクトのインスタンス変数 `cells` に、適切な数の行と列を持つ `Matrix` オブジェクトを生成して代入しています。`Matrix` オブジェクトの生成は、`new:tabulate:` メッセージを `Matrix` クラス (このクラスもちろんオブジェクトです。そのためメッセージを送ることができます。) に送ることで実現しています。`new:tabulate:` は、二つのコロン (:) を持つことから二つの引数を取ることがわかります。引数はコロンの後に書きます。もし、引数を括弧内に一緒に書く言語しか使ったことがなければ、最初はこの書き方に戸惑うかもしれません。しかし、これはただの構文です。あわてることはありません。そ

のうちに、メソッド名が引数の役割を説明してくれるわかりやすい構文だということがわかるでしょう。例えば次の式、`Matrix rows: 5 columns: 2`なら、`Matrix`は2行5列ではなく5行2列であることは明白です。

`Matrix new: n tabulate: [:i:j| self newCellAt: i at: j]`によって  $n \times n$  の行列とその要素を初期化しています。各要素の初期値はその座標に依存します。 $(i,j)^{\text{th}}$  の位置にある要素は、`self newCellAt: i at: j`式を評価した結果で初期化されます。

## 2.7 メソッドをプロトコルにまとめる

メソッドの定義を続ける前に、ブラウザ上部にある3番目のペインを見てみましょう。1番目のペインでクラスをパッケージに分類したように、3番目のペインではメソッドの分類ができます。これにより4番目のペインが大量のメソッドで埋もれてしまうのを防ぐことができます。こうしたメソッドの分類は「プロトコル」と呼ばれます。

クラスにメソッドが少ししかないときは、プロトコルによる階層は必要ありません。そうしたクラスのために、ブラウザにはすべてのメソッドが含まれる仮想プロトコル `all` が用意されています。

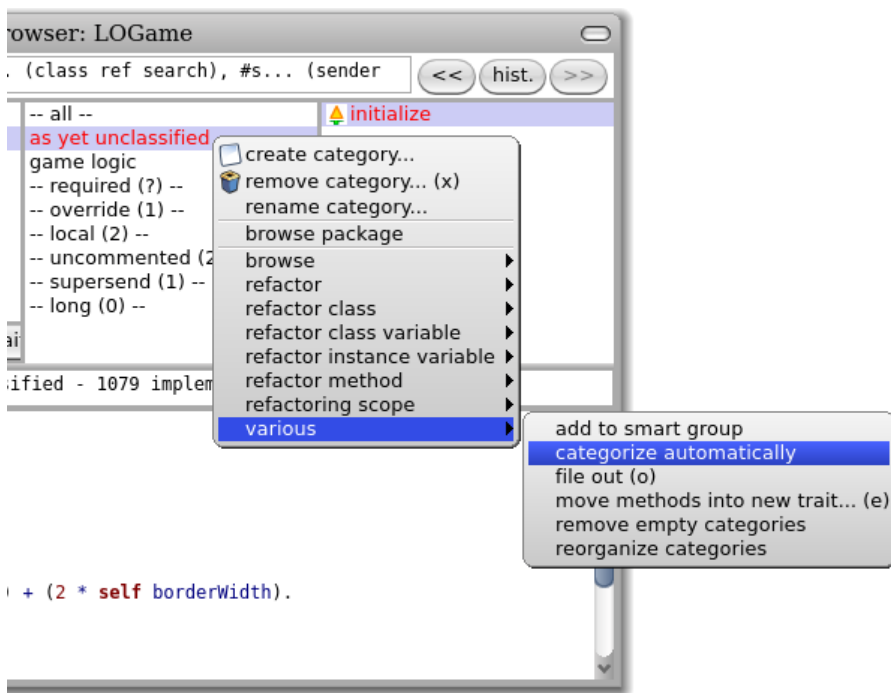


Figure 2.10: 分類されていないメソッドを自動的に分類。

本書の例に従って作業をしているなら、三つ目のペインには *as yet unclassified* (まだ分類されていない) というプロトコルが含まれているかもしれません。

🔍 プロトコルペインをアクションクリックしてください。  
`various ▷ categorize automatically` を選択すると、`initialization` プロトコルが新しくでき、`initialize`メソッドが移ります。

Pharo はどうやってプロトコルを定めるのでしょうか。一般的には Pharo 側で判断することはできません。しかしこの例では、スーパークラスにも `initialize`メソッドが定義されているため、Pharo は `initialize`メソッドを、オーバーライド元のメソッドと同じプロトコルに分類します。

**表記規則** Smalltalker は、メソッドが属するクラスを識別するために “>>” という表記をよく使用します。例えば `LOGame` クラスの `cellsPerSide` メソッドは `LOGame>>cellsPerSide` と表します。この表記が Smalltalk 式ではないことを明示するため、本書では代わりに特殊なシンボル»を使うことにします。先ほどのメソッドであれば、文中では `LOGame»cellsPerSide` と表します。

以後、本書ではこの記法でメソッド名を記述します。もちろん、実際に `Browser` 上でコードを打ち込むときはクラス名や»を打つ必要はありません。クラスペインで適切なクラスが選択されていることを確認するだけで十分です。

それでは、`LOGame»initialize`メソッドで使われている他の二つのメソッドを定義していきましょう。両方とも `initialization` プロトコルに入れることにします。

#### Method 2.5: 定数メソッド

```
LOGame»cellsPerSide
  "ゲーム盤の辺に並ぶセルの数"
  ↑ 10
```

このメソッドは非常にシンプルで、単に 10 という値を返すだけです。このように値をメソッドとして表現しておく、プログラムが成長してこの値が単純に定まらなくなった場合に、値を計算して返すようにメソッドを変更して対処できるという利点があります。

#### Method 2.6: 初期化補助メソッド

```
LOGame»newCellAt: i at: j
  "位置(i,j)にセルを作成し、適切な画面の位置に追加。新しいセルを返す"
  | c origin |
  c := LOCell new.
  origin := self innerBounds origin.
  self addMorph: c.
  c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
  c mouseAction: [self toggleNeighboursOfCellAt: i at: j]
```

🔍 `LOGame»cellsPerSide`メソッドと `LOGame»newCellAt:at:`メソッドを追加してください。


新たに出てきた `toggleNeighboursOfCellAt:at:` セレクタと `mouseAction:` セレクタのスペルをチェックしましょう。

Method 2.6 では、Matrix (行列) の (i,j) の位置に新しい `LOCell` を生成しています。最後の行ではセルの `mouseAction` として `block[self toggleNeighboursOfCellAt: i at: j].` を設定しています。これはマウスをクリックしたときのコールバックとなります。対応するメソッドを定義しましょう。

### Method 2.7: コールバックメソッド

```
LOGame>toggleNeighboursOfCellAt: i at: j
(i > 1) ifTrue: [(cells at: i - 1 at: j) toggleState].
(i < self cellsPerSide) ifTrue: [(cells at: i + 1 at: j) toggleState].
(j > 1) ifTrue: [(cells at: i at: j - 1) toggleState].
(j < self cellsPerSide) ifTrue: [(cells at: i at: j + 1) toggleState].
```

Method 2.7では、セルの (i, j) の東西南北の位置に隣接する四つのセルの状態を切り替えます。ここで唯一難しいのはゲーム盤が有限ということです。そのため、隣接するセルが存在するかどうかを、状態を切り替える前に確認しなければなりません。

 このメソッドを `game logic` という新たなプロトコルに置きます。(プロトコルペイン上で、アクションをクリックして新たなプロトコルを作ってください)

メソッドを移動するには、名前の上でクリックし、新たに作成したプロトコルまでドラッグします (図 2.11)。

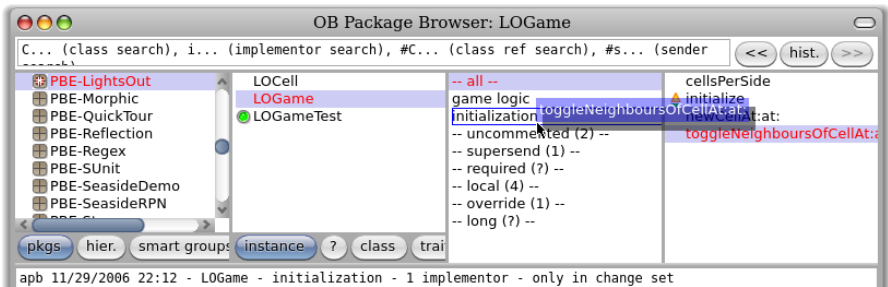


Figure 2.11: メソッドをプロトコルへ持って行く。

Lights Out ゲームを完成させるには、さらにマウスイベントを操作するための二つのメソッドを、`LOCell` クラスに定義する必要があります。

### Method 2.8: 典型的なセッターメソッド

```
LOCell>mouseAction: aBlock
↑ mouseAction := aBlock
```

Method 2.8 ではセルの `mouseAction` 変数に引数の値を設定して、値を返しているだけです。このようにインスタンス変数の値を変更するメソッドはセッターメソッドと呼ばれています。また、インスタンス変数の現在の値を返すメソッドはゲッターメソッドと呼ばれます。

もし他のプログラミング言語でゲッターメソッドとセッターメソッドを使うことに慣れていたら、メソッド名を `setmouseAction` や `getmouseAction` としたくなるかもしれませんが、しかし Smalltalk では違い流儀です。ゲッターメソッドの名前は常に取得するインスタンス変数と同じ名前にします。セッターメソッド名も同じですが、`mouseAction` に対して `mouseAction:` のように末尾を “:” とします。

セッターメソッドとゲッターメソッドをあわせてアクセサメソッドと呼びます。また、これらは慣習として *accessing* プロトコルに置かれます。Smalltalk ではすべてのインスタンス変数はプライベートなものとしてオブジェクトが保持します。そのため、Smalltalk で、他のオブジェクトの変数を読み書きするには、アクセサメソッドを介するしかありません<sup>4</sup>。

🔍 LOCellクラスに移動し、LOCell»mouseAction:を定義してください。定義が終わったら、そのメソッドを *accessing* プロトコルに置いてください。

最後に `mouseUp:` メソッドを定義する必要があります。このメソッドは画面のセル上でマウスボタンを離れたときに、GUI フレームワークから自動的に呼び出されます。

### Method 2.9: イベント ハンドラ

```
LOCell»mouseUp: anEvent
mouseAction value
```

🔍 LOCell»mouseUp:を追加し、`categorize automatically` を実行してください。

このメソッドは `value` メッセージを `mouseAction` インスタンス変数にバインドされたオブジェクトに送ります。LOGame»newCellAt: i at: jで以下のコードを `mouseAction:` に指定したことを思い出してください。

```
[self toggleNeighboursOfCellAt: i at: j]
```

`value` メッセージを送ると、この部分が評価されてセルの状態が切り替わることとなります。

## 2.8 コードを実行してみましょう

これで Lights Out ゲームができあがりました！

手順に沿って進んできたのなら、二つのクラスと七つのメソッドで構成された、このゲームで遊ぶことができるはずです。

<sup>4</sup>厳密にはインスタンス変数はサブクラスからもアクセスできます。



🌀 *workspace* 上で `LOGame new openInWorld`と打ち、`do it`してください。

ゲームが始まります。セルをクリックすると動作を確認できるでしょう。

ところがクリックすると、`PreDebugWindow` と呼ばれるノーティファイアが、エラーメッセージと共に表示されました。理論上は動くはずなのですが.....  
図 2.12 のように、このウィンドウは `MessageNotUnderstood: LOGame>>toggleState` と言ってきています。

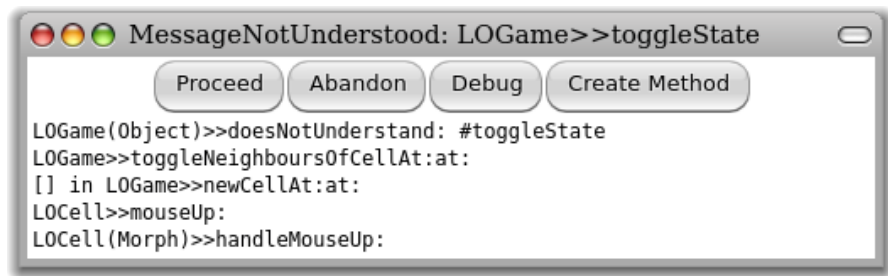


Figure 2.12: cell をクリックしたときにゲームでバグが発生

何が起こったのでしょうか? 原因を探るため、Smalltalk ではさらに強力なツールとなっている、`debugger` を使しましょう。

🌀 ノーティファイアにある `debug` ボタンをクリックしてください。

デバッガが表示されるはずですが、ウィンドウの上部には実行スタックが表示されます。実行スタックには実行されたメソッドが表示されています。その中の一つを選ぶと、真ん中のペインにメソッド内の Smalltalk コードが表示され、エラーを引き起こした部分がハイライトされます。

🌀 (上部の)`LOGame>>toggleNeighboursOfCellAt:at:`と表示されている行をクリックしてください。

デバッガ上で、エラーが発生したメソッドの実行コンテキストを確認することができます (図 2.13)。

デバッガの下側には二つの小さなインスペクタが表示されています。左側のペインでは、レシーバ、つまり選択中のメソッドを実行するためのメッセージを受け取った、オブジェクトの中身をインスペクトできます。インスタンス変数の値は、ここで確認できます。右側のペインでは現在のメソッドの実行状態を示すオブジェクトをインスペクトできます。メソッドの引数や一時変数の値などはこちらで確認します。

デバッガを使うことで、コードを 1 行ずつ実行していくことができます。また、引数や一時変数にバインドされたオブジェクトを見たり、ワークスペース同様に式を評価したりすることも可能です。通常のデバッガと比べて最も驚くのは、デバッグ中にコードを書き換えられることです。ほとんどの時間をブラ

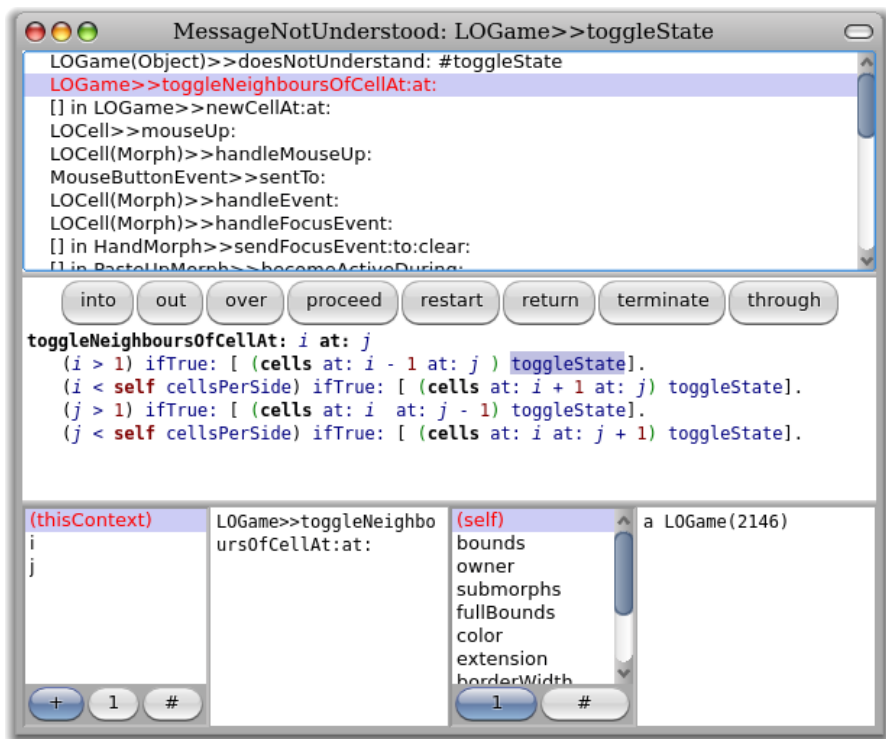


Figure 2.13: デバッガ上で `toggleNeighboursOfCell:at:` メソッドを選択する。

ウザよりもデバッガ上でプログラミングする Smalltalker もいるくらいです。デバッガ上でプログラミングをする利点は、作成中のメソッドが意図通りに動くかを、実行コンテキストの中で、実際の引数値を使って確認できることにあります。

今回の場合、`toggleState`メッセージが、`LGame`インスタンスに送られたと、上部パネルの最初の行に表示されています。これは本来は `LOCell` のインスタンスへ送るべきメッセージです。問題は `cells` の初期化にあるように思われます。`LGame>>initialize` のコードをブラウザで見ると `cells` には `newCellAt:at:` の戻り値が設定されていることがわかります。しかしメソッドの中身を確認すると、戻り値として何も指定していません！メソッドはデフォルトでは `self` を戻り値として返します。従って `newCellAt:at:` の場合は、`LGame` のインスタンスが返ることになります。

🔍 デバッガウィンドウを閉じてください。その後で、“↑c”と `LGame>>newCellAt:at:` メソッドの最後に追加して、`c` を返すようにします。(Method 2.10 参照。)

## Method 2.10: バグの修正。

```

LOGame>newCellAt: i at: j
"位置(i,j)にセルを作成し、適切な画面の位置に追加。新しいセルを返す"
| c origin |
c := LOCell new.
origin := self innerBounds origin.
self addMorph: c.
c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
c mouseAction: [self toggleNeighboursOfCellAt: i at: j].
↑ c

```

第1章で、Smalltalk では値を返す時には↑を使うと習ったのを思い出してください。^をタイプします。

デバッガ上で直接コードを修正した後、Proceed をクリックして、アプリケーションを継続実行することもよくあります。今回の場合は、バグは、失敗したメソッド中というより、オブジェクトの初期化処理にあったので、最も簡単なのは、単に初めからやり直すことです。デバッガウィンドウを閉じ、(halo) で実行中のゲームのインスタンスを破棄し、新たにインスタンスを生成します。

 もう一度 LOGame new openInWorld を実行してください。

これでこのゲームはほぼ正常動作をするはずですが、ただしマウスをクリックしてから離すまでの間にマウスを移動させると、マウス上のセルも切り替わってしまいます。これは SimpleSwitchMorph を継承したことによる振る舞いです。mouseMove: をオーバーライドして、何もしないように修正しましょう。

## Method 2.11: マウスの動作をオーバーライドする。

```

LOGame>mouseMove: anEvent

```

ついに完成しました!

## 2.9 Smalltalk コードの保存と共有

現在、手元には動作する Lights Out ゲームがあります。このゲームを友人と共有するため、どこかに保存したいと思つたとします。もちろん、Pharo のイメージファイルを丸ごと保存し、初めてのプログラムとして動かしてみせることもできます。しかしもし友人が既に自分でコーディングしているイメージファイルを持っていたとしたら、イメージファイル全体は欲しがらないかもしれません。そうしたときに必要なのは、Pharo のイメージファイルからソースコードを取り出し、他のプログラマが自身のイメージファイルに取り込めるようにすることです。

最も簡単なやり方はコードを File out することです。パッケージペインのメニューをアクションクリックすると、PBE-LightsOut 全体をファイルアウトす

るメニュー項目 `various` ▶ `File out` が出てきます。書き出されたファイルは人間が読むことも可能ですが、本来はコンピュータが読むためのものです。このファイルを友人にメールで送れば、ファイルブラウザを使って、ソースコードを自分の Pharo イメージに取り込むことができます。

🕒 PBE-LightsOut パッケージをアクションクリックし、`various` ▶ `File out` でファイルアウトしてください。

“PBE-LightsOut.st” という名のファイルがイメージファイルと同じフォルダ内にできます。テキストエディタでこのファイルを見てみてください。

🕒 まっさらな Pharo のイメージファイルを開き、先ほど出力した `PBE-LightsOut.st` を、ファイルブラウザを使って (`Tools ...` ▶ `File Browser`)、ファイルイン `File in` します。そしてそのイメージ上でゲームが動作することを確認してください。

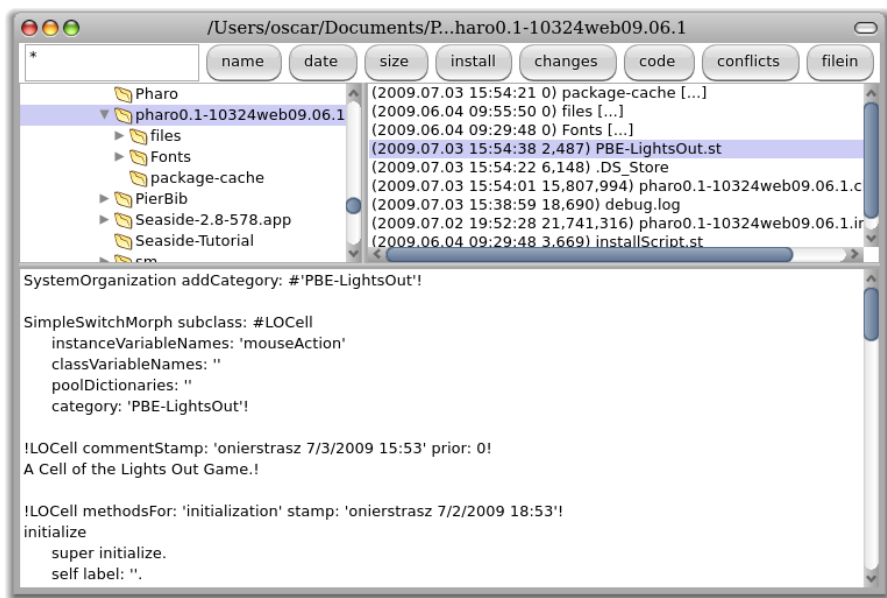


Figure 2.14: Pharo にソースコードをファイルインする。

## Monticello パッケージ

ファイルアウトはコードを保存するには便利ですが、これは明らかに古いやり方です。ほとんどのオープンソースのプロジェクトでは、コードをリポジトリ

で保守することができる CVS<sup>5</sup>や Subversion<sup>6</sup>といったツールを使っています。Pharo プログラマも同様に、Monticello パッケージによるコードの管理の方が、より便利だと思うことでしょう。Monticello パッケージではファイル名の末尾に .mcz が付きます。このファイルはパッケージ内のコードを単に ZIP で圧縮したものです。

Monticello ブラウザを使うことで、FTP や HTTP などの様々なサーバーのリポジトリに、パッケージを保存できます。もちろん自分で管理しているローカルディレクトリのリポジトリに、パッケージを保存することもできます。パッケージのコピーは常にローカルのハードディスク上の *package-cache* フォルダにキャッシュされます。Monticello を使うと、複数のバージョンのプログラムを保存したり、マージしたり、古いバージョンに戻したり、バージョン間の違いを比較したりすることができます。実際のところ、Monticello は分散バージョン管理システムです。つまり開発者の成果物は、CVS や Subversion のように一箇所のリポジトリに置かれるのではなく、複数の異なった場所に保管されます。

もちろん .mcz ファイルを E メールで送ることもできます。受け取った人はそれを *package-cache* フォルダに置くことで、Monticello で閲覧したりロードしたりすることができます。

 **World**メニューから *Monticello* ブラウザを開いてください。

右側のペイン (図 2.15 参照) は Monticello リポジトリのリストです。イメージ内にロードしたコードについてのリポジトリが、すべて並びます。

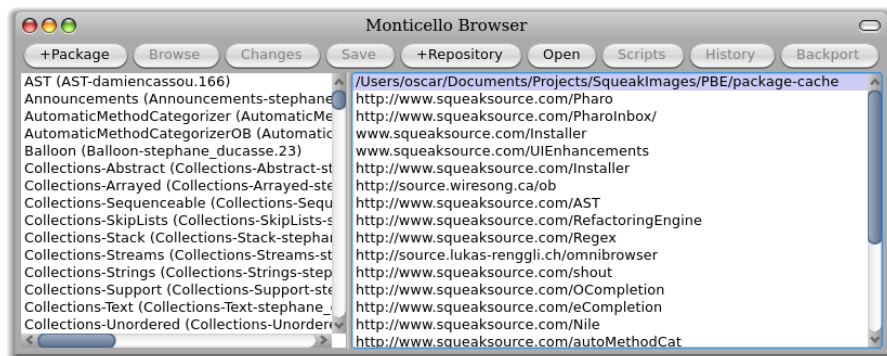


Figure 2.15: Monticello Browser.

Monticello ブラウザのリポジトリリストの一番上には、ローカルディレクトリの *package cache* というリポジトリが表示されます。ここにはネットワークを使ってロード・公開したパッケージのコピーがキャッシュされます。ローカルキャッシュにはローカルでの自分の履歴が保存できるので非常に便利です。つまりインターネットに接続できない場所や、回線が遅くて頻繁にリモートに


<sup>5</sup><http://www.nongnu.org/cvs>

<sup>6</sup><http://subversion.tigris.org>

保存したくない状況でも作業できるということです。


## Monticello を使ったコードの保存と読み込み

Monticello ブラウザの左側には、イメージ内にロードしたパッケージ一覧が、バージョンと共に表示されます。ロード後に修正されたパッケージには、アスタリスクで印が付きます。(これらは時にダーティパッケージ *dirty* と呼ばれます。) パッケージを選択すると、リポジトリリストは、選択したパッケージのコピーを含むリポジトリのみを表示します。

Monticello ブラウザの  **+Package** ボタンを押して、PBE-LightsOut と打ち、PBE-LightsOut パッケージを追加します。


## SqueakSource : Pharo のための SourceForge

コードを保存、共有するのに一番良い方法は、プロジェクト用に SqueakSource サーバーのアカウントを取ることです。SqueakSource は SourceForge<sup>7</sup> のようなものです。SqueakSource は Monticello の HTTP リポジトリの web フロントエンドであり、各種のプロジェクトを管理できます。http://www.squeaksource.com に、公開 SqueakSource サーバーがあります。本書に関係したコードのコピーは http://www.squeaksource.com/PharoByExample.html に保管されています。ウェブブラウザからプロジェクトを見ることもできますが、Pharo から Monticello ブラウザを使ってパッケージを管理する方が効率よく作業できるでしょう。

 ウェブブラウザで <http://www.squeaksource.com> を開いてください。アカウントを作り、Lights Out ゲーム用のプロジェクトを作ってください(つまり登録してください)。

SqueakSource は、Monticello ブラウザ上にリポジトリを追加するための情報を表示します。

SqueakSource にプロジェクトを作成したら、Pharo にその設定情報を伝える必要があるのです。

 PBE-LightsOut パッケージを選択し、Monticello ブラウザの **+Repository** ボタンをクリックしてください。

利用可能な様々なリポジトリタイプが表示されます。SqueakSource のリポジトリを追加するには HTTP を選択してください。サーバー情報を入力するためのダイアログが表示されます。SqueakSource プロジェクトを識別するため、サーバーから提示されたテンプレートをコピーし、Monticello のダイアログに貼り付け、自分のイニシャルとパスワードを記入する必要があります。

---

<sup>7</sup><http://sourceforge.net>

## MCHttpRepository

```
location: 'http://www.squeaksource.com/YourProject'
```

```
user: 'yourInitials'
```

```
password: 'yourPassword'
```

イニシャルとパスワードを空にした場合、プロジェクトを読むことはできても更新はできません。

## MCHttpRepository

```
location: 'http://www.squeaksource.com/YourProject'
```

```
user: ''
```

```
password: ''
```

このテンプレートを入力すると新たなリポジトリが Monticello ブラウザの右側のリストに表示されます。



Figure 2.16: Monticello リポジトリの閲覧

📁 **Save** ボタンを押して、*Light Out* ゲームの最初のバージョンを *SqueakSource* に保存してください。

パッケージを自分のイメージファイルにロードするには、まずバージョンを指定しなければいけません。リポジトリブラウザを使うことで、特定のバージョンを選ぶことができます。**Open** ボタンかメニューのアクションクリックでリポジトリブラウザを開いてください。バージョン選択後、イメージファイルにパッケージを読み込むことができます。

 先ほど保存した PBE-LightsOut リポジトリを開いてください。

Monticello では非常に多くのことができますが、詳細は第 6 章で述べます。<http://www.wiresong.ca/Monticello/> でオンラインドキュメントを読むこともできます。

## 2.10 章のまとめ

この章ではカテゴリ、クラス、そしてメソッドの作成方法について学びました。また、ブラウザ、インスペクタ、デバッガ、そして Monticello ブラウザの使い方についても学びました。

- カテゴリは関連するクラスをグループ化したものです。
- 既存のスーパークラスへメッセージを送ることで、新たなクラスを定義できます。
- プロトコルは関連するメソッドをグループ化したものです。
- ブラウザでメソッド定義を編集して *accept* することで、新たなメソッドを作ったり修正できます。
- インスペクタは、シンプルで汎用的な UI を提供し、ユーザは任意のオブジェクトの値を見てやりとりすることができます。
- ブラウザは定義されていないメソッドや変数をチェックし、修正候補を出してきます。
- Pharo では *initialize* メソッドはオブジェクトが作られた直後に自動的に実行されます。初期化のコードはそこに書きます。
- デバッガは動作中のプログラムの状態を見たり修正したりするための高度な GUI を提供します。
- カテゴリを *File out* することでソースコードを共有することができます。
- Monticello を使って、SqueakSource のような外部のリポジトリ上でコードを共有するのが、より望ましい方法です。



# Chapter 3

## 文法早わかり

Pharo の文法は、他のモダンな Smalltalk の方言と同様、Smalltalk-80 にとっても近いものです。プログラムは一種のピジン英語のようで、声に出して読めるように設計されています。

```
(Smalltalk includes: Class) ifTrue: [ Transcript show: Class superclass ]
```

Pharo の文法規則は必要最小限に抑えられています。本質的には、メッセージを送信する (すなわち式) ための文法しかありません。式は非常に少数の基本要素から組み立てられます。キーワードは 6 個しかありません。さらに、制御構造のための文法も、新しいクラスを宣言するための文法もありません。代わりにオブジェクトにメッセージを送信することによりほとんどのことが達成されます。例えば if-then-else 制御構造の代わりに、Smalltalk では Boolean オブジェクトに ifTrue: といったメッセージを送信します。新しい (サブ) クラスを作成するには、作成したいクラスのスーパークラスにメッセージを送信します。

### 3.1 文法要素

式は以下の構成要素でできています。(i) 6 個の予約語、すなわち self、super、nil、true、false、thisContext の擬似変数 (ii) 数値、文字、文字列、シンボル、配列のためのリテラルオブジェクト 定数式 (iii) 変数宣言 (iv) 代入 (v) ブロック クロージャ (vi) メッセージ送信

表 3.1 は様々な文法要素の例です。

ローカル変数 startPoint は変数名、つまり識別子です。規約により、識別子は「キャメルケース」(すなわち最初を除き単語の先頭を大文字にして連結) で表します。インスタンス変数、メソッドとブロックの引数、一時変数の先頭の文字は小文字でなければなりません。これにより変数の有効範囲がプライベートということがコードの読み手に伝わります。

文法	意味
startPoint	変数名
Transcript	グローバル変数名
self	擬似変数
1	10 進整数
2r101	2 進整数
1.5	浮動小数点数
2.4e7	指数表記
\$a	文字 a
'Hello'	文字列 Hello
#Hello	シンボル #Hello
#{1 2 3}	リテラル配列
{1. 2. 1+2}	動的配列
"a comment"	コメント
x y	変数 x と y の宣言
x := 1	x に 1 を代入
[ x + y ]	x+y を評価するブロック
<primitive: 1>	バーチャルマシンプリミティブもしくはアノテーション
3 factorial	単項メッセージ
3+4	二項メッセージ
2 raisedTo: 6 modulo: 10	キーワードメッセージ
↑ true	値 true を返す
Transcript show: 'hello'. Transcript cr	式セパレータ (.)
Transcript show: 'hello'; cr	メッセージのカスケード (;)

Table 3.1: Pharo の文法の要点

**共有変数** 大文字で始まる識別子はグローバル変数、クラス変数、プール辞書、クラス名です。Transcript はグローバル変数であり、TranscriptStream クラスのインスタンスです。

**レシーバ self** は、現在のメソッドを実行しているオブジェクト自身を指す予約語です。self は「レシーバ」とも呼ばれます。なぜならこのオブジェクトは、メソッドを実行するきっかけになったメッセージを受け取っているからです。self は「擬似変数」と呼ばれます。self には代入できないからです。

**整数** 42 のような通常の 10 進整数に加えて、Pharo では基数表記も使えます。2r101 は基数 2 (すなわちバイナリ) の 101 であり、10 進数の 5 と同じです。

**浮動小数点数** は 10 を基数とした指数で指定することが可能です。2.4e7 は  $2.4 \times 10^7$  です。

**文字** 文字 リテラルにはドル記号を付けます: `$a` は `a` のリテラルです。非印字文字のインスタンスは、`Character space` や `Character tab` のように `Character` クラスに適切なメッセージを送信することにより得ることができます。

**文字列** シングルクォートは文字列 を定義するために使います。クォートを含んだ文字列を表現したい場合は、`'G'day'` のようにクォートを重ねます。

**シンボル** は文字の並びでできている点で文字列に似ています。ただし文字列と異なり、シンボル はグローバルに一意であることが保証されます。`#Hello` というシンボルオブジェクトは一つだけしか存在しません。一方で `'Hello'` という値の文字列オブジェクトは複数あるかもしれません。

**コンパイル時配列** はスペースで区切ったリテラルを `#()` で囲んで定義します。括弧内のすべての要素はコンパイル時に定数でなければなりません。例えば、`#(27 (true false) abc)` は要素が 3 個のリテラル 配列 です。整数 27、真偽値が 2 個入ったコンパイル時配列、そしてシンボル `#abc` が要素になります (これが `#(27 #(true false) #abc)` と同じということに注意してください)。

**動的配列** 波括弧 `{}` は実行時に (動的) 配列を定義します。動的配列の要素は、ピリオドで区切った式です。従って、`{1.2.1+2}` は 1、2、そして `1+2` を評価した結果を要素とする配列を定義します。(波括弧による表記は、`Smalltalk` の方言でも `Pharo` と `Squeak` に特有のもので! 他の `Smalltalk` では、動的配列は明示的に組み立てなければなりません)。

**コメント** はダブルクォートで囲みます。`"hello"` は文字列ではなくコメントなので、`Pharo` コンパイラによって無視されます。コメントは複数行にまたがってもかまいません。

**ローカル変数宣言** ローカル変数を宣言するには、メソッド (およびブロック) 内で、ローカル変数群を縦棒 `||` で囲みます。

**代入** `:=` は変数にオブジェクトを代入します。

**ブロック** 角括弧 `[]` は、ブロッククロージャあるいはレキシカルクロージャとも呼ばれるブロック を定義します。ブロックは関数を表す第一級のオブジェクトです。後で見るように、ブロックは引数を取ることもローカル変数を持つこともできます。

**プリミティブ** `<primitive: ...>` はバーチャルマシン プリミティブ の呼び出しを表します。( `<primitive: 1>` は `SmallInteger` のバーチャルマシンプリミティブです)。プリミティブの後ろにあるコードは、プリミティブが失敗した場合にだけ実行されます。同じ文法はメソッドアノテーションにも使われます。

**単項メッセージ** は単一の単語 (例えば `factorial`) からなり、レシーバ (例えば `3`) に送信されます。

二項メッセージは演算子(例えば +) のようなもので、単一の引数とともにレシーバに送信されます。3+4の場合、レシーバは3で引数は4です。

キーワードメッセージは `raisedTo:modulo:` のように、複数のキーワードからなるメッセージです。キーワードはコロンで終わり、それぞれ引数を取ります。2 `raisedTo: 6 modulo: 10` の式では、`raisedTo:modulo:` がメッセージセレクトアで、コロンの後の6と10が引数です。メッセージは2に送られています。

メソッドリターン ↑はメソッドから値をリターンするために使われます(↑を入力するには↑とタイプしなければなりません)。

文の並び 文の区切りはピリオド(.)です。二つの式の間ピリオドを置くと、それらは独立した文として扱われます。

カスケード セミコロンは、一つのレシーバにメッセージをカスケード送信するために使うことができます。Transcript show: 'hello'; cr では、まず Transcript にキーワードメッセージ show: 'hello' を送り、次に同じレシーバ (Transcript) に単項メッセージ cr を送っています。

Number、Character、String および Boolean クラスについては第8章でより詳細に述べます。

## 3.2 擬似変数

Smalltalk には6個の予約語として、擬似変数があります。nil、true、false、self、super、thisContext です。これらは事前に定義され、代入することができないため、擬似変数と呼ばれます。true、false、nil は定数です。一方 self、super、および thisContext の値は、コードの実行中に動的に変化します。

true と false は、Boolean クラスである True および False の唯一のインスタンスです。詳細については第8章を参照してください。

self は、現在実行しているメソッドのレシーバを指します。

super も現在実行しているメソッドのレシーバを指します。ただし super にメッセージを送信する場合、メソッド探索はそのメソッドを定義しているクラスのスーパークラスから始まります。詳細については第5章を参照してください。

nil は未定義を表すオブジェクトです。クラス UndefinedObject の唯一のインスタンスです。インスタンス変数、クラス変数およびローカル変数は、nil で初期化されます。

thisContext は、実行時スタックのトップフレームを表す擬似変数です。言い換えれば、thisContext は現在実行している MethodContext あるいは BlockClosure を指します。ほとんどのプログラマにとって、thisContext は意識する必要がな

いものですが、デバッガのような開発ツールを実装するには不可欠です。また、例外処理と継続を実装するためにも使われます。

### 3.3 メッセージ送信

Pharo には 3 種類のメッセージ (メッセージ送信) があります。

1. 単項メッセージは引数を取りません。1 factorial は、オブジェクト 1 にメッセージ factorial を送信します。
2. 二項メッセージは引数の一つだけ取ります。1 + 2 は、オブジェクト 1 にメッセージ + を送信します。引数は 2 です。
3. キーワードメッセージは一つ以上の引数を取ります。2 raisedTo: 6 modulo: 10 は、オブジェクト 2 に、メッセージセクタ raisedTo: modulo:、引数 6、引数 10 からなるメッセージを送信します。

単項メッセージのメッセージセクタは、英数字で記述します。先頭は小文字になります。

二項メッセージのメッセージセクタは、以下の文字を 1 文字以上組み合わせることできます。

```
+ - / * ~ < > = @ % | & ? ,
```

キーワードメッセージのセクタは、一連のキーワードを連結したものです。キーワードはそれぞれ小文字で始まりコロンで終わります。

単項メッセージの優先順位が最も高くなります。次に二項メッセージ、最後にキーワードメッセージが評価されます。

つまり、

```
2 raisedTo: 1 + 3 factorial  →  128
```

となります。(最初に 3 に factorial が送られます。次に 1 に + 6 が送られます。最後に 2 に raisedTo: 7 が送られます)。式を評価した結果を示すのに式 → 結果という記法を使っているのを思い出してください。

優先度を別にする、評価順は厳密に左から右となります。つまり、

```
1 + 2 * 3  →  9
```

です。7 ではありません。評価順を変えるには括弧を使わなければなりません。

```
1 + (2 * 3)  →  7
```

複数のメッセージ送信は、ピリオドやセミコロンを使って組み立てることができます。ピリオドで区切られた式の並びは、文として順々に評価されます。

```
Transcript cr.
Transcript show: 'hello world'.
Transcript cr
```

これは、Transcript オブジェクトに `cr` を送信し、次に Transcript に `show: 'hello world'` を送り、最後に Transcript に再び `cr` を送っています。

一連のメッセージを同じレシーバに送信する場合、カスケードとしてより簡潔に表現することができます。レシーバを一度だけ指定し、メッセージの並びをセミコロンで区切ります。

```
Transcript cr;
  show: 'hello world';
  cr
```

これは前の例とまったく同じ意味になります。

### 3.4 メソッドのシンタックス

式は Pharo のあらゆる場所 (例えばデバッガ、ワークスペース、ブラウザなど) で評価することができますが、通常、メソッドはブラウザやデバッガ内で定義します。(メソッドは外部からファイルインすることもできますが、これは Pharo のプログラミングにとって普通のやり方ではありません)。

プログラムは、クラスに属するメソッド単位で開発していきます。(クラスは、既存のクラスに、サブクラスを作成するためのメッセージを送信して定義します。クラスを定義するための特別な文法はありません)。

以下は String クラスの `lineCount` メソッドです。(慣習により、メソッドは `ClassName»methodName` と表記します。そのため `String»lineCount` と書き表せませす)。

#### Method 3.1: *Line count*

```
String»lineCount
  "レシーバの行数を答える。
  crに会うたびに1行増やす。"
  | cr count |
  cr := Character cr.
  count := 1 min: self size.
  self do:
    [c | c == cr ifTrue: [count := count + 1]].
  ↑ count
```

文法上、メソッドは以下のものから構成されます。

1. 名前 (すなわち `lineCount`) と引数 (この例では一つもない) からなるメッセージパターン

2. コメント (どこに書いてもかまいませんが、メソッドが何を行うかの説明を一番上を書く慣習があります)
3. ローカル変数 (すなわち `cr` と `count`) の宣言
4. ドットで区切られたメッセージ式。(ここでは式は 4 個)。

↑(とタイプする) が先頭に書いてある式は、評価されるとメソッドを終了してその式の値を返します。明示的に値を返さずに終了するメソッドの場合、暗黙的に `self` をリターンします。

引数とローカル変数は常に小文字で始めなければなりません。大文字で始まる名前はグローバルな変数であるとみなされます。例えばクラス名 (例: `Character`) は、そのクラスを表すオブジェクトを参照するためのグローバル変数です。

### 3.5 ブロックのシンタックス

ブロックは、遅延評価のためのメカニズムを提供します。ブロックは本質的には無名の関数です。ブロックを評価するには、メッセージ `value` を送ります。ブロックは、明示的なりターン (↑) がなければ、最後の式の値を返します。

```
[1 + 2] value → 3
```

ブロックは引数を持つことができます。引数は先頭にコロンを付けて宣言します。縦棒でブロック本体と引数の宣言とを区切ります。引数が 1 個のブロックを評価するには、ブロックに、引数一つのメッセージ `value:` を送信しなければなりません。引数が 2 個のブロックを評価するには、ブロックに `value: value:` を送信しなければなりません。同様に 4 個までの引数を渡すことができます。

```
[ :x | 1 + x ] value: 2 → 3
[ :x :y | x + y ] value: 1 value: 2 → 3
```

引数の数が 4 個より多いブロックには `valueWithArguments:` を使い、引数を配列にして渡さなければなりません。(引数が多いブロックは、しばしば設計上の問題がある兆候です)。

ブロック内でもローカル変数を宣言することができます。メソッドのローカル変数宣言と同様、縦棒で囲みます。ローカル変数は引数リストの後ろに宣言します。

```
[ :x :y || z | z := x + y. z ] value: 1 value: 2 → 3
```

ブロックはそれを取り囲む環境の変数を参照することができるので、レキシカルクロージャです。次のブロックは、取り囲んでいる環境の変数 `x` を参照しています。

```
|x|
x := 1.
[:y | x + y] value: 2  →  3
```

ブロックはクラス `BlockClosure` のインスタンスです。つまりブロックはオブジェクトなので、他の普通のオブジェクトと同様、変数に代入したり引数として渡したりすることができます。

### 3.6 条件式とループの要点

Smalltalk には制御構造のための特別の文法はありません。代わりに、真偽値や数、コレクションに対し、ブロックを引数とするメッセージを送ることで制御構造を表します。

条件式は、真偽値を返す式の値に、メッセージ `ifTrue:`、`ifFalse:`、`ifTrue:ifFalse:` などを送ることにより表現します。真偽値について、詳しくは第8章を参照してください。

```
(17 * 13 > 220)
ifTrue: [ 'bigger' ]
ifFalse: [ 'smaller' ]  →  'bigger'
```

ループは、ブロック、整数、コレクションなどにメッセージを送ることで表します。ループの終了条件は、繰り返し評価されるかもしれないので、真偽値そのものではなく真偽値を返すブロックでなければなりません。以下は非常に手続き的なループの例です。

```
n := 1.
[ n < 1000 ] whileTrue: [ n := n*2 ].
n  →  1024
```

`whileFalse:` では終了条件が逆になります。

```
n := 1.
[ n > 1000 ] whileFalse: [ n := n*2 ].
n  →  1024
```

`timesRepeat:` により単純な固定回数の繰り返しができます。

```
n := 1.
10 timesRepeat: [ n := n*2 ].
n  →  1024
```

数にメッセージ `to:do:` を送信することができます。するとその数(レシーバ)がループカウンターの初期値になります。`to:` の引数はループカウンターの上限です。また `do:` の引数はブロックで、ループカウンターの現在の値を引数として取ります。



```
result := String new.
1 to: 10 do: [:n | result := result, n printString, ''].
result  →  '1 2 3 4 5 6 7 8 9 10'
```

高階イテレータ コレクションは様々なクラスで実装されています。多くは同じプロトコルをサポートします。コレクションの要素を繰り返し処理するための、最も重要メッセージには、`do:`、`collect:`、`select:`、`reject:`、`detect:` および `inject:into:` があります。これらは高レベルのイテレータを定義しているメッセージであり、それによって非常にコンパクトなコードを書くことができます。

`Interval` は、開始、終了を指定した数の並びを繰り返し処理できるコレクションです。`1 to: 10` は 1 から 10 までの `Interval` を表します。`Interval` はコレクションなので、これにメッセージ `do:` を送信することができます。`do:` の引数はブロックであり、コレクションの要素に対して繰り返し評価されます。

```
result := String new.
(1 to: 10) do: [:n | result := result, n printString, ''].
result  →  '1 2 3 4 5 6 7 8 9 10'
```

`collect:` は各要素を変換して同じサイズの新しいコレクションを作ります。

```
(1 to: 10) collect: [:each | each * each]  →  #(1 4 9 16 25 36 49 64 81 100)
```

`select:` と `reject:` は、真偽値ブロックの条件を満たす (あるいは満たさない) 要素からなる新しいコレクション (部分集合) を作ります。`detect:` は条件を満たした最初の要素を返します。文字列が文字のコレクションであることを覚えておいてください。つまり文字列内の文字はイテレートすることができます。

```
'hello there' select: [:char | char isVowel]  →  'eoe'
'hello there' reject: [:char | char isVowel]  →  'hll thr'
'hello there' detect: [:char | char isVowel]  →  $e
```

最後に、コレクションは `inject:into:` メソッドにより、関数型の畳み込み演算もサポートします。これを使うと、シードとなる値から開始し、コレクションの各要素を注入 (`inject`) していった累積的な値を得ることができます。和や積の累積は `inject:into:` の典型的な応用例です。

```
(1 to: 10) inject: 0 into: [:sum :each | sum + each]  →  55
```

これは  $0+1+2+3+4+5+6+7+8+9+10$  と等価です。

コレクションに関しては第 9 章で詳しく述べます。

### 3.7 プリミティブとプラグマ

Smalltalk ではすべてがオブジェクトです。そしてあらゆることがメッセージ送信により起こります。しかしある地点で底に当たります。ある種のオブジェクトはバーチャルマシン プリミティブ を呼び出さないと動作できません。

例えば、以下のメソッドはすべてプリミティブとして実装されています。メモリ割り当て (`new`、`new:`)、ビット操作 (`bitAnd:`、`bitOr:`、`bitShift:`)、ポインタおよび整数演算 (+、-、<、>、\*、/、=、==...)、および配列のアクセス (`at:`、`at:put:`)。

プリミティブは、文法 `<primitive: aNumber>` で呼び出します。プリミティブを呼び出すメソッドで、通常の Smalltalk コードがさらに続くことがあります。そうしたコードは、プリミティブが失敗した場合にのみ評価されます。

`SmallInteger>+` のコードを見てみましょう。プリミティブが失敗した場合、式 `super + aNumber` が評価され値が返されます。

#### Method 3.2: プリミティブメソッド

```
+ aNumber
"プリミティブ。レシーバと引数で加算し、結果が SmallInteger ならそれを返す。
引数が結果が SmallInteger でなければ失敗する。
必須。メソッド探索なし。Object のドキュメント whatsAPrimitive を参照のこと。"

<primitive: 1>
↑ super + aNumber
```

Pharo では、山括弧を使った記法はメソッドのアノテーションにも使われます。これはプラグマと呼ばれます。

### 3.8 まとめ

- Pharo には予約語が 6 個 (だけ) あります。これらは擬似変数とも呼ばれます。true、false、nil、self、super、そして thisContext です。
- 5 種類のリテラルオブジェクトがあります: 数 (5、2.5、1.9e15、2r111)、文字 (\$a)、文字列 ('hello')、シンボル (#hello)、配列 (#('hello' #hi)) です。
- 文字列はシングルクォートで、コメントはダブルクォートで囲みます。文字列内にクォートを書くには、クォートを重ねます。
- 文字列と異なり、シンボルはグローバルに一意であることが保証されます。
- リテラル配列を定義するには #(...) を使います。動的配列を定義するには {...} を使います。#(1 + 2) size → 3 ですが、{1 + 2} size → 1 なので注意してください。

- 3種類のメッセージがあります: 単項 (例えば `1 asString`、`Array new`)、二項 (例えば `3 + 4`、`'hi'`、`'there'`)、キーワード (例えば `'hi' at: 2 put: $o`) です。
- カスケードによるメッセージ送信では、同じターゲットに連続してメッセージを送れます。各メッセージはセミコロンで区切ります。 `OrderedCollection new add: #calvin; add: #hobbes; size` → `2`。
- ローカル変数は縦棒で囲んで宣言します。代入には `:=` を使います。 `|x| x:=1`。
- 式は、メッセージ送信、カスケード、代入からなります。これらは括弧でグループ化されることもあります。文はピリオドによって区切られた式です。
- ブロッククロージャは角括弧で囲まれた式です。ブロックは引数を取ってもよく、一時変数を使うこともできます。ブロック内の式は、ブロックに、`value...` メッセージを、適切な個数の引数で送ったときに初めて評価されます。  
`[x | x + 2] value: 4` → `6`。
- 制御構造用の文法はありません。条件によってブロックを評価させるメッセージを送ることで実現します。  
(`Smalltalk includes: Class`) `ifTrue: [ Transcript show: Class superclass ]`。



# Chapter 4

## メッセージ構文を理解しよう

Smalltalk のメッセージ構文は非常に単純ですが、標準的なものではないので慣れるのに少し時間がかかるでしょう。この章では、この特殊なメッセージ構文に順応するためのガイダンスを提供します。もし既にこの構文に慣れているのなら、この章は飛ばしてもかまいませんし後で読んでもよいでしょう。

### 4.1 メッセージを読み取る

Smalltalk では、第 3 章で紹介した構文要素 (`:= ↑ . ; # () {} [: |]`) を除けばすべてがメッセージ送信です。C++ のように、+ などの演算子を自作のクラス用に定義することもできますが、すべての演算子の優先順位は同じです。さらに、あるメソッドが受け取る引数の個数は変えられません。『-』は常に二項演算子です; 単項の『-』演算子を得る方法はありません。

Smalltalk では、メッセージが送信される順番はメッセージの種類によって決定されます。そしてメッセージには 3 種類しかありません: 単項、二項 およびキーワード メッセージです。単項メッセージが常に最初に送られ、その次が二項メッセージ、そして最後がキーワードメッセージです。他の多くの言語と同様に、括弧を使って評価順を変えることもできます。これらの規則によって、Smalltalk のコードは可能な限り読みやすく、かつ規則のことで頭を悩ませなくてもよくなっています。

Smalltalk のほとんどの計算はメッセージパッシングによって行われるので、メッセージを正しく読み取ることが非常に重要です。次の用語と概念を理解することが役に立つでしょう:

- メッセージは、セクタ およびオプションのメッセージ引数からなります。
- メッセージは、レシーバに送信されます。

- レシーバとメッセージの組み合わせが、メッセージ送信と呼ばれます (図 4.1)。

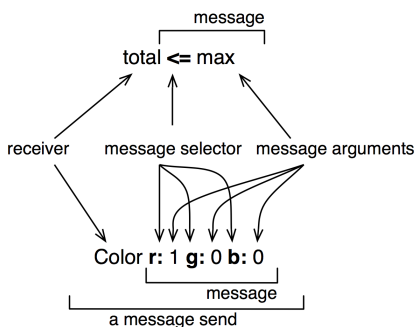


Figure 4.1: レシーバ、セレクト、引数からなるメッセージ送信の例 (二つ)。



Figure 4.2: aMorph color: Color yellow は二つのメッセージ送信の組み合わせ: Color yellow および aMorph color: Color yellow。

メッセージは常にレシーバに送信されます。レシーバは、リテラル・ブロック・変数・他のメッセージを評価した結果、のいずれかです。

図 4.2 では、メッセージのレシーバが同定しやすくなるように、レシーバに下線を引きました。また、それぞれのメッセージ送信を楕円で囲み、送信される順番に従って番号を付けました。

図 4.2 は Color yellow と aMorph color: Color yellow という二つのメッセージ送信を表しています。従って楕円も二つあります。メッセージ送信 Color yellow がまず実行されるので、それを囲む楕円に番号 1 が付いています。全体ではレシーバが二つあります: 一つは aMorph でメッセージ color: ... が送られており、もう一つは Color でメッセージ yellow が送られています。レシーバにはそれぞれ、既に述べたように下線が引いてあります。

レシーバはメッセージ送信の最初の要素になることができます。例えば 100 + 200 の 100 や Color yellow の Color がそうです。しかし、レシーバは他のメッセージ送信の結果であってもかまいません。例えば Pen new go: 100 というメッセージ送信では、メッセージ go: 100 のレシーバは Pen new オブジェクト (Pen new の結果として返されるオブジェクト) です。いずれにせよ、メッセージはレシーバと呼ばれるオブジェクトに送信され、レシーバは他のメッセージ送信の結果であってもよいということです。

表 4.1 にいくつかのメッセージ送信の例を挙げました。必ずしもすべてのメッセージ送信に引数があるわけではない、ということに注意してください。open のような単項メッセージには引数はありません。go: 100 のような単独のキーワードメッセージや + 20 のような二項メッセージは引数を取ります。単純

メッセージ送信	メッセージの種類	結果
Color yellow aPen go: 100	単項 キーワード	色を表すオブジェクト。 レシーバであるペン・オブジェクトが 100 ピクセル移動する。
100 + 20	二項	数値オブジェクト 100 にメッセージ + を数値 20 とともに送信。
Browser open Pen new go: 100	単項 単項およびキーワード	新しいブラウザを開く。 ペン・オブジェクトが生成され 100 ピクセル移動する。
aPen go: 100 + 20	キーワードおよび二項	レシーバであるペン・オブジェクトが 120 ピクセル移動する。

Table 4.1: メッセージ送信と種類の例

なメッセージ・組み合わせられたメッセージ、という分類もできます。Color yellow や 100 + 20 は単純です: 一つのメッセージが一つのオブジェクトに送信されるだけです。一方 aPen go: 100 + 20 は二つのメッセージ送信が組み合わさったものです: + 20 が 100 に送信され、その結果を引数として go: が aPen に送信されます。オブジェクトを返す式 (代入、メッセージ送信およびリテラル) であればレシーバになることができます。Pen new go: 100 では、メッセージ go: 100 が Pen new オブジェクト (Pen new の結果として返されるオブジェクト) に送信されます。

## 4.2 3 種類のメッセージ

Smalltalk ではメッセージの送信順序を決定するいくつかの単純な規則が定義されています。これらの規則は以下の 3 種類のメッセージの別に基づきます:

- 単項メッセージは、あるオブジェクトに他の追加情報なしで送信されるメッセージです。例えば 3 factorial の factorial は単項メッセージです。
- 二項メッセージは、演算子 (しばしば算術的な) からなるメッセージです。これらは、常にレシーバと引数という二つのオブジェクトが関与するため、二項メッセージと呼ばれます。10 + 20 の場合、+ が二項メッセージで引数 20 と共にレシーバ 10 に送信されています。
- キーワードメッセージは、一つあるいはそれ以上のキーワードからなっています。それぞれのキーワードの最後にはコロン (:) が付いており、キーワードごとに一つ引数を取ります。例えば anArray at: 1 put: 10 の場合、キーワード at: は 1 という引数を取り、キーワード put: が引数 10 を取っています。

## 単項メッセージ

単項メッセージは引数を必要としないメッセージです。その構文は receiver selector という形式です。セクタ (selector) は、: を含まない単純な文字列です (例えば factorial、open、class)。

```
89 sin      → 0.860069405812453
3 sqrt     → 1.732050807568877
Float pi   → 3.141592653589793
'blop' size → 4
true not   → false
Object class → Object class "Object のクラスは Object class (!)"
```

単項メッセージは引数を必要としないメッセージです。  
その構文は receiver **selector** という形式です。

## 二項メッセージ

二項メッセージは引数を一つだけ取ります。また二項メッセージのセクタは、次の文字セットからの 1 文字以上の繰り返しでなければなりません: +、-、\*、/、&、=、>、|、<、!、~、および @。構文解析時に問題が起こるので、-- はセクタとしては認められないことに注意してください。

```
100@100    → 100@100 "Point オブジェクトを生成"
3 + 4      → 7
10 - 1     → 9
4 <= 3     → false
(4/3) * 3 = 4 → true "同値性のテストも、Smalltalk では単なる二項メッセージ。分数オブジェクトは値を正確に表現できる"
(3/4) == (3/4) → false "別個に作られた二つの分数は、同じオブジェクトではない"
```

二項メッセージは引数を一つだけ取ります。また二項メッセージのセクタは、次の文字セットからの 1 文字以上の繰り返しでなければなりません: +、-、\*、/、&、=、>、|、<、~、および @。-- は認められません。  
二項メッセージの構文は receiver **selector** argument という形式です。



## キーワードメッセージ

キーワードメッセージは一つ以上の引数を取るメッセージです。そのセレクトタは一つ以上のキーワードを連結したもので、各キーワードは:で終わります。キーワードメッセージの構文は次の形式です: receiver **selectorWordOne:** argumentOne **wordTwo:** argumentTwo

それぞれのキーワードが引数を取らず取ります。つまり r:g:b:というセレクトタは3引数であり、playFileNamed: や at: は1引数、at:put: は2引数です。Colorクラスのインスタンスを作るにはクラスに r:g:b: メッセージを送ります (例えば Color r: 1 g: 0 b: 0 によって赤色を表すオブジェクトを作ることができます)。ここではコロンもセレクトタの一部であるということに注意してください。

Java や C++ であれば、Smalltalk における Color r: 1 g: 0 b: 0 は Color.rgb(1,0,0) というメソッド呼び出しに相当するでしょう。

1 to: 10                   → (1 to: 10) "区間オブジェクトを生成"  
 Color r: 1 g: 0 b: 0     → Color red "色オブジェクトを生成"  
 12 between: 8 and: 15   → true

nums := Array newFrom: (1 to: 5).  
 nums at: 1 put: 6.  
 nums   → #(6 2 3 4 5)

キーワードメッセージは一つ以上の引数を取るメッセージです。そのセレクトタは一つ以上のキーワードを連結したもので、各キーワードはコロン(:)で終わります。キーワードメッセージの構文は次の形式です:  
 receiver **selectorWordOne:** argumentOne **wordTwo:** argumentTwo

## 4.3 メッセージ式の組み合わせ

ここまでに述べた3種類のメッセージにはそれぞれ異なる優先順位があるので、これらをエレガントに組み合わせることができます。

1. 単項メッセージが常に最初に送信され、次に二項、そして最後にキーワードメッセージが送信されます。
2. 括弧で囲まれたメッセージは他のメッセージよりも優先して送信されます。

### 3. 同じ種類のメッセージは左から右に評価されます。

この規則により、Smalltalk のプログラムは極めて自然に読み進められるようになっています。メッセージが意図した通りの順番で送信されることを確実にしたいのであれば、図 4.3 にあるように括弧を足していくこともできます。この図では、メッセージ `yellow` が単項メッセージであり `color:` がキーワードメッセージであるため、`Color yellow` がまず先に送信されます。しかし括弧で囲まれたメッセージがまず送信されるため、(冗長な)括弧を `Color yellow` の周りに書いてこれがまず先に送信されるということを強調することができます。本節の残りではこれらの点について説明します。

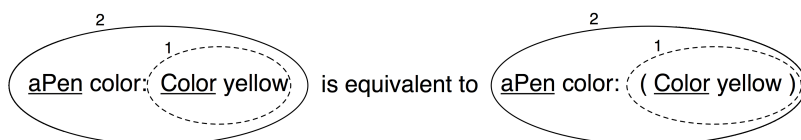


Figure 4.3: 単項メッセージが最初に送信されるので、`Color yellow` がまず送信され、返された色オブジェクトがメッセージ `aPen color:` の引数として渡される。

## 単項 > 二項 > キーワード

単項メッセージがまず送信され、次に二項、そして最後にキーワードメッセージが送信されます。単項メッセージの優先順位が他の種類のメッセージよりも高いとも言えます。

**規則 1:** 単項メッセージがまず送信され、次に二項、そして最後にキーワードメッセージが送信されます。  
単項 > 二項 > キーワード

以下の例が示すように、Smalltalk の構文規則によれば、大概のメッセージ送信は自然に読むことができます:

```
1000 factorial / 999 factorial  →  1000
2 raisedTo: 1 + 3 factorial    →  128
```

ただし残念ながら、算術演算の場合にはこの規則はやや単純すぎます。そのため二項演算子を複数組み合わせるときは、場合によって括弧を使って評価順を強制する必要があります:

```
1 + 2 * 3  →  9
1 + (2 * 3) →  7
```

以下のさらに複雑な (!) 例は、込み入った Smalltalk の式も自然に読めるということの良い例になっています。

```
[aClass | aClass methodDict keys select: [:aMethod | (aClass >> aMethod) isAbstract ]]
value: Boolean    →  an IdentitySet(#or: #| #and: #& #ifTrue: #ifTrue:ifFalse:
#iffalse: #not #iffalse:ifTrue:)
```

ここでは Boolean クラスのどのメソッドが抽象メソッドであるかを調べようとしています<sup>1</sup>。このブロックは、引数として渡されたクラス aClass にそのメソッド辞書のキー集合を問い合わせ、その中から抽象メソッドを選択 (select:) しています。ここでは aClass を Boolean にバインドしています。この式の中で括弧は、二項メッセージ >> を単項メッセージ isAbstract より先に送信するためだけに必要でした (>> はメソッドをクラスから取り出す二項演算子です)。全体を評価するとメソッド名の集合が返ります。これらのメソッドは Boolean の具象サブクラス (True と False) で実装しなくてはなりません。

例。 メッセージ aPen color: Color yellow の中には、Color に送信される単項メッセージ yellow と aPen に送信される キーワードメッセージ color: があります。単項メッセージがまず送信されるので、Color yellow がまず送信 (評価) され (1)、その結果として返された色オブジェクトがメッセージ aPen color: aColor の引数 (aColor) となります (2)。Example 4.1 を見てください。図 4.3 は、これらのメッセージがどのように送信されるかをグラフィカルに表しています。

#### Example 4.1: aPen color: Color yellow の評価順を分解する

```
aPen color: Color yellow
(1)      Color yellow    "単項メッセージがまず送信される"
         → aColor
(2) aPen color: aColor    "キーワードメッセージが次に送信される"
```

例。 メッセージ aPen go: 100 + 20 の中には、二項メッセージ + 20 とキーワードメッセージ go: があります。二項メッセージはキーワードメッセージより先に送信されるので、100 + 20 がまず評価されます (1): + 20 がオブジェクト 100 に送信され、数値 120 が返ります。次に aPen にメッセージ go: 120 が送信されます (2)。Example 4.2 は、このメッセージ送信がどのように実行されるかを示しています。

#### Example 4.2: aPen go: 100 + 20 を分解する

```
aPen go: 100 + 20
(1)      100 + 20        "二項メッセージがまず送信される"
         → 120
(2) aPen go: 120        "キーワードメッセージが次に送信される"
```

<sup>1</sup> 実を言うと、同等の式はさらに簡単に Boolean methodDict select: #isAbstract thenCollect: #selector と書くこともできます。

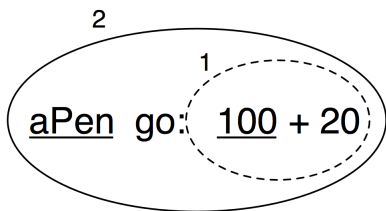


Figure 4.4: 二項メッセージはキーワードメッセージより先に送信される。

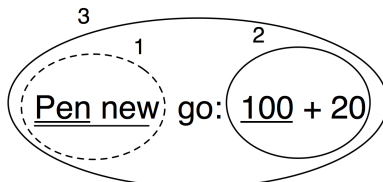


Figure 4.5: Pen new go: 100 + 20 を分解する

例。 練習としてメッセージ Pen new go: 100 + 20 の評価順を分解してみてください。このメッセージには単項メッセージが一つ、キーワードメッセージが一つ、二項メッセージが一つあります (図 4.5)。

## 括弧優先

規則 2。括弧で囲まれたメッセージは他のメッセージよりも優先して送信されます。

(任意のメッセージ) > 単項 > 二項 > キーワード

1.5 tan rounded asString = (((1.5 tan) rounded) asString) → true "括弧は不要"  
 3 + 4 factorial → 27 "(5040 ではない)"  
 (3 + 4) factorial → 5040

以下の例では、lowMajorScaleOn: を play よりも先に送信するために括弧が必要です。

(FMSound lowMajorScaleOn: FMSound clarinet) play

"(1) メッセージ clarinet が FMSound クラスに送信され、クラリネットの音色オブジェクトが作られる。

(2) この音色オブジェクトが lowMajorScaleOn: キーワードメッセージの引数として FMSound に送信される。

(3) 結果の音オブジェクトに play が送信され、演奏される。"

例。 メッセージ (65@325 extent: 134@100) center は、左上が座標 (65, 325) で大きさが 134×100 であるような長方形の、中央の座標を返します。Example 4.3 は、このメッセージがどのように分解されて送信されるかを示しています。まず括弧で囲まれたメッセージが送信されます: 括弧の内側には二つの二項メッセージ 65@325 と 134@100 があるので、これらがまず送信 (評価) されて二つの点オブジェクトが返ります。次に最初の点オブジェクトにキーワードメッセー

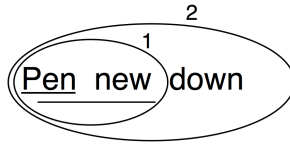


Figure 4.6: Pen new down を分解する

ジ extent: が送信され、長方形オブジェクトが返ります。最後に単項メッセージ center がその長方形オブジェクトに送信され、点オブジェクトが返ります。もし括弧がなかったとすると、数値 100 はメッセージ center を理解できないため、評価時にエラーとなります。

Example 4.3: 括弧の例。

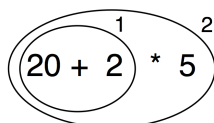
(65@325 extent: 134@100) center	
(1) 65@325	"二項"
→ aPoint	
(2) 134@100	"二項"
→ anotherPoint	
(3) aPoint extent: anotherPoint	"キーワード"
→ aRectangle	
(4) aRectangle center	"単項"
→ 132@375	

左から右への評価

ここまでに、異なる種類のメッセージがどのような優先順位で処理されるかを学びました。残る疑問は同じ優先順位のメッセージがどのような順番で送信されるかということです。これらは左から右に送信されます。この振る舞いは実は Example 4.3 の中で確認しています。Example 4.3 では二つの点オブジェクトの生成メッセージ (@) が先に送信されていました。

**規則 3。** 同じ種類のメッセージがある場合は、評価は左から右の順で行われます。

例。 メッセージ送信 Pen new down では単項メッセージのみが使われているので、左端の Pen new がまず送信されます。その結果は新しいペン・オブジェクトで、このペン・オブジェクトに 2 番目のメッセージ down が送信されます (図 4.6)。



## 伝統的な算術記法との食い違いに関する問題

メッセージ式の組み合わせに関する規則はごく単純なものです。算術演算が二項メッセージとして記述されている場合には、伝統的な記法との食い違いが問題になります。以下は、余計な括弧が必要となる例です。

$3 + 4 * 5$	→	35	"(23 ではない) 二項メッセージが左から右に実行されるため"
$3 + (4 * 5)$	→	23	
$1 + 1/3$	→	(2/3)	"4/3 ではない"
$1 + (1/3)$	→	(4/3)	
$1/3 + 2/3$	→	(7/9)	"1 ではない"
$(1/3) + (2/3)$	→	1	

例。メッセージ送信  $20 + 2 * 5$  では、+ と \* という二項メッセージだけが使われています。しかし Smalltalk では + と \* の間に特に優先順位の違いはありません。どちらも単なる二項メッセージであって、\* は + よりも優先順位が高いわけではありません。Example 4.4 のように、ここでは左端の + がまず送信され (1)、その結果に \* が送信されます (2)。

### Example 4.4: $20 + 2 * 5$ を分解する

"二項メッセージ間に優先順位の違いがないので、算術の約束としては \* がまず先に送信されるべきだが、ここでは左端の + がまず評価される。"

$20 + 2 * 5$	
(1) $20 + 2$	→ 22
(2) 22 * 5	→ 110

Example 4.4 の結果は 30 ではなく 110 となります。これはもしかすると意図したものとうちがたかもしれませんが、メッセージ送信の規則をそのまま適用した結果であり、Smalltalk の簡素なモデルの代償と言えるかもしれません。正しい (意図した) 結果を得るためには括弧を使う必要があります。括弧で囲まれたメッセージが先に評価されるため、メッセージ送信  $20 + (2 * 5)$  は Example 4.5 に示されたような結果を返します。

### Example 4.5: $20 + (2 * 5)$ を分解する

"括弧で囲まれたメッセージが先に評価されるので \* が + の前に送信される。これは正しい振る舞いである。"

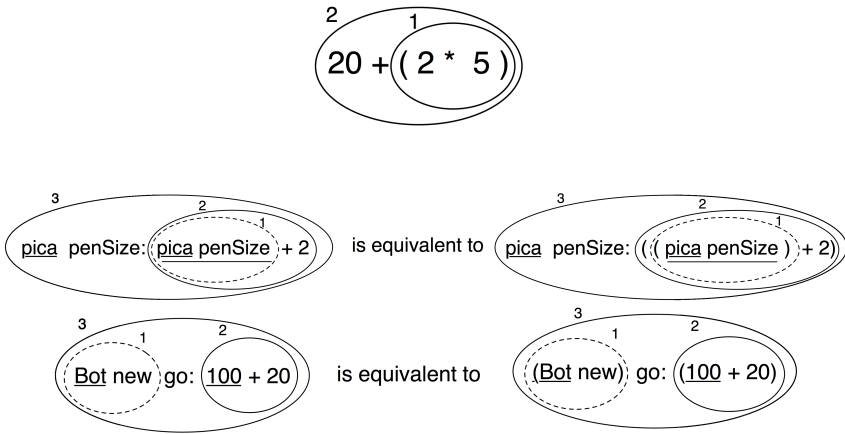


Figure 4.7: 括弧の有無に関わらない等価なメッセージ。

規則に基づく評価順	明示的に括弧を付けて書いた等価なメッセージ送信
aPen color: Color yellow	aPen color: (Color yellow)
aPen go: 100 + 20	aPen go: (100 + 20)
aPen penSize: aPen penSize + 2	aPen penSize: ((aPen penSize) + 2)
2 factorial + 4	(2 factorial) + 4

Figure 4.8: メッセージ送信と、括弧を完全に付けた等価な式の対応例

```

20 + (2 * 5)
(1)  (2 * 5)  →  10
(2)  20 + 10  →  30
    
```

Smalltalk では、+ や \* のような算術演算子に特別な優先順位はありません。+ や \* も単なる二項メッセージで、\* の優先順位の方が + よりも高いということはありません。意図した結果を得るには括弧を使ってください。

最初の規則は、単項メッセージは二項メッセージやキーワードメッセージに優先する、というものでした。だから単項メッセージの周りに括弧を書く必要はありません。表 4.8 に、普通に書いたメッセージ送信と優先順位規則を無視して明示的に括弧を付けて書いた等価なメッセージ送信を並べて書きました。この表では、左右どちらのメッセージ送信も同じ実行順序となり同じ結果を返します。

## 4.4 キーワードメッセージの切れ目を見つけるためのヒント

初心者にとってしばしば問題なのは、いつ括弧が必要なかがわかりにくいことです。そこで、コンパイラがどのようにキーワードメッセージを認識するかを見てみましょう。

### 括弧か括弧なしか？

文字 `[, ], (および)` がコードの領域を区切るために使われます。このように囲まれた領域の中では、`:` で終端された語 (キーワード) と引数の並びで、`.` や `;` で分断されていない最長のものがキーワードメッセージになります。つまりこのような並びが文字 `[, ], (および)` で囲まれた領域にある場合、その並びがその領域内に局所的なキーワードメッセージになります。

以下の例では、`rotatedBy:magnify:smoothing:` と `at:put:` という二つの独立したキーワードメッセージが使われています。

```
aDict
  at: (rotatingForm
    rotateBy: angle
    magnify: 2
    smoothing: 1)
  put: 3
```

文字 `[, ], (および)` がコードの領域を区切るために使われます。このように囲まれた領域の中では、`:` で終端された語 (キーワード) と引数の並びで、`.` や `;` で分断されていない最長のものがキーワードメッセージになります。つまりこのような並びが文字 `[, ], (および)` で囲まれた領域にある場合、その並びがその領域内に局所的なキーワードメッセージになります。

ヒント。 もしまだ優先順位規則がよくわからないのであれば、まずは括弧をなるべくたくさん付けるようにして同じ優先順位を持つメッセージ同士であっても確実に区別できるようにしてかまいません。

以下の例では、`x isNil` は単項メッセージでありキーワードメッセージ `ifTrue:` よりも先に送信されるため、括弧は本来必要ありません。

```
(x isNil)
  ifTrue:[...]
```



以下の例では、`includes:` と `ifTrue:` はどちらもキーワードメッセージなので、括弧を使う必要があります。

```
ord := OrderedCollection new.
(ord includes: $a)
  ifTrue:[...]
```

括弧を付けなかったとしたら、`includes:ifTrue:` という意味不明なメッセージがコレクション (`ord`) に送信されてしまいます!

## [] と () のどちらを使うか

もしかすると、いつ鍵括弧を括弧の代わりに使うべきがよくわからないかもしれませんね。基本原則としては、もしある式が何回 (もしかすると 0 回かも) 評価されるかあらかじめわからないときにはその式を [] で囲みます。[*expression*] は *expression* からブロック クロージャ (すなわちオブジェクト) を生成します。クロージャは、必要に応じて何回でも (0 回の可能性も含めて) 評価することでできるオブジェクトです。[] の中にはメッセージ送信、変数、リテラル、代入あるいは別のブロックを書くことができます。

`ifTrue:` や `ifTrue:ifFalse:` のような条件分岐がブロックを必要とするのはこのためです。同じ原則に従い、`whileTrue:` メッセージのレシーバと引数も、何回評価されるかはあらかじめ決定できないので、これらを鍵括弧で囲む必要があります。

一方通常の括弧は、メッセージ送信の順序のみに影響を与えます。すなわち (*expression*) では *expression* は常に一回だけ評価されます。

[ x isReady ] whileTrue: [ y doSomething ]	"レシーバと引数の両方ともブロックでなくではならない"
4 timesRepeat: [ Beeper beep ]	"引数は 2 回以上実行されるのでブロックでなくではならない"
(x isReady) ifTrue: [ y doSomething ]	"レシーバは 1 回だけ評価されるのでブロックではない"

## 4.5 式の並び

複数の式 (すなわちメッセージ送信や代入...) をピリオドで区切って並べると、それらの式は順番に評価されます。ただし変数宣言とその直後の式の間にはピリオドは書かないので注意してください。このような式の並びにおいて、最後の式の値が全体の値となり、これ以外の式が返した値は無視されます。ピリオドは区切りであって終端子ではないということに注意してください。従って式の並びの最後にはピリオドを付けても付けなくてもかまいません。

| box |

```
box := 20@30 corner: 60@90.
box containsPoint: 40@50 → true
```

## 4.6 カスケードメッセージ

Smalltalk には、複数のメッセージを同じレシーバに送信するためのセミコロン (;) を使った記法があります。これは Smalltalk 用語ではカスケードと呼ばれます。

```
式メッセージ1;メッセージ2
```

```
Transcript show: 'Pharo is '.
Transcript show: 'fun '.
Transcript cr.
```

⇔  
等価

```
Transcript
show: 'Pharo is';
show: 'fun ';
cr
```

カスケードメッセージのレシーバは、別のメッセージ送信の結果であるかもしれないということに注意してください。具体的に説明すると、一連のカスケードの中で最初のカスケードメッセージを受け取ったオブジェクトが、カスケードされたすべてのメッセージのレシーバになります。以下の例では、カスケードの最初メッセージは `setX:setY:` です。なぜなら、次にカスケードが続いているのは `setX:setY:` だからです。カスケードメッセージ `setX:setY:` のレシーバは `Point new` によって新たに作られた点オブジェクトであり、`Point` ではありません。次のメッセージ `isZero` は、`setX:setY:` と同じレシーバである点オブジェクトに送信されます。

```
Point new setX: 25 setY: 35; isZero → false
```

## 4.7 まとめ

- メッセージは常にレシーバと呼ばれるオブジェクトに送信されます。レシーバは他のメッセージ送信の結果であることもあります。
- 単項メッセージは、引数を必要としないメッセージです。単項メッセージの形式は `receiver selector` です。
- 二項メッセージは、レシーバと引数という二つのオブジェクトが関与するメッセージです。二項メッセージのセレクトはまた、次の文字: +、-、\*、/、|、&、=、>、<、~、および @ からなります。二項メッセージの形式は `receiver selector argument` です。

- キーワードメッセージは、二つ以上のオブジェクトが関与し、そのセレクトタにコロン (:) が使われているようなメッセージです。  
キーワードメッセージの形式は: receiver **selectorWordOne:** argumentOne **wordTwo:** argumentTwo です。
- **規則 1:** 単項メッセージがまず送信され、次に二項、そして最後にキーワードメッセージが送信されます。
- **規則 2:** 括弧で囲まれたメッセージは他のメッセージよりも先に送信されます。
- **規則 3:** 同じ種類のメッセージがある場合は、評価は左から右の順で行われます。
- Smalltalk では、+ や \* のような伝統的な算術演算子の優先順位はすべて同じです。+ や \* も単なる二項メッセージで、\* の優先順位の方が + よりも高いということはありません。異なる結果を得るには括弧を使わなければなりません。



Part II

# **Developing in Pharo**



# Chapter 5

## Smalltalk オブジェクトモデル

Smalltalk のプログラミングモデルは簡潔で一貫しています：すべてはオブジェクトで、オブジェクト間のやりとりはメッセージを送りあうことによるのみ行われます。一方で、この簡潔さと一貫性が、他の言語に慣れたプログラマにとって難しさの元になっています。この章では、Smalltalk のオブジェクトモデルの中心となる概念を示します。特に、クラスをオブジェクトとして表現することが何をもたらすのかを議論します。

### 5.1 モデルの規則

Smalltalk のオブジェクトモデルは、統一的に適用できるいくつかの簡潔な規則に基づいています。

**Rule 1.** すべてのものはオブジェクトである。

**Rule 2.** すべてのオブジェクトはクラスのインスタンスである。

**Rule 3.** すべてのクラスにはスーパークラスがある。

**Rule 4.** すべてはメッセージ送信で起こる。

**Rule 5.** メソッド探索は継承の連鎖をたどっていく。

では、これらの規則を一つ一つ詳細に見ていきましょう。

### 5.2 すべてのものはオブジェクトである

「すべてのものはオブジェクトである」という信念は、感染力が高いものです。ほんの短い間でも Smalltalk を使えば、この規則がいかに物事を簡単にするか

に、驚くことでしょう。例えば、整数も本物のオブジェクトで、他のあらゆるオブジェクトと同様、メッセージを送ることができます。

```
3 + 4      → 7  "+ 4'を3に送ることで、結果として7が得られます"
20 factorial → 2432902008176640000 "factorialを送ると、大きな数が得られます"
```

20 factorialの内部表現と7の内部表現は実際には異なっていますが、それらはどちらもオブジェクトなので、コード上では — factorialの実装すら — そうしたことを知る必要はありません。

また、この規則によって次の最も重要な帰結が生じます。

クラスもオブジェクトである。

もっと言えば、クラスはセカンドクラスオブジェクトではありません。クラスは正真正銘のファーストクラスオブジェクトで、メッセージを送ったり、インスタクトしたりできます。つまり Pharo は本当の意味でリフレクション可能なシステムだということです。このことが開発者に強力な表現力を与えています。

Smalltalk の実装の深層部には、3種類のオブジェクトがあります。(1) 参照渡しされるインスタンス変数を持つ、通常のオブジェクト、(2) 値渡しされる小さな整数 (*SmallInteger*)、(3) 配列のように連続したメモリ領域を持つインデックス付きオブジェクトです。Smalltalk の美しさは、これら3種類のオブジェクトの違いについて通常は気にする必要がないというところにあります。

## 5.3 すべてのオブジェクトはクラスのインスタンスである

すべてのオブジェクトはクラスを持っています。オブジェクトに class メッセージを送るとわかります。

```
1 class      → SmallInteger
20 factorial class → LargePositiveInteger
'hello' class → ByteString
#(1 2 3) class → Array
(4@5) class  → Point
Object new class → Object
```

クラスはインスタンス変数を通じてインスタンスの構造を定義します。そしてメソッドによってインスタンスの振る舞いを定義します。各メソッドの名前はセレクトと呼ばれます。名前はそのクラスの中でユニークです。

クラスはオブジェクトで、すべてのオブジェクトはクラスのインスタンスであることから、クラスもまた、あるクラスのインスタンスということになり



ます。クラスをインスタンスとするクラスはメタクラスと呼ばれます。クラスを定義するたびに、システムは自動的にメタクラスを生成します。メタクラスは、インスタンスとしてのクラスの構造と振る舞いを定義します。99%は、メタクラスについて考える必要はないでしょう。当面は無視しても大丈夫です。(第13章でメタクラスについて詳しく見ていきます。)

## インスタンス変数

Smalltalkでのインスタンス変数は、そのインスタンスにとってのプライベートな変数です。これは、JavaやC++のインスタンス変数(“フィールド”とか“メンバ変数”とも呼ばれています)が同じクラスの他のインスタンスからのアクセスを許しているのとは対照的です。JavaやC++ではオブジェクトのカプセル化の境界がクラスであるのに対し、Smalltalkではインスタンスになっていると言えます。

Smalltalkでは、“アクセサメソッド”を定義しない限り、同じクラスの二つのインスタンスはお互いのインスタンス変数にアクセスすることができません。他のオブジェクトのインスタンス変数に直接アクセスするための言語構文がないのです。(実際にはリフレクションという仕組みによって、他のオブジェクトからインスタンス変数の値にアクセスすることもできます。こうしたメタプログラミングによって、他のオブジェクトの中身を見ることを目的にしたツールであるオブジェクトインスペクタのようなツールを記述することができます。)

インスタンス変数は、その変数を定義しているクラスやサブクラスのインスタンスメソッドから、名前でもアクセスすることができます。このことから、Smalltalkのインスタンス変数はC++やJavaのプロテクト変数に似ていると言えます。しかしながら、やはりプライベートと見た方が好ましいでしょう。というのも、インスタンス変数をサブクラスからアクセスすることは、Smalltalkでは良くないスタイルと考えられているからです。

### 例

Point→dist: (Method 5.1) メソッドは、レシーバと引数で与えた点(aPoint)との距離を計算します。メソッド内から、レシーバのインスタンス変数xとyに、直接アクセスしています。しかし、もう一つの点であるaPointのインスタンス変数については、xメッセージやyメッセージを送ってアクセスしなければなりません。

Method 5.1: *the distance between two points*

```
Point>>dist: aPoint
  "aPointとレシーバの距離を答える。"
  | dx dy |
  dx := aPoint x - x.
  dy := aPoint y - y.
  ↑ ((dx * dx) + (dy * dy)) sqrt
```

```
1@1 dist: 4@5 → 5.0
```

クラスでのカプセル化ではなく、インスタンスでのカプセル化を選択する決め手となったのは、インスタンスでのカプセル化が同じ抽象化の異なる実装を共存させることができるからです。例えば、`Point>>dist:`メソッドは、引数 `aPoint` がレシーバと同じクラスなのかどうか知る必要もありませんし、気にする必要もありません。引数オブジェクトは極座標かもしれませんし、データベースのレコードかもしれませんし、分散システムの他の計算機上にあるかもしれません。xメッセージやyメッセージに応じてくれる限り、Method 5.1のコードはちゃんと動作するでしょう。

## メソッド

すべてのメソッドはパブリックです。<sup>1</sup>メソッドは、その目的に応じてプロトコルにグループ分けされています。慣例により、よく使われるプロトコルというものがあります。例えば、*accessing* はアクセサメソッド、*initialization* はオブジェクト初期化のメソッドという具合になっています。*private* プロトコルは、外から見るべきでないメソッドをグループ化するのに使われます。しかし、そういった“private”メソッドがあるからといって、実際にメッセージを送ることを防ぐものではありません。

メソッドはそのオブジェクトのすべてのインスタンス変数にアクセスできます。Smalltalk 開発者の中には、アクセサを通してのみインスタンス変数にアクセスすることを好む人もいます。このプラクティスは価値あるものですが、クラスのインターフェースを乱雑にする側面があり、下手をすると外部の世界にプライベートな状態を晒すことにもなります。

## インスタンス側とクラス側

クラスはオブジェクトなので、クラス自身のインスタンス変数やメソッドを持つことができます。これらはクラスインスタンス変数やクラスメソッドと呼ばれていますが、実際のところ普通のインスタンス変数やメソッドと何も変わりません。クラスインスタンス変数は単にメタクラスで定義されたインスタンス変

<sup>1</sup>正確には、ほとんどすべて、です。Pharo では、pvtで始まるセレクトタを持つメソッドはプライベートで、pvtメッセージはselfにのみ送ることができます。しかし、pvtメソッドはあまり使われていません。

数にすぎませんし、クラスメソッドは単にメタクラスで定義されたメソッドにすぎません。

クラスはメタクラスのインスタンスですが、クラスとそのメタクラスは別々のクラスです。しかし、プログラマであるあなたにはおよそ関係ない話です。プログラマが気にかけることは、オブジェクトの振る舞いやそれを生成するクラスを定義することなのです。

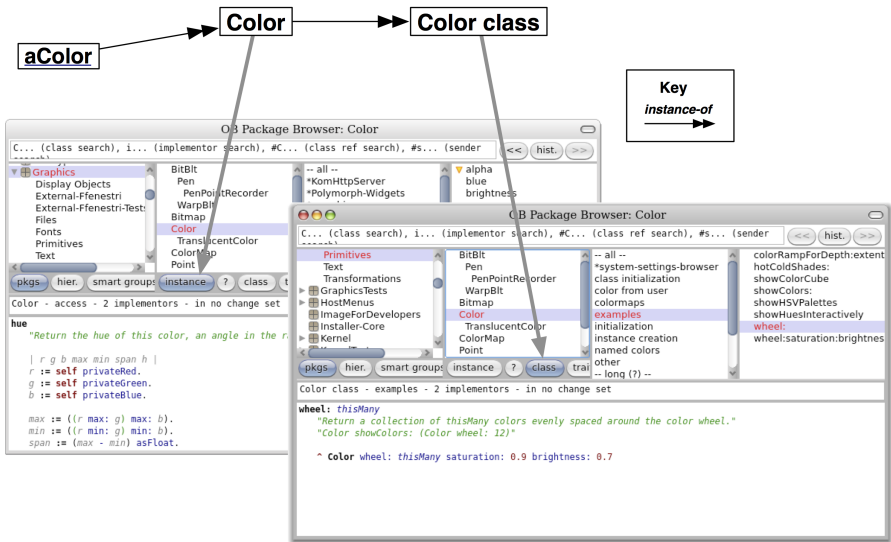


Figure 5.1: クラスとそのメタクラスをブラウズする。

こうしたことから、ブラウザは、クラスとメタクラスの両方を、一つのものの二つの側面として見るできるようになっています。図 5.1 に示されている、“インスタンス側”とクラス側”です。図は Color クラスをブラウズしているところですが、**instance** ボタンをクリックすると、Color のインスタンス (例えば青色) にメッセージが送られたときに実行されるメソッド一覧が表示されます。**class** ボタンを押すと、Color class クラスのブラウズになり、Color クラス自体にメッセージが送られたときに実行されるメソッドを見ることとなります。例えば、Color blue は Color クラスに blue メッセージを送ります。ということは、この blue メッセージは Color のクラス側に定義されているはずですが。インスタンス側ではありません。

aColor := Color blue.	"クラス側のblueメソッド"
aColor → Color blue	
aColor red → 0.0	"インスタンス側のアクセサメソッド"red"
aColor blue → 1.0	"インスタンス側のアクセサメソッド"blue"

新たなクラスの定義は、インスタンス側で提供されるテンプレートを埋め


ることで行えます。このテンプレートをアクセプトすると、システムはクラスだけでなく、対応するメタクラスも生成します。`class` ボタンをクリックするとメタクラスをブラウズできます。メタクラス生成テンプレートで編集するのは、クラスインスタンス変数のリストだけです。

いったんクラスが生成されると、`instance` ボタンのクリックで、そのクラス（およびそのサブクラス）のインスタンス用のメソッドを編集したりブラウズしたりできます。例えば、図 5.1 を見ると、`Color` クラスのインスタンスに `hue` メソッドが定義されています。対して、`class` ボタンを押すと、メタクラス（この場合、`Color class`）のメソッドのブラウズ、定義に切り替わります

## クラスメソッド

クラスメソッドは非常に便利なものです。`Color class` をブラウズすると良い例が見つかります。メソッドには大きく分けて二種類あります。`Color class`→`blue` のように、そのクラスのインスタンスを作るものと、`Color class`→`showColorCube` のように、ユーティリティ的な機能を提供するものです。他の用途に使われているクラスメソッドを見かけることもあるでしょうが、この二つが典型的な用例です。

クラス側にユーティリティ的なメソッドを置くと、余計なインスタンスを生成することなく実行できるので便利です。実際、そういったユーティリティメソッドの多くは、簡単に実行できるコメントを含んでいます。

 `Color class`→`showColorCube` メソッドをブラウズして、コメント "`Color showColorCube`" の引用符の内側をダブルクリックして、`CMD-d` を押してみてください。

このメソッドが実行されたことがわかるでしょう。（元に戻すには `World` ▷ `restore display (r)` を実行してください。）

Java や C++ に慣れている人には、クラスメソッドは静的メソッドに似ているように見えるかもしれませんが、Smalltalk の一貫性の点で、いくぶんの違いがあります。Java の静的メソッドは単に静的に名前解決される手続きに過ぎないのですが、Smalltalk のクラスメソッドは動的に探索されるメソッドです。つまり、Smalltalk ではクラスメソッドを継承したりオーバーライドしたり `super` に送信したりできますが、Java の静的メソッドではできません。

## クラスインスタンス変数

通常のインスタンス変数では、あるクラスのインスタンスはすべて同じ名前のインスタンス変数を持っていて、サブクラスのインスタンスもそれらの名前を継承します。しかし、各インスタンスのインスタンス変数はそれぞれ個別の値を持ちます。インスタンス変数もまったく同様です。各クラスはそれぞれ個別のクラスインスタンス変数を持っています。サブクラスはクラスインスタンス

変数を継承しますが、サブクラスはプライベートなコピーとしてクラスインスタンス変数を保持することになります。オブジェクトがインスタンス変数を共有しないのと同様に、クラスとサブクラスもクラスインスタンス変数を共有しません。

あるクラスが生成したインスタンスの数を管理するために `count` というクラスインスタンス変数を使うことはできますが、そのサブクラスはどれも自分自身の `count` 変数を持つため、サブクラスによって生成されるインスタンスの数は別々に数えられることとなります。

例: クラスインスタンス変数はサブクラスと共有されない。二つのクラス `Dog`(犬) と `Hyena`(ハイエナ) を定義します。 `Hyena` は `Dog` からクラスインスタンス変数 `count` を継承します。

### Class 5.2: 犬クラスとハイエナクラス

```
Object subclass: #Dog
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE-CIV'

Dog class
  instanceVariableNames: 'count'

Dog subclass: #Hyena
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE-CIV'
```

`Dog` の `count` を 0 に初期化するクラスメソッドと、新しいインスタンスが生成されたらそれに 1 を足すクラスメソッドを定義しましょう:

### Method 5.3: 新しい犬の数を数える

```
Dog class>>initialize
  super initialize.
  count := 0.

Dog class>>new
  count := count + 1.
  ↑ super new

Dog class>>count
  ↑ count
```

これで、新しい `Dog` のインスタンス(犬)が生成されるたびに、`count` に 1 が足されていきます。 `Hyena` (ハイエナ) も同様ですが、別々に数えられていきます:

```

Dog initialize.
Hyena initialize.
Dog count    → 0
Hyena count  → 0
Dog new.
Dog count    → 1
Dog new.
Dog count    → 2
Hyena new.
Hyena count  → 1

```

インスタンス変数がインスタンスごとにプライベートなのと同様に、クラスインスタンス変数もクラスごとにプライベートだということにも注意してください。クラスとそのインスタンスは別々のオブジェクトなので、以下のことが言えます。

クラスはそのインスタンスのインスタンス変数にアクセスできない。

インスタンスは、そのクラスのクラスインスタンス変数にアクセスできない。

このことから、インスタンスを初期化するメソッドは常にインスタンス側で定義しなければなりません — クラス側からはインスタンス変数にアクセスできないので、初期化できないのです。クラスができることは、インスタンスを新しく生成するときに、アクセサなどを用いて初期化用のメッセージを送ることだけです。

同様にインスタンスがクラスインスタンス変数にアクセスするには、クラスにアクセサメッセージを送り間接的に行うしかありません。

Java にはクラスインスタンス変数に相当するものではありません。Java と C++ の静的変数は、Smalltalk ではどちらかということ 5.7 節で説明するクラス変数に似ています。すべてのサブクラスとインスタンスが同一の静的変数を共有するというものだからです。

**例: シングルトンを定義する。** シングルトンパターン<sup>2</sup>はクラスインスタンス変数とクラスメソッドを使う典型例です。ctWebServer クラスを実装してみましょう。シングルトンパターンを使い、WebServer クラスには一つしかインスタンスができないようにします。

<sup>2</sup>Sherman R. Alpert, Kyle Brown and Bobby Woolf, *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998, ISBN 0-201-18462-1.

ブラウザの `instance` ボタンをクリックして、`WebServer` クラスを以下のように定義します。(Class 5.4)

#### Class 5.4: シングルト クラス

```
Object subclass: #WebServer
  instanceVariableNames: 'sessions'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Web'
```

そして、`class` ボタンをクリックして、クラス側でクラスインスタンス変数 `uniqueInstance` を追加します。

#### Class 5.5: シングルト クラスのクラス側

```
WebServer class
  instanceVariableNames: 'uniqueInstance'
```

この結果、`WebServer` クラスは、新たなインスタンス変数を追加で持つことになります。 `superclass` や `methodDict` といった変数もスーパークラスから継承しているからです。

さて、これでメソッド `uniqueInstance` を定義できます。Method 5.6 を見てください。このメソッドではまず、`uniqueInstance` が初期化されているかをチェックします。初期化されていない場合は、インスタンスを生成し、クラスインスタンス変数 `uniqueInstance` に代入します。最後に、`uniqueInstance` の値を返しています。`uniqueInstance` はクラスインスタンス変数なので、このクラスメソッド内で直接アクセスすることができます。

#### Method 5.6: `uniqueInstance` (クラス側)

```
WebServer class>>uniqueInstance
  uniqueInstance ifNil: [uniqueInstance := self new].
  ↑ uniqueInstance
```

`Webserver uniqueInstance` が最初に実行されるときには、`WebServer` クラスのインスタンスが生成され、変数 `uniqueInstance` に代入されます。以後は、新しいインスタンスを生成することなしに、前回生成されたインスタンスが返されます。

Method 5.6 の条件分岐の内側にあるインスタンス生成のコードが `self new` という形であって、`WebServer new` ではないことに留意してください。何が違うのでしょうか? `uniqueInstance` メソッドは `WebServer class` に定義されているので、同じことだと思うかもしれませんが、`WebServer` のサブクラスが作られるまでは確かに同じなのです。しかし、もし `WebServer` のサブクラスとして `ReliableWebServer` が作られ、`uniqueInstance` メソッドを継承していたらどうなるのでしょうか。普通は `ReliableWebServer uniqueInstance` の結果は `ReliableWebServer` のインスタンスとなつてほしいはずですが。こうしたときに、`self` を使うことで、そのようにすることができます。`self` はメッセージを受け取るクラス自身だからです。な


お、WebServerと ReliableWebServer はそれぞれ自分のクラスインスタンス変数 `uniqueInstance`を持っていることにも留意してください。それぞれの変数は、もちろん、別々の値になります。

## 5.4 すべてのクラスにはスーパークラスがある

Smalltalk では、クラスは単一のスーパークラス から、振る舞いおよび構造の記述を継承します。つまり Smalltalk は単一継承 です。

SmallInteger superclass	→	Integer
Integer superclass	→	Number
Number superclass	→	Magnitude
Magnitude superclass	→	Object
Object superclass	→	ProtoObject
ProtoObject superclass	→	nil

伝統的には、Smalltalk の継承階層のルートは Object クラスです。(なぜなら、すべてのものはオブジェクトだからです。) Pharo では、ルートは ProtoObject と呼ばれるクラスになっています。しかし、通常はこのクラスを気にする必要はありません。ProtoObject はすべてのオブジェクトが持っていなければならない最低限のメッセージをカプセル化しています。しかしながら、たいていのクラスは Object からの継承となっています。Object はほとんどすべてのオブジェクトが理解し対応すべきメッセージを追加でたくさん定義しています。特別な理由がない限り、アプリケーションのクラスを生成する際には、Object や、そのサブクラスを継承すべきです。

 新しいクラスを生成するには、既存のクラスに `subclass: instanceVariableNames: ...` というメッセージを送ります。他にもいくつかクラスを生成するためのメソッドがあります。Kernel-Classes ▸ Class ▸ subclass creation プロトコルを見て、どのようなメソッドがあるか見てみてください。

Pharo には多重継承はありませんが、お互いに関係のないクラスで同じ振る舞いを共有するためのトレイト というメカニズムをサポートしています。トレイトはメソッドの集合で、継承関係のないクラス間で再利用されます。トレイトを使うと、同じコードを繰り返し定義することなく、別々のクラス間で共有することができます。

### 抽象メソッドと抽象クラス

抽象クラスは、インスタンスを生成するためではなく、サブクラスを定義するために存在するクラスです。抽象クラスは通常は不完全なもので、メソッドの中身すべてが実装されてはいるわけではありません。“空の”メソッド—他のメソッドがそのメソッドがあることを想定しているのに、実装が定義されていないメソッド—を抽象メソッドと呼びます。



Smalltalk には、抽象メソッドや抽象クラスを示す特別な構文はありません。コード規約として、抽象メソッドの本文に `self subclassResponsibility` と書きます。これはいわゆる“マーカーメソッド”で、サブクラスでそのメソッドの具体的な実装を定義する責任があることを示しています。 `self subclassResponsibility` を書いたメソッドは必ずオーバーライドしなければならず、直接実行してはならないものです。もしオーバーライドし忘れたら、メソッド実行の結果、例外が発生してしまいます。

抽象メソッドを持つクラスが抽象クラスです。実際には抽象クラスのインスタンスを生成しないようにする仕組みはありません。抽象メソッドが実行されるまでは、普通に動作します。

### 例: Magnitude クラス

Magnitude は互いを比較するクラスを定義するのに役立つ抽象クラスです。Magnitude のサブクラスは `<および=`、`hash` の三つのメソッドを定義しなければなりません。Magnitude では、それらのメッセージを使って `>および>`、`<=`、`max:`、`min:`、`between:and:` 等のメソッドを定義しています。こうしたメソッドはサブクラスで継承されて使われます。抽象メソッド `<` は、Method 5.7 のようになっています。

#### Method 5.7: Magnitude `<`

```
Magnitude < aMagnitude
    "レシーバが引数より小さいかどうかを答える。"
    ↑self subclassResponsibility
```

対照的に、`>=` は具象メソッドで、`<` を使って定義されています。

#### Method 5.8: Magnitude `>=`

```
>= aMagnitude
    "レシーバが引数と等しいかそれ以上かどうかを答える。"
    ↑(self < aMagnitude) not
```

他の比較メソッドも同様です。

Character は Magnitude のサブクラスで、`subclassResponsibility` となっていた `<` メソッドをオーバーライドし、メソッドを再定義しています (Method 5.9 参照)。Character は `=` メソッドおよび `hash` メソッドも定義していて、`>=` および `<=`、`~=` 等のメソッドを Magnitude から継承しています。

#### Method 5.9: Character `<`

```
Character < aCharacter
    "レシーバが aCharacter より小さければ、true と答える。"
    ↑self asciiValue < aCharacter asciiValue
```

## トレイト

トレイト は、継承なしにクラスの振る舞いに取り込むことができるメソッドの集合です。これによって、単一継承の容易さを保ちながら、複数のクラスにとって有用なメソッドを共通化することができます。

新しいトレイトを定義する方法は、単にサブクラス定義のテンプレートを、Trait 用のメッセージに置き換えるだけです。

### Class 5.10: 新しいトレイト を定義する

```
Trait named: #TAuthor
  uses: {}
  category: 'PBE-LightsOut'
```

ここでは、*PBE-LightsOut* カテゴリに *TAuthor*トレイトを定義しています。このトレイトでは既存のトレイトを使っていません。一般には、`uses:`の引数として、他のトレイトのトレイト 合成式を使うことができます。ここでは単に空の配列にしています。

トレイトはメソッドを持つことができますが、インスタンス変数は持ちません。例として `author`メソッドをクラス階層のどこに位置するかに関係なくいろいろなクラスに追加することを考えてみましょう。以下のように書きます。

### Method 5.11: *author* メソッド

```
TAuthor»author
  "著者の頭文字を返す。"
  ↑'on' "oscar nierstraszの頭文字"
```

クラスに既存のスーパークラスがあっても、このトレイトを使うことができます。第2章で定義した *LOGGame*クラスを例にします。単に、*LOGGame*クラスを定義するテンプレートを書き換え、`uses:`の引数で *TAuthor*を使うように指定するだけです。

### Class 5.12: トレイト を使う

```
BorderedMorph subclass: #LOGGame
  uses: TAuthor
  instanceVariableNames: 'cells'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE-LightsOut'
```

*LOGGame*のインスタンスを生成すると、期待通り `author`メッセージに答えてくれるようになります。

```
LOGGame new author → 'on'
```

トレイト合成式では複数のトレイトを+演算子でつなげることができます。もし複数のトレイトで同じ名前のメソッドを定義しているなどの不整合があっ

ても、-演算子でそうしたメソッドを明示的に除外したり、新規のクラスやトレイト側で再定義したりすることで、解消することができます。@演算子でメソッドにエイリアスとして別名を付けることも可能です。

トレイトはシステムの中核部分で使われています。Behaviorクラスが良い例です。

### Class 5.13: Behaviorの定義で使われているトレイト

Object subclass: #Behavior

uses: TPureBehavior @ {#basicAddTraitSelector:withMethod:->

#addTraitSelector:withMethod:}

instanceVariableNames: 'superclass methodDict format'

classVariableNames: 'ObsoleteSubclasses'

poolDictionaries: "

category: 'Kernel-Classes'

ここでは、TPureBehaviorの addTraitSelector:withMethod:メソッドに、basicAddTraitSelector:withMethod:として別名を付けています。現在、トレイトをサポートする機能がブラウザに追加されているところです。

## 5.5 すべてはメッセージ送信で起こる

この規則は、Smalltalkプログラミングの本質を捉えています。

手続き的プログラミングでは、手続きが呼ばれたときにどのコードが使われるかは、呼び出し側が決めます。呼び出し側が、名前で、静的に、手続きや関数を選んで実行します。

オブジェクト指向プログラミングでは、“メソッドを呼び出し”ません。“メッセージを送信”します。この用語の違いは重要です。オブジェクトにはそれぞれの責任があります。オブジェクトが何をすべきか、その手続き指示するようなことはしません。そうではなく、オブジェクトにメッセージを送ることで、何かをするようにお願いするのです。メッセージはコードではありません。単なる名前と引数の組なのです。メッセージを受けたレシーバは、頼まれたことを実行するため、メソッドを自ら選択して、どう対応するのかを決めます。オブジェクトが異なれば、同じメッセージに対して反応するためのメソッドも異なっていてよいのです。つまりメソッドはメッセージを受け取ったときに動的に選択されることとなります。

3 + 4	→	7	”整数「3」にメッセージ「+」と引数「4」を送信する”
1@2 + 4	→	5@6	”点「1@2」にメッセージ「+」と引数「4」を送信する”

結果として同じメッセージを異なるオブジェクトに送信することができ、オブジェクトはそのメッセージに応えるために各自でメソッドを持つことができます。+4というメッセージにどう反応するのかを SmallInteger 3や Point 1@2に直接指示しないのです。それぞれが+メソッドを持っていて、そのメソッドに従って+4に反応するのです。

Smalltalk のメッセージ送信モデルの帰結として言えることがあります。オブジェクトは非常に小さなメソッドのみを持ち、他のオブジェクトに仕事を移譲する傾向となり、巨大で手続き的なメソッドを保持し、過大な責任をかかえることを避けるようになります。ジョセフ・ペルリン (Joseph Pelrine) はこの原則を次のように簡潔に表現しています。

誰かに任せられる仕事を自分でやらないこと。

多くのオブジェクト指向言語は、オブジェクトに対して静的・動的両方の操作方法を提供していますが、Smalltalk には動的なメッセージ送信しかありません。静的な操作の代わりに、例えばクラスがオブジェクトとなっており、単にクラスにメッセージを送ればよいのです。

Smalltalk では、ほとんどすべてのことはメッセージ送信で起こります。しかし時にはメッセージ送信ではない処理も発生します。

- 変数宣言はメッセージ送信ではありません。実際、変数宣言は実行可能ですらありません。変数宣言は、オブジェクトの参照を格納する領域を確保する意味しか持ちません。
- 代入はメッセージ送信ではありません。変数への代入はそれを定義したスコープ内で名前を束縛します。
- リターンはメッセージ送信ではありません。リターンは計算結果をメッセージの送り手に返します。
- プリミティブはメッセージ送信ではありません。プリミティブは仮想マシン側で実装されています。

これらのわずかな例外を除き、ほとんどすべてはメッセージ送信で起こります。特に、Smalltalk には“パブリックなフィールド”はないので、他のオブジェクトのインスタンス変数の値を変えるには、メッセージを送ってそのオブジェクト自身のフィールドを更新するようお願いするしかありません。もちろん、すべてのインスタンス変数についてセッターやゲッターを定義するのは、オブジェクト指向として良いスタイルとは言えません。ジョセフ・ペルリンは次のようにも言っています。

自身のデータを他人にいじらせないこと。

## 5.6 メソッド探索は継承の連鎖をたどっていく

オブジェクトがメッセージを受け取ると、何が起こるのでしょうか？

その過程はとてもシンプルです。メッセージを受けたレシーバのクラスは、メッセージを処理するのに必要なメソッドを探索します。もしクラスがそのメソッドを持っていなければ、自身のスーパークラスに尋ねます。継承の連鎖をたどって、メソッドを探していくのです。メソッドが見つければ、渡した引数の値はそのメソッドの引数に束縛され、仮想マシンがメソッドを実行します。

メソッド探索は基本的には簡単なことなのですが、いくつか注意しなければなりません。

- メソッドが明示的に値を返していない場合は、どうなるのでしょうか？
- あるクラスがスーパークラスのメソッドを再実装している場合は、どうなるのでしょうか？
- self 送信と super 送信は何が違うのでしょうか？
- メソッドが見つからない場合には、どうなるのでしょうか？

ここでは、メソッド探索の概念的な規則を示します。仮想マシンを実装する人はありとあらゆるトリックや最適化を駆使してメソッド探索を高速化しますが、それは彼らに任せておきましょう。ここで示す規則から外れているとはわからないように、やっているはずです。

まず、探索の基本戦略から見ていきましょう。その上で、上記の一連の疑問について考察します。

## メソッド探索

EllipseMorphのインスタンスを生成してみます。

```
anEllipse := EllipseMorph new.
```

ここで、このオブジェクトに defaultColorメッセージを送ると、Color yellowという結果が得られます。

```
anEllipse defaultColor → Color yellow
```

EllipseMorphクラスは defaultColorを実装しているので、適切なメソッドが直ちに見つかります。

### Method 5.14: レシーバのクラス自体に実装されたメソッド

```
EllipseMorph»defaultColor
"レシーバのデフォルトの色/塗り方を答える"
↑ Color yellow
```

一方、anEllipseに openInWorldメッセージを送ると、メソッドはすぐには見つかりません。anEllipseクラスは openInWorldを実装していないからです。そのた

め、スーパークラスの BorderedMorph を探索していきます。探索は openInWorld メソッドが Morph クラスで見つかるまで続きます。

Method 5.15: 継承されたメソッド

```
Morph»openInWorld
"このモーフをワールド"に追加する。"

self openInWorld: self currentWorld
```

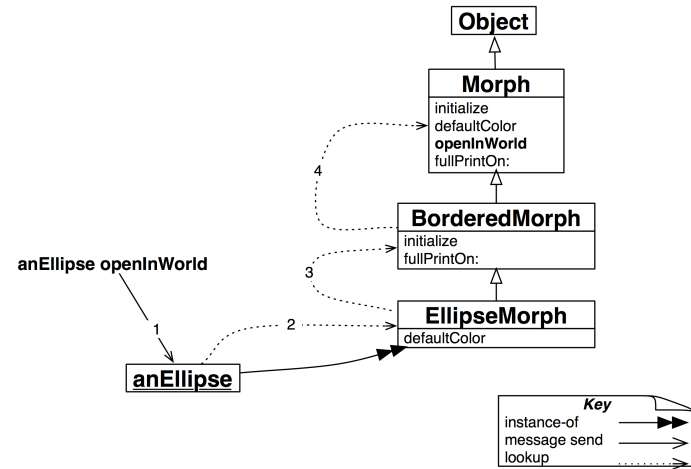


Figure 5.2: メソッド探索は継承の連鎖をたどっていく

### self を返す

EllipseMorph»defaultColor (Method 5.14) は Color yellow を明示的に返している一方で、Morph»openInWorld (Method 5.15) には戻り値を示すものがないことに注目してください。

実際にはメソッドは常に値を返します。もちろん、値というのはオブジェクトです。答える値はメソッド中の↑で決まることもありますが、↑を実行することなくメソッドの終わりまで来てしまったとしても、やはりメソッドは値を返します。その場合、メソッドは、そのメッセージを受け取ったオブジェクト自身を返すのです。これを、“self を返す”と言います。Smalltalk では擬似変数 self はメッセージを受けたレシーバそのものを指します。Java の this のようなものです。

つまり、Method 5.15 は Method 5.16 と同じことです。

## Method 5.16: 明示的に self を返す

Morph»openInWorld

"このモーフをワールド"に追加する。"

self openInWorld: self currentWorld

↑self "不要なので、通常はわざわざ書かないこと！"

なぜ上のように↑selfと書かない方がよいのかというと、明示的に値を返す書き方は、メッセージを送った側に何か意味あるものを返していることを示すものだからです。self を明示的に返す場合、センダ側がその戻り値を使うことを期待しているといった意味になるのです。上記のコードではそういう意図はないので、self を返すことを明示しない方がよいでしょう。

これは Smalltalk では日常的なイディオムで、ケント・ベックは“興味を引くような戻り値 (Interesting return value)”<sup>3</sup>と呼んでいます。

センダにその値を使ってほしいときにだけ、戻り値を示すこと。

## オーバーライドと拡張

図 5.2 の EllipseMorph クラスの階層を見ると、Morph クラスと EllipseMorph クラスのどちらも defaultColor を実装しています。実際、Morph new openInWorld で新しいモーフを開くと、青い色になっています。楕円の場合には黄色になります。

これを、EllipseMorph が Morph から継承した defaultColor メソッドをオーバーライドしている、と言います。継承したメソッドは、anEllipse からは見えなくなります。

継承したメソッドをオーバーライドするのではなく、機能を追加して拡張したいこともあります。つまり、オーバーライドされるメソッドに、サブクラスで新たな機能を付け加えて実行したいということです。Smalltalk では、super に対するメッセージ送信でこれを実現しています。単一継承をサポートする多くのオブジェクト指向言語で典型的な方法です。

initialize メソッドは、この仕組みの最も重要な例です。あるクラスの新しいインスタンスを初期化するときには、スーパークラスから継承したインスタンス変数も必ず初期化しなければなりません。しかし、その初期の仕方は、継承の連鎖の中で各スーパークラスの initialize メソッドに既に記述されています。サブクラスは継承したインスタンス変数の初期化には本来関わりがありません。

以上のことから、initialize メソッドを実装するときには、まず super initialize を送信してから、他の変数の初期化を書くようにするのが良いプラクティスです。

<sup>3</sup>Kent Beck, *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

Method 5.17: *Super initialize*

```
BorderedMorph»initialize
  "レシーバの状態を初期化する"
  super initialize.
  self borderInitialize
```

initializeメソッドは、super initializeで始めること。

**self、super へのメッセージ送信**

super へのメッセージ送信は、オーバーライドされてしまう振る舞いをサブクラス側で組み入れるのに使われます。継承されたものであろうとなかろうと、複数のメソッドを組み合わせるには、通常 self へのメッセージ送信を用います。

self 送信と super 送信の違いは何でしょうか? self と同様に、super はメッセージを受けたレシーバ自身を表します。違うのは、メソッド探索だけです。super 送信があると、レシーバのクラスから探索を始めるのではなく、super への送信を行っているメソッドのクラスのスーパークラスから探索を始めます。

super はスーパークラスではないことに注意してください! スーパークラスと思うのは、よく見られる勘違いです。また、レシーバのスーパークラスから探索を始めるというのも、間違いです。どういうことか、以下に例を示します。

constructorString というメッセージを例にしましょう。これは、任意のモーフに送信することができます。

```
anEllipse constructorString → '(EllipseMorph newBounds: (0@0 corner: 50@40)
  color: Color yellow)'
```

戻り値は、評価することでモーフを復元できる文字列です。

self 送信と super 送信をどう組み合わせれば、この文字列が得られるのでしょうか? まず、anEllipse constructorString が実行されると、Morph クラスの constructorString が見つかります。図 5.3 を見てください。

## Method 5.18: self 送信

```
Morph»constructorString
  ↑ String streamContents: [:s | self printConstructorOn: s indent: 0].
```

Morph»constructorString メソッドは printConstructorOn:indent: を self に送っています。このメッセージもまた探索されますが、探索は EllipseMorph から始まって、Morph で見つかります。そのメソッドの内部で、さらに printConstructorOn:indent:nodeDict: が self に送られます。それが fullPrintOn: を self に送ります。fullPrintOn: は EllipseMorph から探索を始め、fullPrintOn: が BorderedMorph で見つかります。(もう一度、図 5.3 を見てください)。重要なのは、self への送



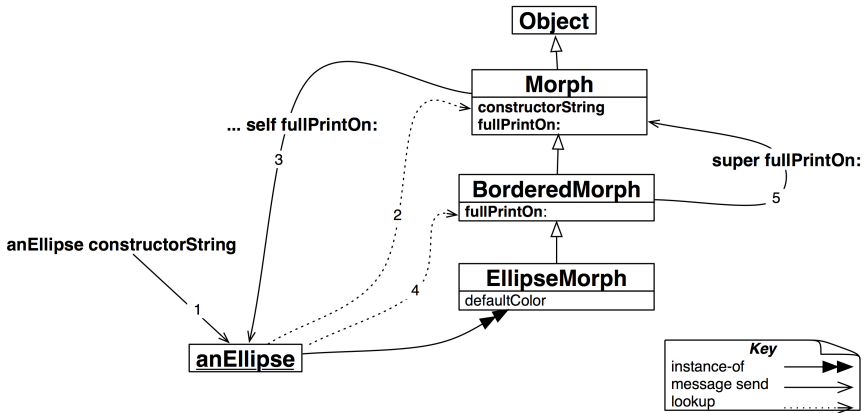


Figure 5.3: self 送信と super 送信

信ではレシーバのクラス、この場合には anEllipseのクラスから、メソッド探索が始まるということです。

self 送信は、動的なメソッド探索を、レシーバのクラスから開始します。

Method 5.19: super 送信と self 送信を組み合わせる

```

BorderedMorph»fullPrintOn: aStream
  aStream nextPutAll: '('.
  super fullPrintOn: aStream.
  aStream nextPutAll: ') setBorderWidth: '; print: borderWidth;
  nextPutAll: ' borderColor: ', (self colorString: borderColor)

```

一方、BorderedMorph»fullPrintOn:では、super へメッセージ送信をして、スーパークラスから継承した fullPrintOn:の振る舞いを拡張しています。super への送信なので、メソッド探索は super 送信を行うクラスのスーパークラスである Morph クラスから始まります。すると、Morph»fullPrintOn:が即座に見つかるので、それが評価されます。

super 送信の探索はレシーバのスーパークラスから開始しているわけではない、ということに注意してください。もしレシーバのスーパークラスから探索を開始すると、BorderedMorphから探索を始めるということになり、無限ループになってしまいます!

super 送信は、静的なメソッド探索を、super 送信を行っているメソッドを定義しているクラスのスーパークラスから開始します。

super 送信の解説を踏まえ、図 5.3 を見ると、super は静的束縛だということがわかります。super 送信をしているコードがどのクラスにあるかにより、すべて決まります。対照的に、self 送信は動的です。今実行しているメッセージのレシーバを表しています。つまり、self に送られたすべてのメッセージは、そのレシーバのクラスから探索を始めます。

## 理解されないメッセージ

探索してもメソッドが見つからなかったら、どうなるのでしょうか？

楕円に foo メッセージを送ってみましょう。まず、foo メソッドを探して Object (正確には ProtoObject) まで継承の連鎖を一通りたどる、通常の探索が開始されます。メソッドが見つからない場合、仮想マシンからそのオブジェクトに self doesNotUnderstand: #foo メッセージが送信されます。(図 5.4 を参照)

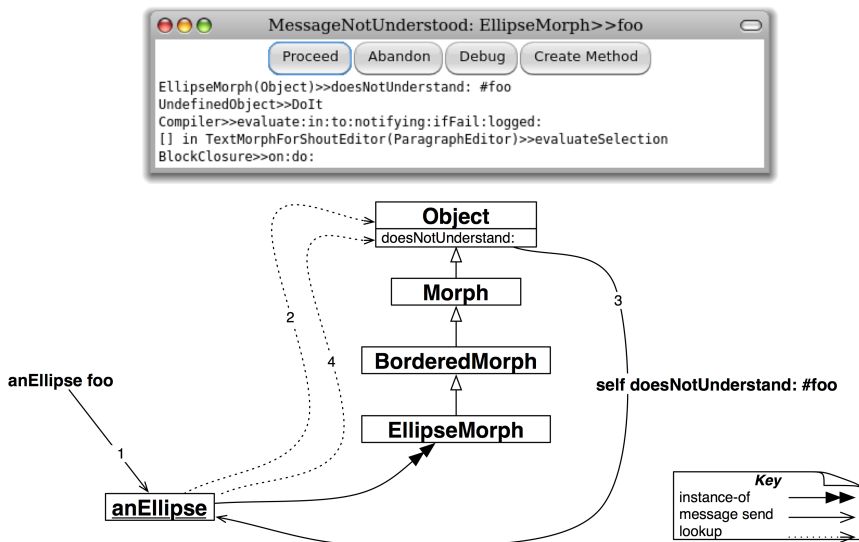


Figure 5.4: Message foo is not understood

これも通常の動的なメッセージ送信なので、また EllipseMorph から探索が始まります。ただし、今度は doesNotUnderstand: メソッドを探します。結局のところ、Object が doesNotUnderstand: を実装しています。このメソッドでは、MessageNotUnderstood オブジェクトが生成されます。MessageNotUnderstood オブ

ジェクトは、現在実行しているコンテキストでデバッグを立ち上げることができます。

なぜ明らかなエラーに対してこんなに複雑なことをするのかというと、こうすると開発者がエラーを捕まえて別のアクションをさせることが容易になるからです。Objectの任意のサブクラスは `doesNotUnderstand:` をオーバーライドすることで、別のやり方でエラー処理を行うことができます。

実際、あるオブジェクトから別のオブジェクトに自動的にメッセージを移譲するように簡単に実装できます。Delegatorオブジェクトとして、実装していないメッセージを別のオブジェクトに移譲して改めてメッセージを処理させたり、エラーを発生させたりすることも可能です。

## 5.7 共有変数

ここでは Smalltalk の特徴の中で、五つの規則ではカバーされていないものを見ていきましょう。共有変数のことです。

Smalltalk には3種類の共有変数があります：(1) グローバルな共有変数 (2) インスタンスとクラスの間で共有される変数 (クラス変数) (3) クラスのグループの間で共有される変数 (プール変数) です。これらの共有変数の名前は大文字で始まります。これは、変数が複数のオブジェクトの間で共有されているというこの警告となっています。

### グローバル変数

Pharo では、すべてのグローバル変数は Smalltalk という名前空間に格納されます。Smalltalk は SystemDictionary クラスのインスタンスとして実装されています。グローバル変数はどこからでもアクセス可能です。各クラスはグローバル変数で名付けられています。特別なオブジェクトや共通して使われるオブジェクトもグローバル変数で名前が付いています。

変数 Transcript (トランスクリプト) は TranscriptStream のインスタンスを指しています。clsindTranscriptStream はスクロール付きのウィンドウに書き込むストリームです。以下のコードは、Transcriptに情報を表示して改行します。

```
Transcript show: 'Pharoは楽しくてパワフル!'; cr
```

do it する前に、**World ▸ Tools ... ▸ Transcript** でトランスクリプトを開いておくください。

**HINT** トランスクリプトへの書き込みは遅いです。トランスクリプトウィンドウが開いているときはなおさらです。トランスクリプトに書き込んでいて遅いと思ったときは最小化してみるのも一つの手です。

## 他の便利なグローバル変数

- Smalltalkは SystemDictionaryのインスタンスです。Smalltalk自身を含むすべてのグローバル変数を定義してします。この辞書のキーはシンボルで、Smalltalk のコード内でグローバル変数を示すのに使われます。例えば、

```
Smalltalk at: #Boolean → Boolean
```

Smalltalkはそれ自身がグローバル変数なので、

```
Smalltalk at: #Smalltalk → a SystemDictionary(lots of globals)
```

さらに、

```
(Smalltalk at: #Smalltalk) == Smalltalk → true
```

となります。

- Sensor は EventSensor のインスタンスで、Pharo への入力を表しています。例えば、Sensor keyboard はキーボードからの次の入力を答えます。そして、Sensor leftShiftDownは、左シフトキーが押されているときに true と答えます。さらに、Sensor cursorPointは現在のマウスの位置を示す Pointを返します。
- World は PasteUpMorph のインスタンスで、スクリーンを表します。World boundsはスクリーン全体の範囲を示す矩形を答えます。すべてのモーフは Worldのサブモーフです。
- ActiveHand は HandMorph の現在のインスタンスで、カーソルのグラフィックを表します。マウスでモーフをドラッグしている間は ActiveHandのサブモーフになっています。
- Undeclared も辞書です。宣言されていない変数群を保持しています。宣言されていない変数を参照するメソッドを書くと、通常はブラウザが、グローバル変数やインスタンス変数として宣言するよう促します。それでも、変数の宣言を後で削除してしまった場合には、コードは未宣言の変数を参照することになります。Undeclaredをインスペクトすることで、奇妙な振る舞いの原因がわかることもあるでしょう。
- SystemOrganization は SystemOrganizer のインスタンスです。クラスの構成情報をまとめて保持しています。より正確に言えば、クラス名をカテゴリに分類します。

```
SystemOrganization categoryOfElement: #Magnitude → #'Kernel-Numbers'
```

グローバル変数は極力使わないというのが現在のプラクティスとなっています。通常はクラスインスタンス変数やクラス変数を使い、クラスメソッドでアクセスする方が望ましいです。もし Pharo を一から再実装するようなことが

あれば、グローバル変数は、クラス群を除いては、ほとんどシングルトンに置き換えられるでしょう。

通常、グローバル変数を定義するには、`do it`で、大文字で始まる未宣言の識別子に代入するだけです。すると、パーサがグローバル変数を定義するよう促します。プログラムでグローバル変数を定義するには、`Smalltalk at: #AGlobalName put: nil`を実行します。削除するには、`Smalltalk removeKey: #AGlobalName`です。

## クラス変数

クラスおよびクラスのすべてのインスタンスで、データを共有したいときがあります。これはクラス変数で可能です。クラス変数という用語は、変数のライフタイムがクラスと同じということを示しています。しかし、その用語の字面からは、クラス変数はクラスの間だけでなく、インスタンスとも共有されるということではありません。図 5.5 を見てください。実際のところ、共有変数という名称の方が適切だったでしょう。役割がより明らかになりますし、使う際(特に変更する場合)の危険性も伝わります。

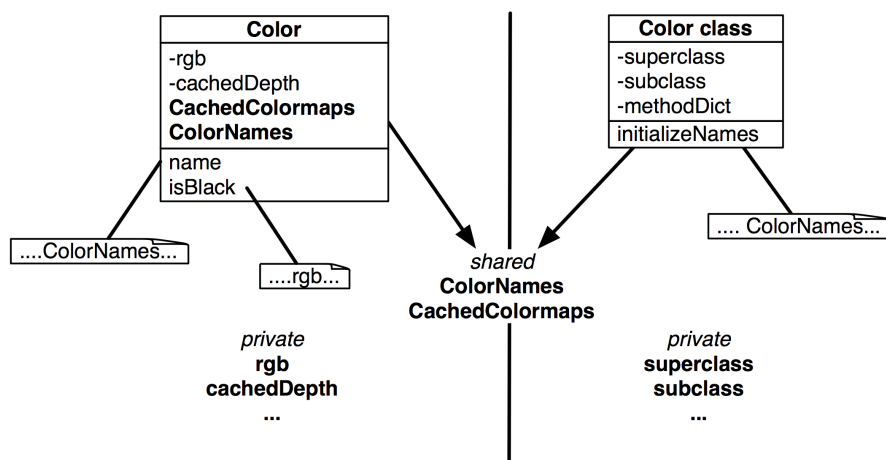


Figure 5.5: インスタンスメソッドとクラスメソッドが別々の変数にアクセスしている。

図 5.5 では、`rgb`と `cachedDepth`は `Color`のインスタンス変数であり、そのため、`Color`のインスタンスからアクセス可能です。また、`superclass`や `subclass`、`methodDict`等はクラスインスタンス変数なので、`Color`クラスからのみアクセスできます。

新しく出てきたのは `ColorNames`と `CachedColormaps`のクラス変数で、`Color`に定義されています。変数が大文字であることが、共有されているというヒントを与えてくれます。実際、`Color`の全インスタンスからこれらの共有変数にアク

セスでき、さらに `Color` クラス自身とそのすべてのサブクラスからもアクセス可能です。インスタンスメソッドとクラスメソッドの両方からこれらの共有変数にアクセスできるのです。

クラス変数はクラス定義テンプレートの中で宣言されます。例えば、`Color` クラスでは大量のクラス変数を定義しており、色オブジェクトの生成を高速化しています。その定義を下図 (Class 5.20) に示します。

### Class 5.20: `Color` とクラス変数

```
Object subclass: #Color
  instanceVariableNames: 'rgb cachedDepth cachedBitPattern'
  classVariableNames: 'Black Blue BlueShift Brown CachedColormaps ColorChart
  ColorNames ComponentMask ComponentMax Cyan DarkGray Gray
  GrayToIndexMap Green GreenShift HalfComponentMask HighLightBitmaps
  IndexedColors LightBlue LightBrown LightCyan LightGray LightGreen LightMagenta
  LightOrange LightRed LightYellow Magenta MaskingMap Orange PaleBlue
  PaleBuff PaleGreen PaleMagenta PaleOrange PalePeach PaleRed PaleTan
  PaleYellow PureBlue PureCyan PureGreen PureMagenta PureRed PureYellow
  RandomStream Red RedShift TranslucentPatterns Transparent VeryDarkGray
  VeryLightGray VeryPaleRed VeryVeryDarkGray VeryVeryLightGray White Yellow'
  poolDictionaries: ''
  category: 'Graphics-Primitives'
```

クラス変数 `ColorNames` は、よく使われる色の名前を持つ配列です。この配列は `Color` とそのサブクラス `TranslucentColor` の、すべてのインスタンスで共有されていて、インスタンスメソッドやクラスメソッドからアクセス可能です。

`ColorNames` は `Color class>>initializeNames` で一度だけ初期化され、`Color` のインスタンスからアクセスされます。`Color>>name` メソッドで、色の名前を探すためにこの変数が使われています。このようになっているのは、ほとんどの色は名前を持たないため、インスタンス変数 `name` を色ごとに持つのは不適切と考えたからです。

### クラスの初期化

クラス変数があることで、どうやってそれを初期化するかが問題になります。一つの解は、初期化を遅延させることです。これは、クラス変数へのアクセサメソッドを実装し、その中で変数が初期化されていなければ初期化を行うように書くことで実現できます。この方法の場合、常にアクセサを使い、クラス変数には絶対に直接アクセスしないようにすることになります。また、アクセサ実行の際に、初期化の必要性をチェックするコストが上乗せされます。さらに言えば、アクセサを使った場合、変数は事実上もはや共有されていないことになるので、クラス変数を使う意味も薄れます。

### Method 5.21: `Color class>>colorNames`

```
Color class>>colorNames
  ColorNames ifNil: [self initializeNames].
```

## ↑ ColorNames

もう一つの解は、クラスメソッド `initialize` をオーバーライドすることです。

Method 5.22: *Color class*»*initialize*

```
Color class»initialize
...
self initializeNames
```

この方法の場合、`initialize` メソッドを定義したら、`Color initialize` といった具合に明示的に実行しておくのを忘れないようにします。クラス側の `initialize` メソッドは、コードをメモリーにロードしたときには自動的に実行されますが、ブラウザで入力してコンパイルしたときや編集して再コンパイルしたときには、自動的に実行されないのです。

## プール変数

プール変数は、直接継承関係にないクラスの間でも共有される変数です。プール変数は元々はプール辞書に格納されていましたが、今では専用のクラス (`SharedPool` のサブクラス) のクラス変数として定義しなければなりません。ここで忠告しておきますが、なるべく使うのを避けてください。プール変数が必要になることは非常にまれであり、特定の状況下だけです。以後は、プール変数を使ったコードを読むのに必要十分な説明にとどめておきます。

プール変数にアクセスするにはクラス定義でそのプールを示さなければなりません。例えば、`Text` クラスではプール辞書 `TextConstants` を使うことを示しています。プール辞書 `TextConstants` は `CR` や `LF` といった、テキストに関するすべての定数を保持しています。`#CR` がキーで、値として `Character cr`、つまり改行文字が結び付けられています。

Class 5.23: *Text* クラスのプール辞書

```
ArrayedCollection subclass: #Text
  instanceVariableNames: 'string runs'
  classVariableNames: ''
  poolDictionaries: 'TextConstants'
  category: 'Collections-Text'
```

こうすることで、`Text` クラスのメソッドは、メソッド本文で直接プール辞書のキーにアクセスすることができます。つまり、明示的に辞書に聞かずとも、変数を参照する構文でアクセスできます。例えば、以下のようにしてメソッドを書けます。

Method 5.24: *Text*»*testCR*

```
Text»testCR
↑ CR == Character cr
```

繰り返します。プール変数やプール辞書はなるべく使わないことをお勧めします。

## 5.8 章のまとめ

Pharo のオブジェクトモデルはシンプルで一貫したものです。すべてはオブジェクトで、ほとんどすべてのことはメッセージ送信で起こります。

- すべてのものはオブジェクトである。整数のようなプリミティブな存在もオブジェクトであり、クラスもファーストクラスのオブジェクトです。
- すべてのオブジェクトはクラスのインスタンスである。クラスは、プライベートなインスタンス変数によってオブジェクトの構造を定義します。そして、パブリックなメソッドによって振る舞いを定義します。各クラスはそれぞれのメタクラスの唯一のインスタンスです。クラス変数はクラスとそのすべてのインスタンスに共有されるプライベートな変数です。クラスはそのインスタンスのインスタンス変数に直接アクセスすることはできません。そしてインスタンスもクラスインスタンス変数にアクセスできません。そういったアクセスが必要ならば、アクセサを定義しなければなりません。
- すべてのクラスにはスーパークラスがある。単一継承の階層のルートは `ProtoObject` です。しかし、クラスを定義するときには、通常は `Object` クラスかそのサブクラスから継承します。抽象クラスを定義する特別な構文はありません。単に抽象メソッドを持てば抽象クラスとみなされます。抽象メソッドは `self subclassResponsibility` のみからなるメソッドです。Pharo は単一継承のみをサポートしていますが、共有したいメソッド群をトレイトとしてパッケージングすることで、それらを簡単に共有できます。
- すべてはメッセージ送信で起こる。「メソッドを呼び出す」のではなく、「メッセージを送信」します。するとレシーバがそのメッセージに対応するメソッドを選択します。
- メソッド探索は継承の連鎖をたどっていく。self 送信は動的で、レシーバのクラスからメソッド探索を始めます。一方、super 送信は静的で、super 送信を記述しているクラスのスーパークラスから探索を始めます。
- 3種類の共有変数がある。グローバル変数は、システムのどこからもアクセス可能です。クラス変数はクラスとそのサブクラスとインスタンス間で共有されます。プール変数は指定したクラス間で共有されます。共有変数の使用は、できるだけ避けるべきです。



## Chapter 6

# Pharo のプログラミング環境

この章の目的は、Pharo のプログラミング環境を使い、どのようにプログラムを開発するかを示すことです。既にブラウザを使ってメソッドやクラスをどのように定義するかについて見てきたわけですが、この章ではブラウザのその他の機能について示し、他のいくつかのブラウザについて紹介していきます。

もちろん作ったプログラムが期待したように動作しないのはよくあることです。Pharo には優れたデバッガがあります。最も役立つツールではありますが、最初に使うときは悩むかもしれません。デバッグの手順について説明し、いくつかのデバッガの機能について実演します。

Smalltalk 固有の特徴の一つは、プログラミングをしているとき、静止したプログラムテキストの世界ではなく、動いているオブジェクトからなる世界にいるということです。このことにより、プログラミング中に非常に素早いフィードバックを得ることが可能で、このフィードバックによって生産性がより向上します。動いているオブジェクトを見たり、実際に変更したりするために二つのツールがあります。それはインスペクタとエクスプローラです。

ファイルやテキストエディタではなく、動いているオブジェクトからなる世界でプログラミングを行っているため、Smalltalk イメージファイルからプログラムを取り出すために、何らかの明示的な作業をしなければいけません。この作業のための昔からのやり方は、ファイルアウトやチェンジセットを作ることで、すべての Smalltalk の方言でサポートされています。これらは、基本的に他のシステムで読み込めるようにコード化されたテキストファイルです。Pharo 上でプログラムを取り出すための新しい方法は、サーバー上のバージョン管理されたりポジトリにコードをアップロードすることです。このために、Monticello と呼ばれるツールを使います。これは特にチームで作業する際に、よりパワフルで効果的な手段となります。

## 6.1 概要

Smalltalk と現在のグラフィカルインターフェースは一緒に発展してきました。Smalltalk の最初の公開リリースである 1983 年以前でさえ、Smalltalk は独自のグラフィカルな開発環境を備え、すべての Smalltalk の開発はその上で行われてきました。では、Pharo の主なツールを見ていきましょう。

- **Browser** (ブラウザ) は、中心的な開発ツールです。クラスやメソッドの作成、定義、構築を行う際にこのツールを使います。これを使うことですべてのクラスライブラリもブラウズできます。ソースコードを別々のファイルに保存する他の環境とは異なり、Smalltalk ではすべてのクラスやメソッドがイメージファイルに含まれます。
- **Message Names** (メッセージネーム) ツールを使うことにより、特定のセレクトタもしくは部分文字列を含んだセレクトタを持つすべてのメソッドを探することができます。(訳注: Pharo 1.4 では Finder ツールに統合されており、ドロップダウンメニューで Selectors を選択することで利用できます)
- **Method Finder** (メソッド・ファインダ) ツールは、メソッドを探すのに役立ちますが、メソッド名と同様に、何をするのかによりメソッドを検索します。(訳注: Pharo 1.4 では Finder ツールに統合されており、ドロップダウンメニューで Examples を選択することで利用できます)
- **Monticello Browser** (Monticello ブラウザ) は、Monticello パッケージからコードを読み込んだり、保存したりするための出発点です。
- **Process Browser** (プロセスブラウザ) は、Smalltalk 上で実行されているすべてのプロセス (スレッド) を見るのに役立ちます。
- **Test Runner** は、SUnit テストを実行、デバッグすることができます。詳細については第 7 章を参照してください。
- **Transcript** (トランスクリプト) は、Transcript 出力ストリームに対するウィンドウで、1.5 節で既に述べた通り、ログメッセージを書くのに役立ちます。
- **Workspace** (ワークスペース) は入力することができるウィンドウです。これはいろいろな目的に使えますが、最もよく使われるのは Smalltalk 式を入力して、`do it` で実行することです。workspace の使い方は 1.5 節にも図解されています。

**Debugger** (デバッガ) は名前からわかる通り明確な役割を持っています。しかし、他のプログラミング言語のデバッガと比べて、より重要な役割を担っていることに気づくでしょう。Smalltalk では debugger 上でプログラムできる

のです。デバッグはメニュー画面から起動されるのではなく、たいてい、失敗するテストプログラムを実行したり、`CMD-` と入力して起動しているプロセスを中断したり、あるいは、コード中に `self halt` 式を挿入したりすることで、起動されます。

## 6.2 ブラウザ

Smalltalk の歴史と共に、長年にわたり多くの異なるクラスブラウザが開発されてきました。Pharo では、様々なビューを一つにまとめたブラウザを提供することで、この物語を単純にしました。図 6.1 は最初にブラウザを開いたときに表示されるものです。<sup>12</sup>

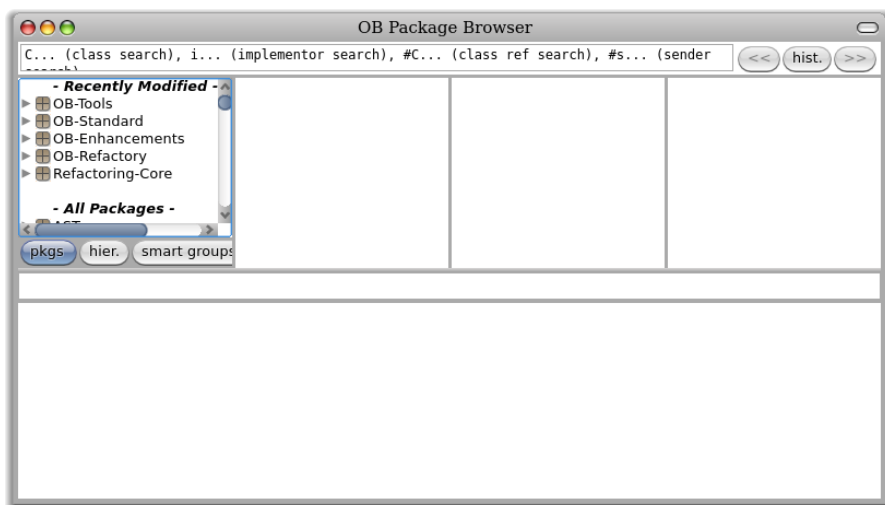


Figure 6.1: ブラウザ

ブラウザ上部の四つの小さなペインは、システム内メソッドの階層ビューを表しています。これは、NeXTstep の *File Viewer* や Mac OS X の *Finder* のカラムモードといった、ディスク上のファイルを見るためのビューと同様のものです。左端のペインはクラスのパッケージリストです。そのうちの一つ (例え

<sup>1</sup>もしブラウザが図 1.12 のように表示されない場合、デフォルトのブラウザを変更する必要があるかもしれないことを思い出してください。FAQ 5, p. 326 参照

<sup>2</sup>訳注：Pharo のデフォルトブラウザは何度も変更されています。バージョン 1.3 では、OBSystemBrowser が使用されており、この章の説明とは異なる場合があります。バージョン 1.4 ではデフォルトブラウザは SystemBrowser であり、OBSystemBrowser は別途インストールする必要があります。バージョン 1.4 Summer では OBSystemBrowser がデフォルトブラウザになっています

ば *Kernel*) を選択すると、すぐ右隣にあるペインにパッケージ中のすべてのクラスが表示されます。

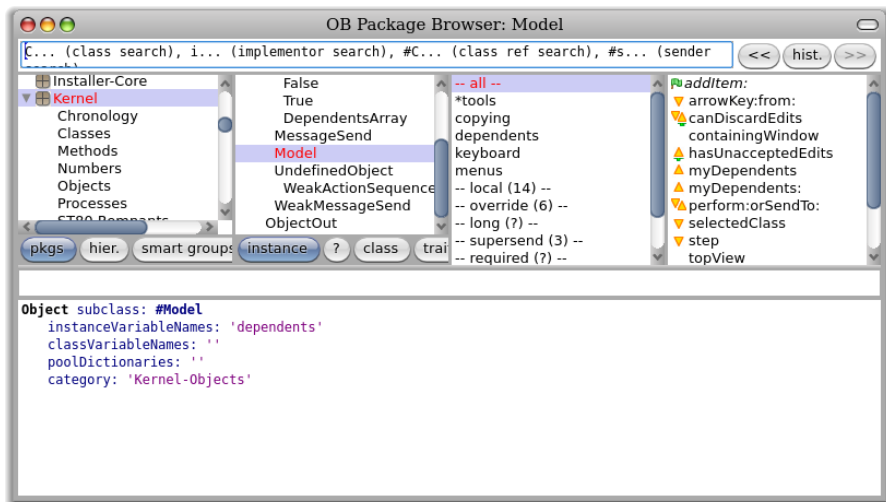


Figure 6.2: ブラウザでModelクラスを選択

同様に、左から2番目のペインでクラスの一つ、例えば *Model* (図 6.2 参照) を選ぶと、その右隣の3番目のペインにクラスに対して定義されたプロトコルが表示され、デフォルトで仮想プロトコル *--all--* が選択されます。プロトコルとはメソッドをカテゴリ化するためのもので、クラスの振る舞いを概念的に首尾一貫して細分化することで、それらを探したり考えたりするのに役立ちます。左から4番目のペインは、選択したプロトコルで定義されているすべてのメソッド名が表示されます。ここでメソッド名を選ぶと、そのメソッドのソースコードがブラウザ下部にある大きなペインに表示されます。このペインではソースコードのブラウザや編集、編集後の保存ができます。ブラウザで *Model* クラス、*dependsnts* プロトコル、そして *myDependents* を選択していくと、図 6.3 のように表示されるはずですが。

Mac OS X の *Finder* とは違い、ブラウザ上部四つのペインは完全に同じものというわけではありません。クラスやメソッドは *Smalltalk* 言語の一部であるのに対し、パッケージとプロトコルはそうではなく、これらはブラウザの各ペインに表示すべき情報量を制限するために導入されたものです。例えば、もしプロトコルがなければ、ブラウザは選択したクラスのすべてのメソッドのリストを表示する必要があります。多くのクラスでは、このようなリストは大きすぎて扱うのが不便になってしまいます。

このことから、新しいパッケージやプロトコルを作る方法は、新しいクラスやメソッドを作る方法とは異なります。新しいパッケージを作るにはパッケージペインでアクションクリックし *new package* を選びます。新しいプロトコル

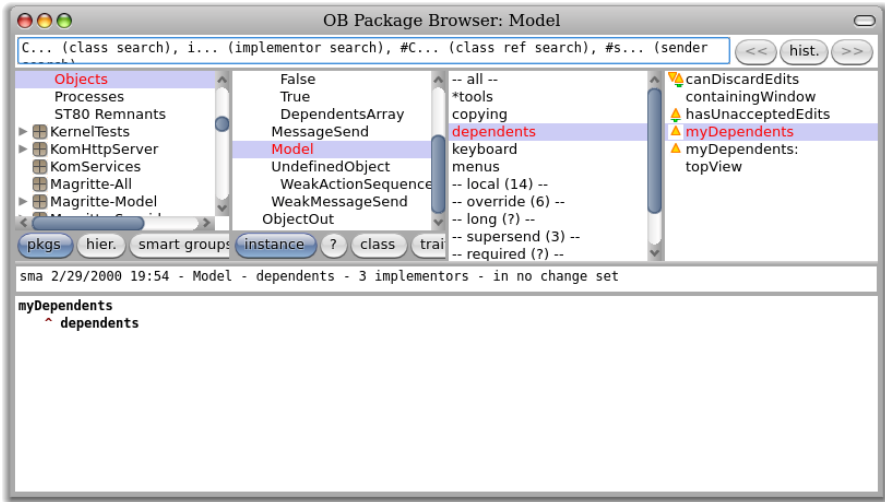


Figure 6.3: ModelクラスのmyDependentsメソッドをブラウザで表示

を作るにはプロトコルペインでアクションをクリックし **new protocol** を選びます。ダイアログの中に新しく作ったものの名前を入力すれば完了です。パッケージやプロトコルについては、名前や内容を入力する以外に必要な作業は何もありません。



Figure 6.4: クラス作成テンプレートをブラウザに表示

これとは対照的に、新しいクラスやメソッドを作るには、実際に何らかの

Smalltalk コードを書く必要があります。(左端のペインで) 現在選択されているパッケージをクリックすると、ブラウザ下部のペインにクラス生成のテンプレートが表示されます (図 6.4)。このテンプレートを編集することで、新しいクラスを作ります。Object を作りたいサブクラスの親となる既存のクラス名で置き換え、NameOfSubclass を新しいサブクラス名に置き換えます。既にわかっているのであれば、インスタンス変数名を入力します。新しいクラスに対するカテゴリは、デフォルトで現在選択しているパッケージのカテゴリとなりますが、<sup>3</sup>必要なら変更することもできます。既にサブクラス化の元となる既存クラスをブラウザで選択しているのならば、クラスペインをアクションクリックし `class templates ... ▸ subclass template` を選択することにより、ほんの少し初期化が異なる同様のテンプレートを得ることができます。また、既存のクラス定義でクラス名を新しいものに変えることでもクラス定義を編集できます。どの場合でも新しい定義をアクセプトすると、(#から始まる名前) 新しいクラスが (つまり対応するメタクラスのインスタンスとして) 作られます。クラスを作ることはそのクラスを参照するグローバル変数を作ることでもあるため、名前によって既存の全クラスを参照することができるわけです。

新しいクラス名が、(すなわち#で始まる) シンボルとしてクラスの作ったテンプレートの中に現れる理由や、クラスが作られた後は (すなわち#のない) 識別子としてクラス名を使えば、そのクラスをコードから参照できるようになる理由が理解できたでしょうか？


新しいメソッドを作るプロセスも同様です。最初にメソッドを作りたいクラスを選択し、次にプロトコルを選択します。図 6.5 のように、ブラウザは編集可能なメソッド作成テンプレートを表示します。

## コード空間を渡り歩く

ブラウザはコードを探したり分析したりするためのいくつかのツールを提供します。これらのツールは、様々なコンテキストメニューをアクションクリックすることで利用できます。また、特に最も頻繁に使われるツールなどにはキーボードショートカットが用意されています。

## 新しいブラウザウィンドウを開く

時折、複数のブラウザウィンドウを開きたくなる場合があります。コードを書いているときは少なくとも二つ必要でしょう。一つは入力中のメソッドのため、もう一つは物事がどう動いているのか調べるのにシステム周りを見るためです。テキスト中でクラス名を選択した後にキーボードショートカット `CMD-b` を使えば、そのクラスを表示するブラウザを起動することができます。

 次のことを試してみてください: ワークスペースウィンドウ内でクラス名

<sup>3</sup>パッケージとカテゴリが正確には同じものではないことを思い出してください。6.3 節にてそれらの関係について詳しく見ていきます

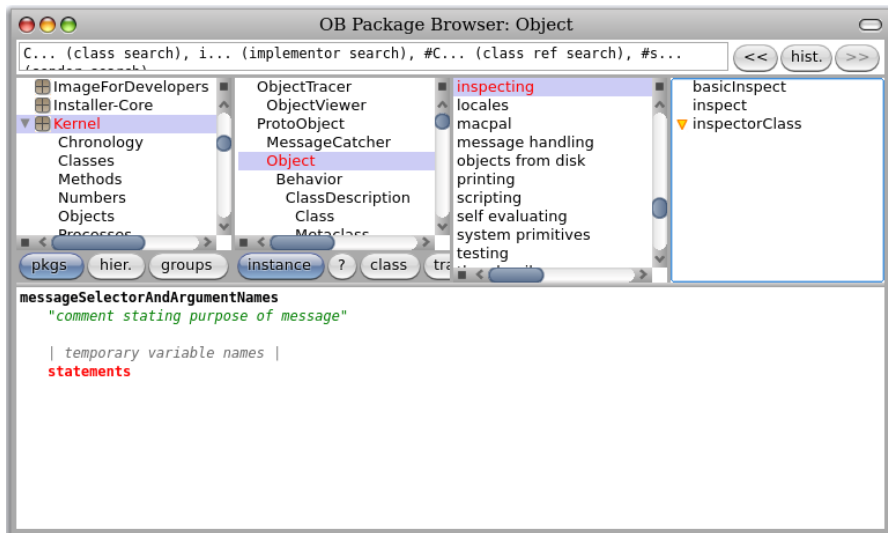


Figure 6.5: メソッド作成テンプレートをブラウザに表示

(例えば Morph) を入力し、それを選択し `CMD-b` を押してください。この方法はよく役に立ち、どのテキストウィンドウでも使えます。

### メッセージのセンダとインプリメンタ

メソッドペインで `browse ... senders (n)` をアクションクリックすると、選択したメソッドを使用しているすべてのメソッドのリストが表示されます。ブラウザを開いて Morph を選択した後、メソッドペインの中の `drawOn:` メソッドをクリックすると、`drawOn:` 本体がブラウザ下部に表示されます。 `senders (n)` (図 6.6) を選ぶと、`drawOn:` を一番上の項目としたメニューが現れ、以下 `drawOn:` が送るすべてのメッセージ (図 6.7) が順に表示されます。このメニューの項目を選ぶと、選択したメッセージ (図 6.8) を送る仮想イメージ内の全メソッドのリストがブラウザで表示されます。

`senders (n)` 中の “n” は、メッセージのセンダを探すためのキーボードショートカットが `CMD-n` であることを示しています。この方法は、どのテキストウィンドウでも使えます。

🔍 コードペインにあるテキスト “`drawOn:`” を選択して `CMD-n` を押せば、直ちに `drawOn:` のセンダが表示されます。

AtomMorph $\rightarrow$ drawOn:にある `drawOn:` のセンダを探せば、それが `super send` であることがわかるでしょう。つまり、実行されるメソッドは AtomMorph のスーパークラスにあることがわかります。ではいったいどのクラスでしょうか？

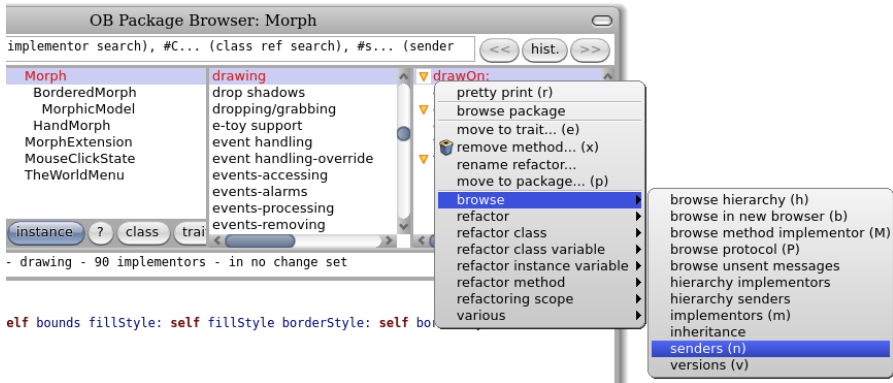


Figure 6.6: senders (n) メニュー項目。

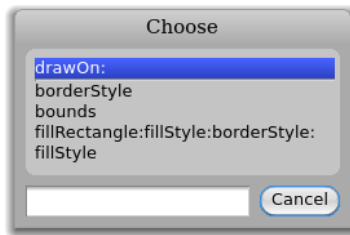


Figure 6.7: メッセージのセンドを選択。

`browse ▸ hierarchy implementors` をアクションクリックすれば、そのメソッドが `EllipseMorph` クラスにあることがわかります。

図 6.8 のリストの 6 番目にあるセンド `Canvas»draw` を見てみましょう。このメソッドは、引数として渡されたどんなオブジェクトに対しても `drawOn:` を送ることがわかります。つまり、あらゆるクラスのインスタンスが引数となる可能性があります。データフロー解析がいくつかのメッセージのレシーバクラスを把握する助けにはなりますが、一般的には、ブラウザを使ってどのメッセージ送信がどのメソッドの実行につながるのかわかる簡単な方法はありません。このような理由から“センド”ブラウザは、メッセージが示唆するメソッドだけ、つまり選択されたセクタを持つすべてのメッセージセンドを表示します。それにも関わらず、どのようにメソッドを使うのかわかる必要があるときには、すぐに使用例を導けるのでセンドブラウザが大いに役立ちます。同じセクタを持つすべてのメソッドは同じように使われるはずなので、メッセージの使い方はどれも似ているはずで

インプリメンタブラウザも似たような動作をしますが、メッセージセンドのリストの代わりに、同じセクタを持つメソッドを実装している全クラスのリストを表示します。このことを確かめるために、メソッドペインにある `drawOn:` を選び、`browse ▸ implementors (m)` を選びます (もしくはコードペイン中のテキ



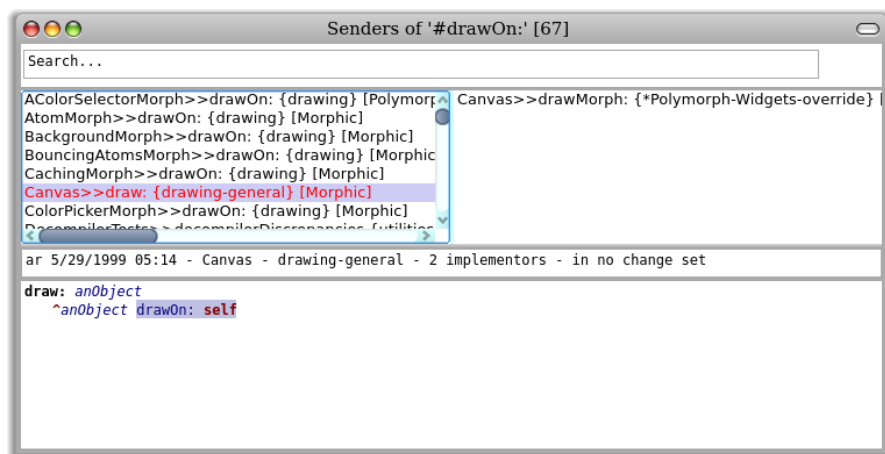


Figure 6.8: センダブラウザは Canvas>draw メソッドが、引数に対して drawOn: メッセージを送っていることを表しています。

スト “drawOn:” を選択し、CMD-m を押します)。すると、drawOn: メソッドを実装する 90 余りのクラスが、スクロール可能なメソッドリストのウィンドウとして表示されます。これほど多くのクラスがこのメソッドを実装していることに驚く必要はありません。drawOn: は、画面上に自分自身を描画することができるすべてのオブジェクトによって理解されるメッセージなのです。

## メソッドのバージョン

新しいバージョンのメソッドを保存しても、古いものは失われません。Pharo はすべての古いバージョンを保存し、異なるバージョンを比較したり、古いバージョンに戻したり (“復帰”) することができます。browse ▸ versions (v) メニュー項目により、選択したメソッドに加えられた一連の変更を調べることができます。図 6.9 では、buildWorldMenu: メソッドの二つのバージョンを示しています。

上部のペインには、そのメソッドのバージョンごとに 1 行ずつ表示されており、このメソッドを書いたプログラマのイニシャル、保存した日時、クラスとメソッドの名前、そして定義されているプロトコルが表示されます。現在の (アクティブな) バージョンはリストの一番上にあります。選択されたバージョンは、下部のペインに表示されます。選択されたメソッドと現在のバージョンの違いを表示したり、選択したバージョンに復帰したりするためのボタンが用意されています。

バージョンブラウザの存在は、必要なコードを残すべきか悩む必要がなく、単に削除すれば良いことを意味しています。もしそのコードが本当に必要だとわかったら、いつでも古いバージョンに復帰することができますし、古いバー

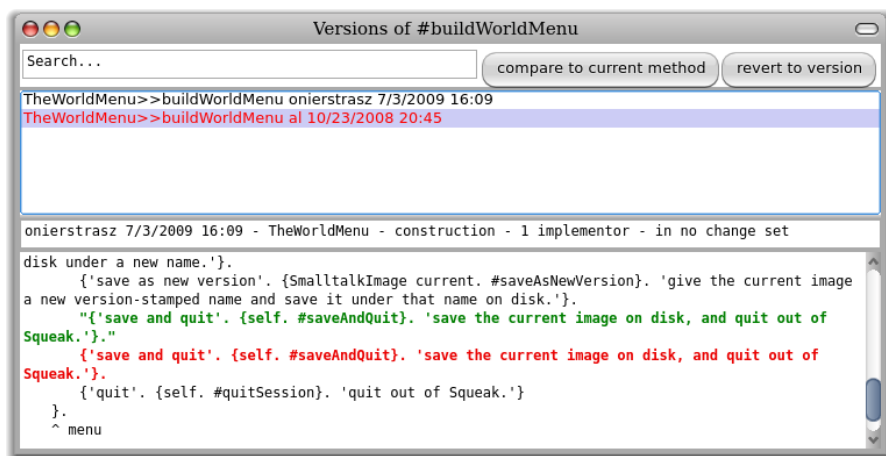


Figure 6.9: バージョンブラウザが二つの TheWorldMenu>buildWorldMenu:メソッドを表示

ジョンから必要なコードの断片を得て別のメソッドに貼り付けることもできます。バージョンブラウザを使う癖を付けましょう。もはや必要のなくなったコードをコメントアウトするのは悪い習慣です。なぜなら、そのことで現在のコードが読みにくくなるからです。Smalltalker は読みやすいコードを非常に高く評価します。

**HINT** メソッド全体を削除した後で、元に戻したい場合はどうすればいいでしょう？ チェンジセットの中に削除したものをを見つけることができます。そこでアクションクリックすることにより、以前のバージョンを確認できます。チェンジセットブラウザは 6.8 節で述べられています。

## メソッドのオーバーライド

インヘリタンスブラウザは、表示されたメソッドによってオーバーライドされたすべてのメソッドを表示します。どのように動作するかを見るために、ブラウザで ImageMorph>drawOn: メソッドを選択してください。メソッド名の隣にある三角形のアイコンに注目してください。(図 6.10) 上向きの三角形は、ImageMorph>drawOn:が継承されたメソッド(すなわち Morph>drawOn:)をオーバーライドしていることを示しており、さらに下向きの三角形は、サブクラスによってオーバーライドされていることを示しています。(アイコンをクリックすることで、これらのメソッドに移動することもできます。)ここで、**browse ▾ inheritance**を選択してください。インヘリタンスブラウザが、オーバーライドされたメソッドの階層関係を示します。(図 6.10 参照)

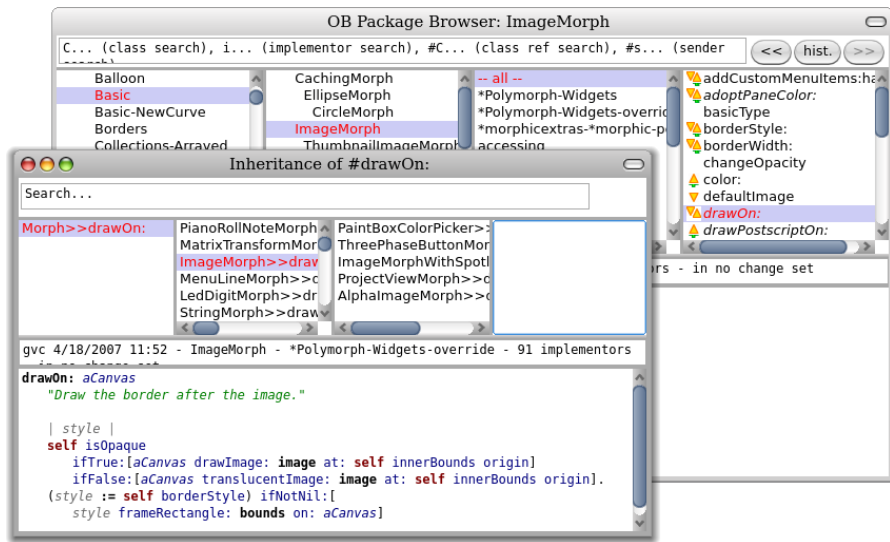


Figure 6.10: ImageMorph»drawOn: とそれがオーバーライドしたメソッド。選択したメソッドの兄弟関係がリストに表示されている。

## 階層ビュー

標準では、ブラウザは左端のペインにパッケージのリストを表示します。しかしクラスの階層ビューに切り替えることは可能です。関心のある特定のクラスを選択してください。例えば ImageMorph を選択し、**hier.** ボタンをクリックします。選択したクラスのすべてのスーパークラスとサブクラスを含むクラス階層を、左端のペインで見ることができます。2 番目のペインは選択したクラスのメソッドを実装するパッケージのリストが表示されます。<sup>4</sup>図 6.11 では、ImageMorph のスーパークラスが Morph であることを階層ブラウザが示しています。

## 変数への参照を見つける

クラスペインにあるクラスをアクションクリックし、**browse ▸ chase variables** を選択することで、インスタンス変数やクラス変数が使われている場所を見つけることができます。追跡ブラウザを使うことで、すべてのインスタンス変数やクラス変数のアクセサ、さらにこれらのアクセサを送るメソッドなどを探索することができます (図 6.12)。

<sup>4</sup> 訳注：バージョン 1.4 では階層ブラウザが表示されますが、2 番目のペインにパッケージのリストは表示されません

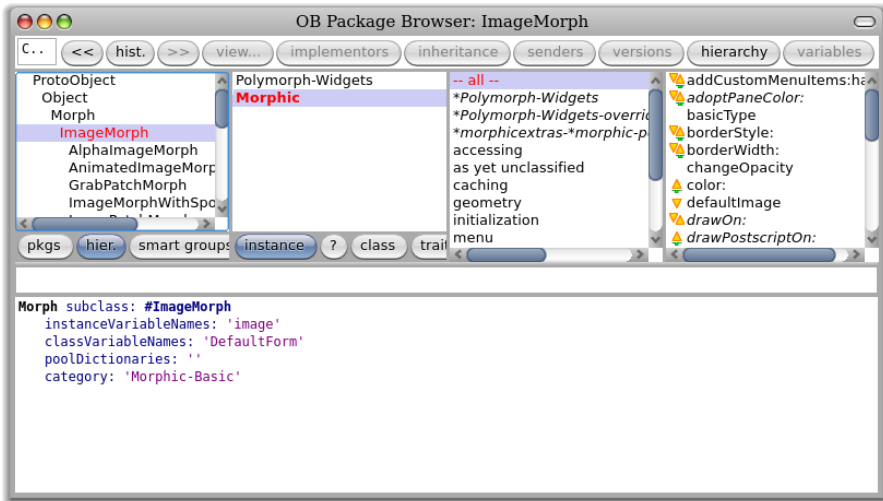


Figure 6.11: ImageMorphの階層ビュー

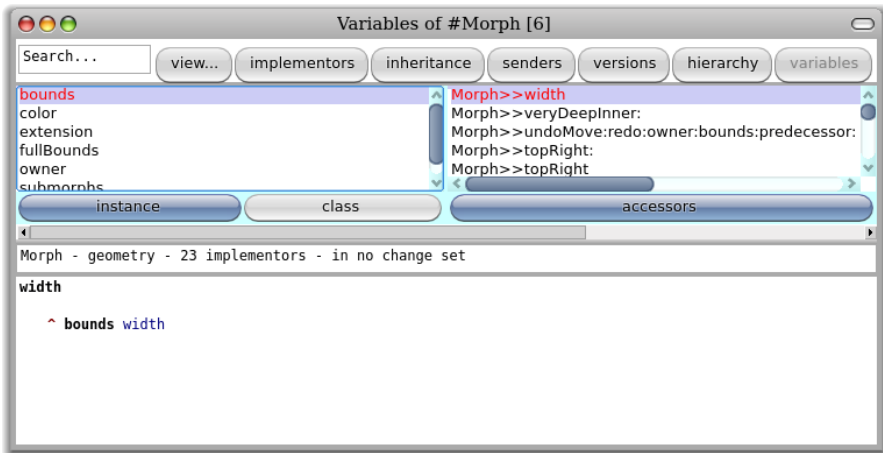


Figure 6.12: Morphに対する追跡ブラウザー

## ソース

メソッドペインをアクションクリック することで利用可能となる `various ▸ view ...` メニュー項目は、“how to show” メニューを表示します

が<sup>5</sup>、これは選択されたメソッドをソースペインでブラウザがどのように表示するかを選ばせるものです。 `source` (ソースコード)、 `prettyPrint` (整形されたソースコード)、 `byteCode` (バイトコード) および `decompile` (バイトコードから逆コンパイルされたソースコード) といった選択肢があります。

“how to show” メニューで選択した `prettyPrint` は、ソースを保存する前に整形出力したメソッドと同じもの<sup>67</sup>ではないことに注意してください。このメニューは、ブラウザの表示方法を制御するだけで、システムに格納されたコードには何の影響も及ぼしません。二つのブラウザを開いて一方のブラウザで `prettyPrint` を選び、もう一方の `Browser` で `source` を選ぶことで、このことを確認できます。同じようにして、二つのブラウザで同じメソッドを表示させた状態で、一方で `byteCode` を選択し、もう一方で `decompile` 選択することは、Pharo の仮想マシンのバイトコードの命令セットについて学ぶ良い方法です。

## リファクタリング

コンテキストメニューから多くの標準的なリファクタリングが行えます。四つのペインのどれでもアクションクリックすれば、現在リファクタリング可能な操作を見ることができます。図 6.13 を参照してください。

以前は、リファクタリングのためにリファクタリングブラウザと呼ばれる特殊なブラウザを使う必要がありましたが、現在はどのブラウザからも利用できます。

## ブラウザのメニュー

ブラウザのペインをアクションクリックすることで多くの追加機能を利用することができます。たとえメニュー項目上のラベルが同じであっても、その意味は文脈に依存します。例えばパッケージペイン、クラスペイン、プロトコルペイン、そしてメソッドペインのすべてが `File out` メニュー項目を持っています。しかしそれらは異なる動作をします。パッケージペインの `File out` メニューは、パッケージ全体をファイル出力します。クラスペインの `File out` メニューはクラス全体をファイル出力します。プロトコルペインの `File out` メニューはプロトコル全体をファイル出力します。そしてメソッドペインの `File out` メニューは表示されたメソッドだけをファイル出力します。このことは明らかだと思えるかもしれませんが、初心者にとっては混乱の元になります。

おそらくパッケージペインの中で最も便利なメニュー項目は `find class... (f)` です。カテゴリは活発に開発しているコードについては役立ちますが、ほとんど

<sup>5</sup>訳注：バージョン 1.4 ではメソッドペインの下にある `View` ボタンを押すと “Choose” メニューが表示されます

<sup>6</sup>`pretty print (r)` はメソッドペインの最初のメニュー項目もしくはコードペインの真ん中下側にあります。

<sup>7</sup>訳注：バージョン 1.4 では、メソッドペインおよびコードペインでの整形出力機能はありません。

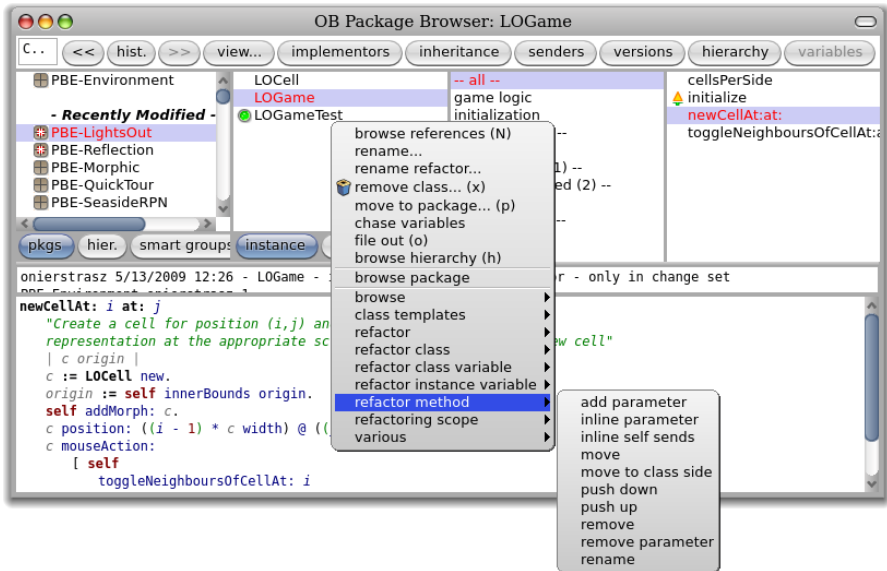


Figure 6.13: リファクタリング操作

誰もシステム全体のカテゴリ化について知りませんし、クラスがどのパッケージにあるか推測するよりも、クラス名の先頭の数字を入力した後で `shortf` と入力の方がずっと早いです。 `recent classes...` はたとえクラス名を忘れてしまっても、最近ブラウズしたクラスへ素早く戻するのに役立ちます。

ブラウザの左上にある検索ボックスに名前を入力することで、特定のクラスやメソッドを探すこともできます。リターンキーを入力するとシステムに問い合わせがなされ、その結果が表示されます。検索語の前に `#` を付けることで、クラスへの参照や、メッセージのセンダを探せることに気をつけてください。選択したクラスの特定のメソッドを探すのであれば、(デフォルトで表示される) `-all-` プロトコルをブラウズした方が速いでしょう。メソッドペインにマウスを移動し、探そうとしているメソッド名の最初の文字を入力してください。メソッドペインがスクロールして、探しているメソッド名が見えるでしょう。


🔍 `OrderedCollection»removeAt:` で二つの操作方法を試してください。

メニューには他の多くのオプションがあります。少々時間を取って、ブラウザ上のどこにどのような機能があるかを探しておく、後ほど報われることでしょう。

🔍 クラスペインメニューの `Browse Protocol`、`Browse Hierarchy`、そして `Show Hierarchy` の結果を比較してください。

## プログラムによるブラウズ

`SystemNavigation` クラスはシステム周りを操作するために役に立つユーティリティメソッドのいくつかを提供しています。典型的なブラウザが提供する機能の大半は、`SystemNavigation`によって実装されています。

 `drawOn::`のセンダを見るためには、ワークスペースを開いて以下のコードを評価してください。

```
SystemNavigation default browseAllCallsOn: #drawOn:
```

センダの検索を特定のクラスのメソッドに制限するには以下のようにします。

```
SystemNavigation default browseAllCallsOn: #drawOn: from: ImageMorph
```

開発ツールもオブジェクトなので、プログラムから完全にアクセス可能であり、自分自身のツールを開発することや既存のツールを必要に応じて変更することもできます。

`implementors` メニュー項目と等価なプログラムは以下のようなものです。

```
SystemNavigation default browseAllImplementorsOf: #drawOn:
```

他に何ができるのかを知るには、ブラウザで `SystemNavigation` クラスを探してください。その他の操作例については、FAQ(付録 A)で見つけることができます。

## 6.3 Monticello

2.9 節において、Pharo のパッケージ作成ツールである Monticello の概要について説明しました。しかし、Monticello は、その章で述べたこと以上の特徴を備えています。Monticello はパッケージを扱うので、Monticello について説明する前に、まずパッケージが何なのかということを正確に説明する必要があります。

### パッケージ: Pharo コードの宣言的なカテゴリ化

かなり前の 2.3 節において、パッケージとは概ねカテゴリと同等のものであると指摘しました。ここでは、その関係が実際には何であるか見ていきます。パッケージシステムとは、Smalltalk ソースコードを組織化するためのシンプルで軽量な方法のことです。これは、カテゴリとプロトコルに対して簡単な命名規則を利用します。

例を使って説明しましょう。あるフレームワークを開発していると仮定しましょう。これは Pharo から関係データベースを簡単に利用するためのもの

です。あなたはこのフレームワークを `PharoLink` と呼ぶことにしました。そして、作成したクラスすべてを含むような一連のカテゴリを作ったとします。例えば、`'PharoLink-Connections'` カテゴリは `OracleConnection MySQLConnection PostgresConnection` を含み、`'PharoLink-Model'` カテゴリは `DBTable DBRow DBQuery` 等を含みます。しかし、作成したコードのすべてがこれらのクラスの中にあるわけではありません。例えば、オブジェクトを SQL 的な形式に変換するための一連のメソッドもあるかもしれません。

```
Object»asSQL
String»asSQL
Date»asSQL
```

これらのメソッドは、`PharoLink-Connections` や `PharoLink-Model` カテゴリと同じパッケージに属しています。しかし、明らかに `Object` クラス全体がそのパッケージに属するわけではありません。つまり、クラスの残りが別のパッケージにあるとしても、特定のメソッドをパッケージに含める方法が必要になります。

それを行う方法は、(`Object`, `String` `Date` 等) のメソッドを、`prot*PharoLink` (最初のアスタリスクに注意してください) というプロトコルに置くことです。`PharoLink-...` カテゴリと `*PharoLink` プロトコルの組み合わせが、`PharoLink` パッケージを形作ります。正確には、パッケージに置かれる規則は以下の通りです。

`Foo` と名付けられたパッケージは次のものを含みます。

1. `Foo` カテゴリもしくは `Foo-` で始まる名前のカテゴリ内にある、クラスのすべてのクラス定義
2. `*Foo` や `*foo`<sup>8</sup> という名前のプロトコルか、`*Foo-` や `*foo-` で始まる名前のプロトコルにある、任意のクラス内のすべてのメソッド
3. `Foo` カテゴリか、`Foo-` で始まる名前のカテゴリのクラスにあるすべてのメソッド。ただし `*` で始まる名前のプロトコルにあるメソッドは除外される。

これらの規則の結果、各クラスの定義と各メソッドが、ちょうど一つのパッケージに属することになります。最後の規則にある例外は、それらのメソッドが他のパッケージに属するために必要となります。規則 2 にある、大文字小文字が無視されるというのは、カテゴリ名がキャメルケース (空白を含まない) のに対して、(必ずというわけではないが) 典型的なプロトコル名が小文字である (空白を含む場合がある) からです。

`PackageInfo` クラスがこれらの規則を実装しており、このクラスを試すことで規則の感触を掴むことができます。

 ワークスペース上で以下の式を評価してください。

```
mc := PackageInfo named: 'Monticello'
```

<sup>8</sup>比較の際に大文字か小文字かは無視されます。



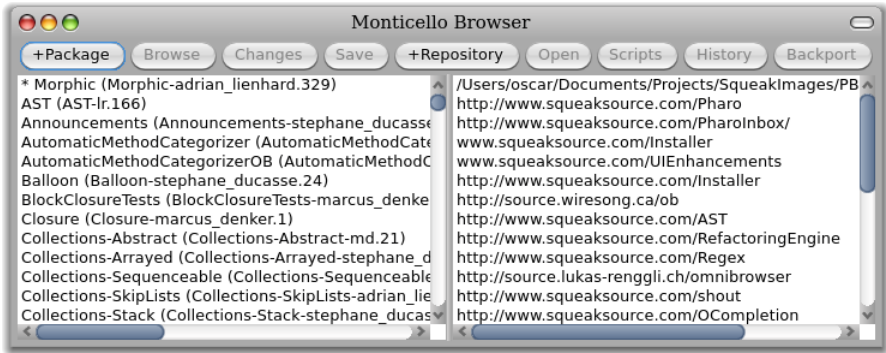



Figure 6.14: Monticello ブラウザ

さて、このパッケージについて探索できるようになりました。例えば、ワークスペース上で `mc classes` を表示することで、Monticello パッケージを構成するクラスの長いリストが得られます。 `mc coreMethods` は、これらのクラスにあるすべてのメソッドに対する `MethodReference` のリストが返されます。 `mc extensionMethods` は、おそらく最も面白い問い合わせでしょう。この問い合わせは、Monticello パッケージに含まれるすべてのメソッドのうち、Monticello クラスを除いたものを返します。

パッケージは比較的新しく Pharo に加えられたものです。しかし、パッケージの命名規則は、既に使われていたものをベースとしていたため、パッケージの採用を意識していない古いコードを分析する際にも `PackageInfo` を使うことができます。

 (PackageInfo named: 'Collections') `externalSubclasses` を表示してください。この式は、Collection パッケージに含まれていない Collections のすべてのサブクラスの一覧を返します。

## Monticello の基礎

Monticello は、合衆国 3 番目の大統領であり「バージニア信教自由法」の起草者であるトーマス・ジェファーソンの山頂の家にちなんで名付けられました。この名前はイタリア語で“小さな山”を意味し、イタリア語の発音に従い“c”を chair の“ch”と発音して「モンティチェロ」といいます。

Monticello ブラウザを開くと、図 6.14 のように、二つのペインと 1 列に並んだボタンが表示されます。左側のペインには、実行中の仮想イメージにロードされたすべてのパッケージが表示され、名前の後の括弧の中にはパッケージを特定するバージョンが表示されます。

右側のペインには、Monticello が知っているすべてのソースコードリポジ

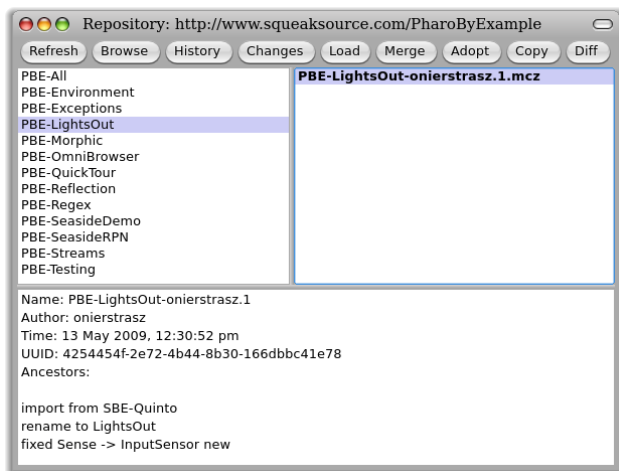



Figure 6.15: リポジトリブラウザ

トリが表示されており、通常はそれらのリポジトリからコードがロードされます。左側のペインにあるパッケージを選ぶと、右側のペインには、選択したパッケージのバージョンを含みリポジトリのみを表示するようにフィルタリングされます。

リポジトリの一つは *package-cache* と呼ばれるディレクトリで、実行中のイメージファイルのあるディレクトリのサブディレクトリです。遠隔地にあるリポジトリからコードを読み込んだり、書き込んだりする場合、コピーが *package-cache* にも保存されます。ネットワークが利用できない場合にパッケージへアクセスする必要があるとき、*package-cache* が役立ちます。また、例えば電子メールの添付ファイルとして Monticello ファイル (.mcz) が直接与えられた場合、そのファイルを *package-cache* ディレクトリに置くのが一番手軽なアクセス方法となります。

リストに新しいリポジトリを追加するには、**+Repository** ボタンをクリックし、ポップアップメニューからリポジトリの種類を選んでください。では HTTP リポジトリを追加しましょう。

 *Monticello* を開き、**+Repository** ボタンをクリックし、**HTTP** を選びます。以下のようにダイアログ内容を編集してください。

```
MCHttpRepository
location: 'http://squeaksource.com/PharoByExample'
user: ''
password: ''
```

その後、このリポジトリをリポジトリブラウザで開くために、**Open** ボタンを

クリックします。図 6.15 のようなものが見えるでしょう。左側のペインには、リポジトリ内のすべてのパッケージのリストがあります。その中の一つを選択すれば、右側のペインにこのリポジトリ内にある選択されたパッケージのすべてのバージョンが表示されます。

その中の一つのバージョンを選択すると、**Browse** ボタンでの (イメージファイルに読み込むことなしに) ブラウズや **Load** ボタンでの読み込み、そして、**Changes** ボタンでは、選択したバージョンを読み込んだ場合の変更点を見ることができません。**Copy** ボタンでは、任意のバージョンのパッケージのコピーを作成して、他のリポジトリに書き込むこともできます。

見ればわかる通り、バージョン名には、パッケージ名、バージョン作成者のイニシャル、およびバージョン番号が含まれています。バージョン名はリポジトリ内のファイル名でもあります。これらの名前を変更してはいけません。なぜなら Monticello の適切な操作は、これらの名前に依存しているからです! Monticello のバージョンファイルは単なる zip アーカイブであり、興味があれば zip ツールを使ってファイルを解凍できますが、その内容を見る最良の方法は、Monticello 自体を使うことです。

Monticello でパッケージを作成するには、二つのことを行う必要があります。一つはコードを書くこと、もう一つはそれを Monticello に伝えることです。

🕒 **PBE-Monticello** というパッケージを作成し、図 6.16 のように二つのクラスをその中に置いてください。もちろん、**Object** などの既存のクラスのメソッドを作り、120 ページにある規則を使ってこのクラスと同じパッケージに置いてください—図 6.17 を参照のこと。

作成したパッケージを Monticello に伝えるために、**+Package** ボタンをクリックし、パッケージ名、この場合は“PBE”を入力してください。Monticello は PBE をパッケージリストに加えます。パッケージはアスタリスクでマークされ、仮想イメージ内のバージョンがどのリポジトリにも書き込まれていないことを示します。ここで、Monticello に二つのパッケージを持つことに注意してください。一つは PBE、もう一つは PBE-Monticello です。PBE は PBE-Monticello や PBE- で始まる他のすべてのパッケージを含むため、これでよいのです。

最初はこのパッケージに関連している唯一のリポジトリは、図 6.18 に示されているようにパッケージキャッシュだけです。それで問題なく、コードを保存することができますし、パッケージキャッシュに格納されます。**Save** ボタンをクリックすると、図 6.19 のように保存するパッケージのバージョンに対するログメッセージを入力するよう求められます。このメッセージをアクセプトすると Monticello はパッケージを保存します。保存したことを示すため、Monticello のパッケージペインに付けられていたアスタリスクがなくなり、バージョン番号が付与されます。

もしパッケージに変更を加えた—例えばクラスの一つにメソッドを追加した—場合、アスタリスクが再び現れて、変更が保存されていないことを示します。パッケージキャッシュをリポジトリブラウザで開いて保存したバージョンを選べば、**Changes** ボタンや他のボタンを使うことができます。もちろんリポ



Figure 6.16: “PBE” パッケージの二つのクラス。

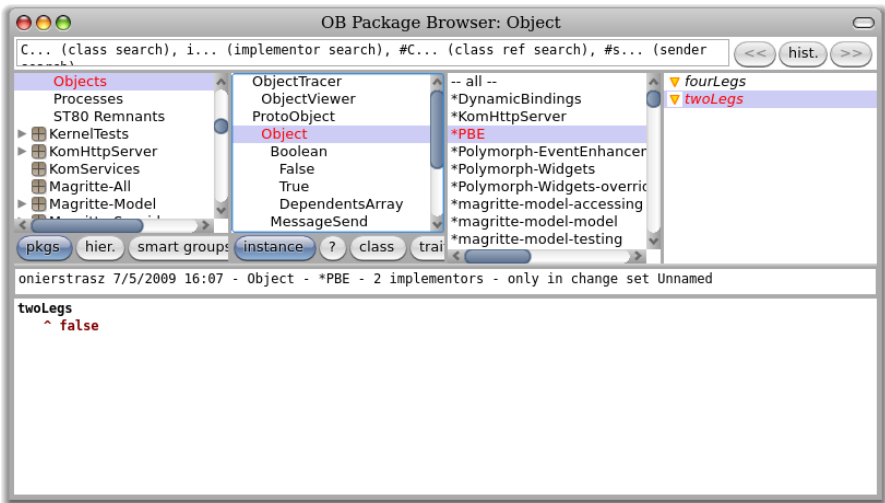


Figure 6.17: “PBE” パッケージに置かれる拡張メソッド。

ジトリに新しいバージョンを保存することもできます。Refresh ボタンを押してリポジトリビューを一度更新すれば、図 6.20 のように表示されるはずです。

新しいパッケージをパッケージキャッシュ以外のリポジトリに保存するためには、必要に応じてリポジトリを追加するなどして、まず Monticello にその

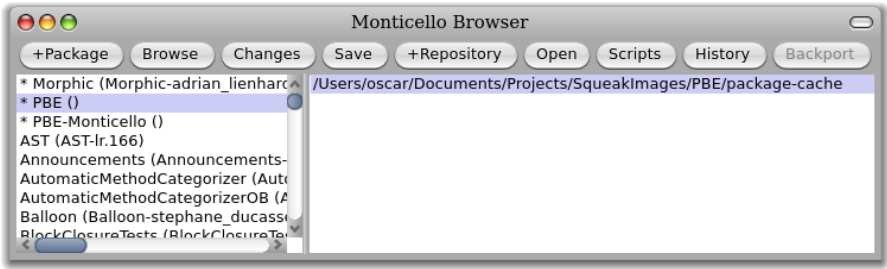


Figure 6.18: Monticello でまだ保存されていない PBE パッケージ。

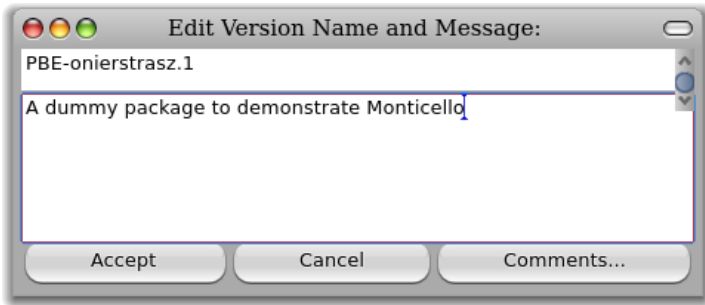


Figure 6.19: パッケージの新バージョンに対するログメッセージを入力する。

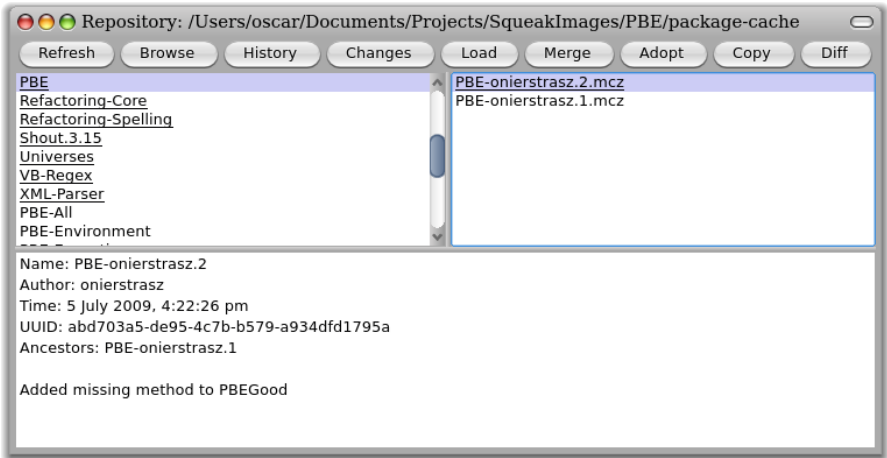


Figure 6.20: 作成したパッケージの二つのバージョンがパッケージキャッシュにあります。

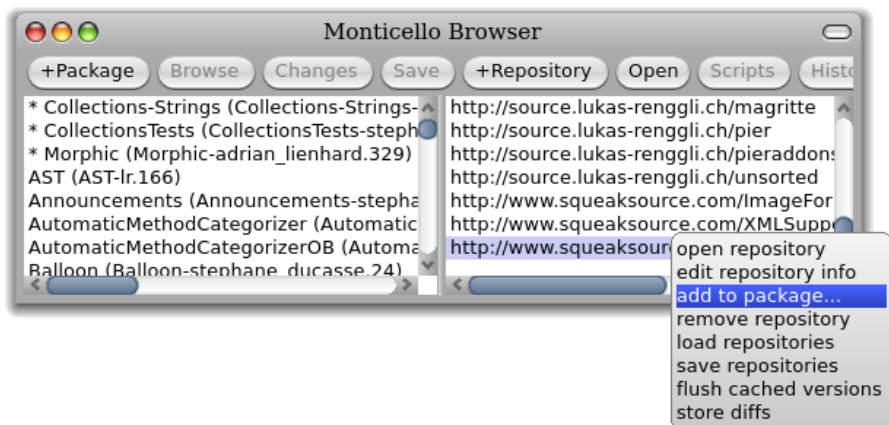


Figure 6.21: パッケージに関連づけられたポジトリのセットにリポジトリを追加する。

リポジトリについて知らせる必要があります。その後、パッケージキャッシュ上のリポジトリブラウザで **Copy** ボタンを使って、コピー先のリポジトリを選ぶことができます。図 6.21 のようにリポジトリ上でアクションクリックし、**add to package ...** を選ぶことで、お望みのリポジトリをパッケージと関連づけることもできます。いったんパッケージがリポジトリに関連づけば、Monticello ブラウザでリポジトリとパッケージを選んで **Save** ボタンをクリックすることで、新しいバージョンを保存することができます。もちろんリポジトリへ書き込むためのパーミッションは必要です。SqueakSource 上の PharoByExample リポジトリは世界から読むことができますが、書き込むことはできません。そこに保存しようとしてもエラーメッセージが表示されるでしょう。しかし、SqueakSource にあなた専用のリポジトリを作成することは可能です。http://www.squeaksource.com にある Web インターフェースを使って、あなたのパッケージを保存するリポジトリを利用できます。これは、友人とコードを共有したり複数のコンピュータを使ったりする場合などの仕組みとして特に有用です。

書き込みパーミッションを持たないリポジトリに対して保存しようとした場合、ともかくそのバージョンはパッケージキャッシュに書き込まれます。つまり、リポジトリの情報を編集するか (Monticello ブラウザでアクションクリック) 別のリポジトリを選んで、パッケージキャッシュブラウザから **Copy** ボタンを押せば、そのバージョンを復活させることができます。

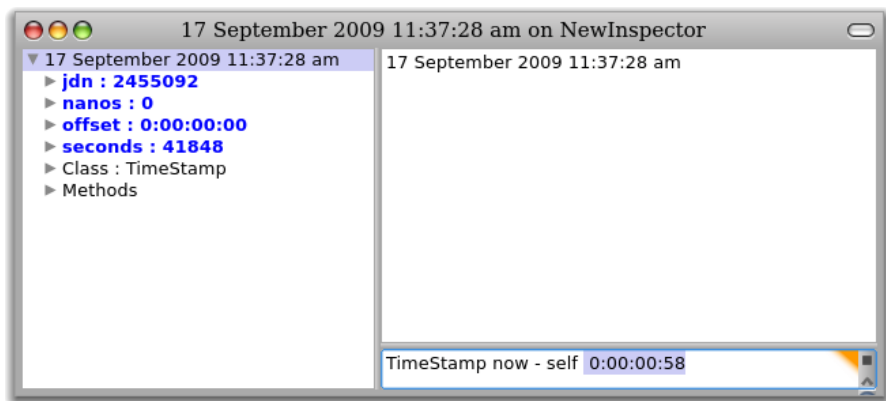


Figure 6.22: TimeStamp now をインスペクトする

## 6.4 インスペクタと [Explore]

Smalltalk が他の多くの言語と異なる特徴の一つは、静的なコードの世界ではなく、生きたオブジェクトの世界へ開かれた窓を提供しているということです。そこではどのようなオブジェクトでもプログラマによって調べることが可能です。さらには変更することさえできます。— ただし、システムを構成している基本的なオブジェクトを変更するようなときには注意が必要です。もちろん言うまでもなく、最初にイメージを保存しておいてください！

### インスペクタ

🔍 インスペクタを使ってどんなことができるか、実際に見てみるためにワークスペースで `TimeStamp now` と入力し、アクションクリックで `inspect it` を選んでください。

(メニューを使う前にテキストを選択しておく必要はありません。テキストが選択されていない場合には、メニュー操作は現在の行すべてを対象にするからです。 `inspect it` を選ぶ代わりに `CMD-i` とタイプしても同じです)

図 6.22 のようなウィンドウが現れます。これがインスペクタで、特定のオブジェクトの内部に向かって開かれた窓と考えることができます。特定のオブジェクトとはこの場合、 `TimeStamp now` という式を評価することによって生成された `TimeStamp` のインスタンスオブジェクトのことです。

ウィンドウのタイトルバーには、インスペクトしているオブジェクトが印字形式で表示されています。左側のペインの先頭の `self` を選択すると、右側のペインにそのオブジェクトが文字列で表示されます。左側のペインはそのオ

プロジェクトのツリー構造ビューとなっており、`self` をツリーの根としています。インスタンス変数は、名前の横にある三角形を展開することで探求できます。<sup>9</sup>

インスペクタの下部にある横長のペインは小さなワークスペースウィンドウです。このウィンドウでは、擬似変数の `self` が左側のペインで選択したオブジェクトに束縛されている<sup>10</sup>ので便利です。そこで、

```
self - TimeStamp today
```


上記の式をワークスペースペインで `inspect it` すると、結果は `Duration` オブジェクトになります。このオブジェクトは、今日の午前0時と `TimeStamp now` を評価して得られた `TimeStamp` オブジェクト（インスペクトしているもの）との、時間間隔を表しています。さらに `TimeStamp now - self` を評価すれば、あなたがこのセクションを読むのに費やした時間を教えてくれるでしょう！

`self` だけでなく、オブジェクトのすべてのインスタンス変数も、ワークスペースペインの有効範囲にあります。つまり、変数に対して式で参照したり代入したりすることもできます。例えば、左のペインのツリーの根にあるオブジェクトを選んでから、ワークスペースペインで `jdn := jdn - 1` を評価すると、インスタンス変数 `jdn` の値が実際に変化することがわかりますし、`TimeStamp now - self` の値も1日分増えるでしょう。

`Dictionary` や `OrderedCollection` や `CompiledMethod`、さらにいくつかのクラスに対しては、インスペクタに特別な拡張がなされていて、これらのオブジェクトの内容をより簡単に検査することができます。

## オブジェクトエクスプローラ

オブジェクト エクスプローラは概念的にはインスペクタとよく似ていますが、情報の表し方が異なっています。この相違点を調べるために、先ほどまでインスペクトしていたものと同じオブジェクトをエクスプローラで探求してみましょう。

 インスペクタの左側のペインで `self` を選び、アクションクリック で `explore (I)` を選んでください

エクスプローラ ウィンドウは図 6.23 のように表示されます。ツリー構造の根の隣にある小さな三角形をクリックすると、ビューが図 6.24 のように変化します。このビューは、エクスプロアの対象とするオブジェクトのインスタンス変数を表示します。`offset` の隣の三角形をクリックすると、そのインスタンス変数が見えます。エクスプローラは、複雑な階層構造を探求しなければならない場合に便利なため、この名前が付けられています。

オブジェクトエクスプローラのワークスペースペインの働きは、インスペクタのもの若干異なります。`self` はツリー構造のルートとなっているオブジェ

<sup>9</sup> 訳注：Pharo 1.2 以降ツリーではなく一覧表示に変更されています

<sup>10</sup> 訳注：Pharo 1.2 以降で `self` は常にインスペクトしているオブジェクトを束縛しています



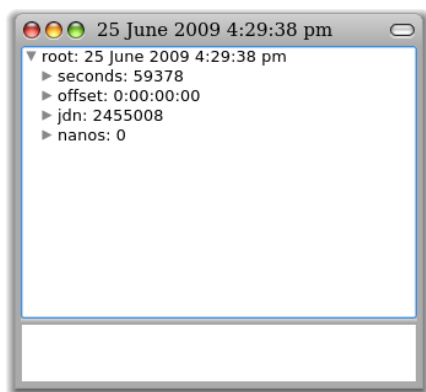


Figure 6.23: TimeStamp now を探求する

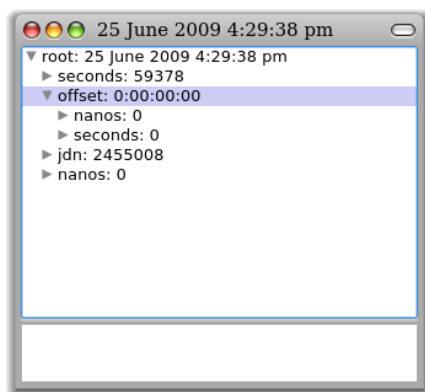



Figure 6.24: インスタンス変数を探求する

クトではなく、現在選択されているオブジェクトに束縛されるようになっています。また、選択されたオブジェクトのインスタンス変数も同様のスコープに従います。


エクスペローラの真価を見るために、深い入れ子構造を持ったオブジェクトを調べてみましょう。

 ワークスペースで Object explore を評価してください

これは Pharo の Object クラスを表すオブジェクトです。メソッド辞書を表示オブジェクトに加えて、そのクラスのコンパイル済みメソッドも直接ナビゲートできることに注意してください（図 6.25 参照）。

## 6.5 デバッガ

デバッガは間違いなく Pharo のツールの中で一番強力なツールです。ただデバッグのためというだけでなく、新しいコードを書くのにも利用されます。デバッガのデモンストレーションをするために、バグを作るところから始めましょう！

 ブラウザを使って、以下のメソッドを String クラスに追加してください

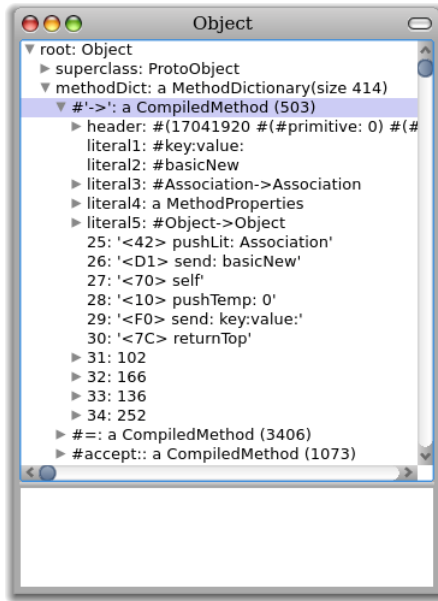


Figure 6.25: Object クラスを探求する

## Method 6.1: バグがあるメソッド

```
suffix
```

”自分自身をファイル名と仮定して、最後のドットから後ろの部分をサフィックスとして返す”

```
| dot dotPosition |
```

```
dot := FileDirectory dot.
```

```
dotPosition := (self size to: 1 by: -1) detect: [:i | (self at: i) = dot].
```

```
↑ self copyFrom: dotPosition to: self size
```

もちろん、こんな簡単なメソッドは動くのが当たり前だろうと確信するので、SUnit のテストも書かずにワークスペースで 'readme.txt' suffix と入力し、print it(p) を実行します。すると驚くことに、期待した答え 'txt' の代わりに、図 6.26 のような PreDebugWindow がポップアップします。

PreDebugWindow はどんなエラーが起きたのかタイトルバーでわかるようになっていました。さらに、エラーに至るまでのメッセージのスタックトレースが表示されます。トレースの一番下を起点とし、UndefinedObject>>Dolt という行が、ワークスペースで 'readme.txt' suffix を選び、Pharo に print it させたときのコンパイルおよび実行されたコードを表しています。もちろん、このコードは ByteString クラスのオブジェクト ('readme.txt') にメッセージ suffix を送ります。これは String クラスから継承している suffix メソッドの実行を引き起こし、このことがスタックトレースの次の行で ByteString(String)>>suffix として表されて

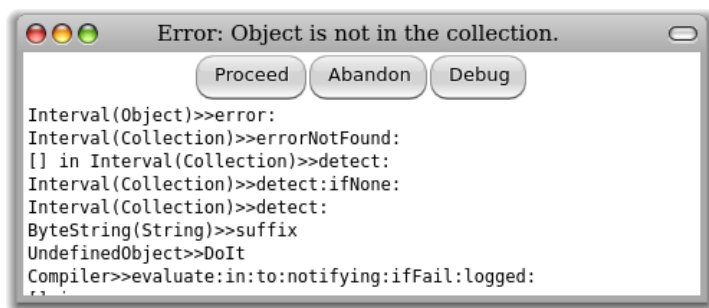


Figure 6.26: PreDebugWindow によるバグの通知

います。スタックをさらにたどると、`suffix` メソッドは `detect:` メッセージを送信し、...そして `detect:ifNone:` メソッドが `errorNotFound` を送信していることがわかります。

なぜドットが見つからなかったのかを調べるために、デバッガ本体が必要なので、`Debug` ボタンをクリックします。

図 6.27 にデバッガを示します。最初は威嚇的に見えますが、使うのはとても簡単です。タイトルバーと一番上のペインは `PreDebugWindow` にあるものとてもよく似ています。しかし、デバッガではスタックトレースはメソッドブラウザと合体させており、スタックトレース上の行を選択すると対応するメソッドが下のペインに表示されます。エラーを引き起こした実行はまだイメージ内にありますが、中断状態にあることを意識することは重要です。スタックトレースの各行は、実行を継続するのに必要なすべての情報を持った実行スタック上のフレームを表しています。これには、計算に伴うすべてのオブジェクトと各インスタンス変数、実行中のメソッドのすべての一時変数も含まれています。

図 6.27 では、一番上のペインで `detect:ifNone:` メソッドが選択されています。メソッドの本体は中央のペインに表示されています。`value` メッセージが青くハイライトされているのは、このメソッドではメッセージ `value` を送信し、その結果を待っている状態であることを示しています。

デバッガの一番下の四つのペインは、実際には (ワークスペースペインを持たない) 二つの小さなインスペクタです。左側のインスペクタは現在のオブジェクト、つまり、中央のペインで `self` に相当するオブジェクトを表しています。別のスタックフレームを選ぶと、`self` の示すオブジェクトが変わり、`self`-インスペクタの内容も変わります。左下のペインの `self` をクリックすると、`self` が `Interval` オブジェクト (10 to: 1 by -1) であることがわかります。これは期待通りのオブジェクトです。デバッガの小さなインスペクタにワークスペースペインは必要ありません。というのも、すべての変数がメソッドペインの有効範囲内にあるからです。このペインに対する式の入力や選択、それらの式の評価など自由に行うことができます。そうした変更は、いつでも `cancel()` メニュー

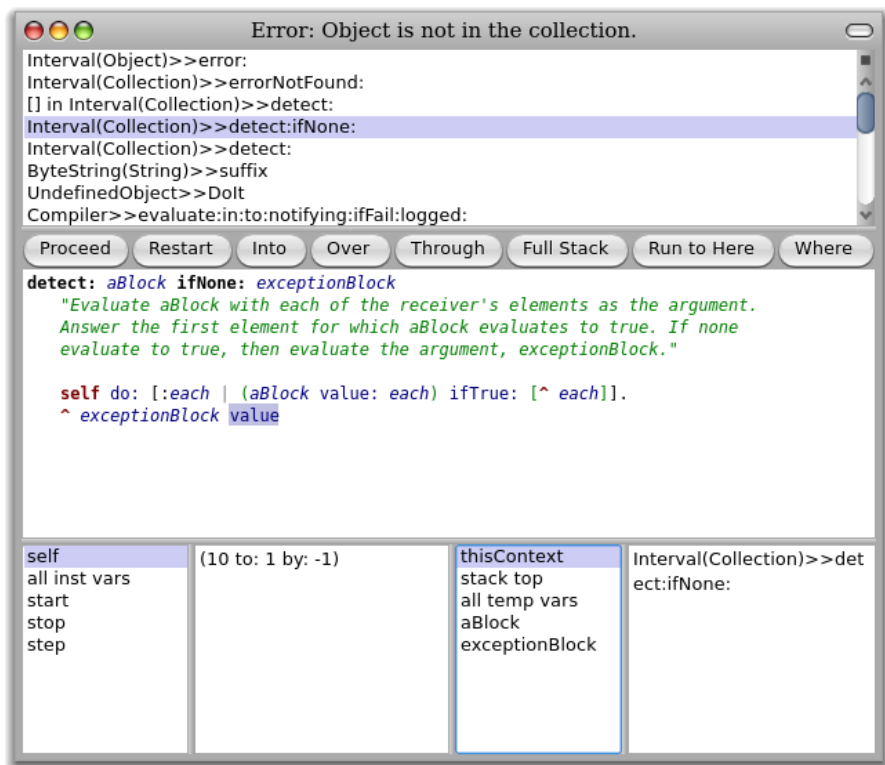


Figure 6.27: デバッガ

や `CMD-I` を使って取り消せます。

右側のインスペクタには、現在のコンテキストにおける一時変数が表示されます。図 6.27 では、`value` が、引数 `exceptionBlock` に送られています。

スタックトレースで一つ下のメソッドを見ると、`exceptionBlock` は `[self errorNotFound: ...]` です。エラーメッセージが表示されるのは驚くことではありません。

ちなみに、ミニインスペクタに表示された変数に対して完全なインスペクタやエクスプローラを開きたい場合は、単に変数名をダブルクリックするか、変数名を選んでアクションクリックし、`inspect(i)` もしくは `explore(I)` を選びます。こうしておく、他のコードの実行中に変数の変化を見たい場合に便利です。

メソッドウィンドウに戻って見てみると、メソッドの最後から 2 番目の行で、文字列 `'readme.txt'` からドットを見つけようとしており、その実行が目的の行に達していないことがわかります。Pharo は逆向きに実行させることはでき

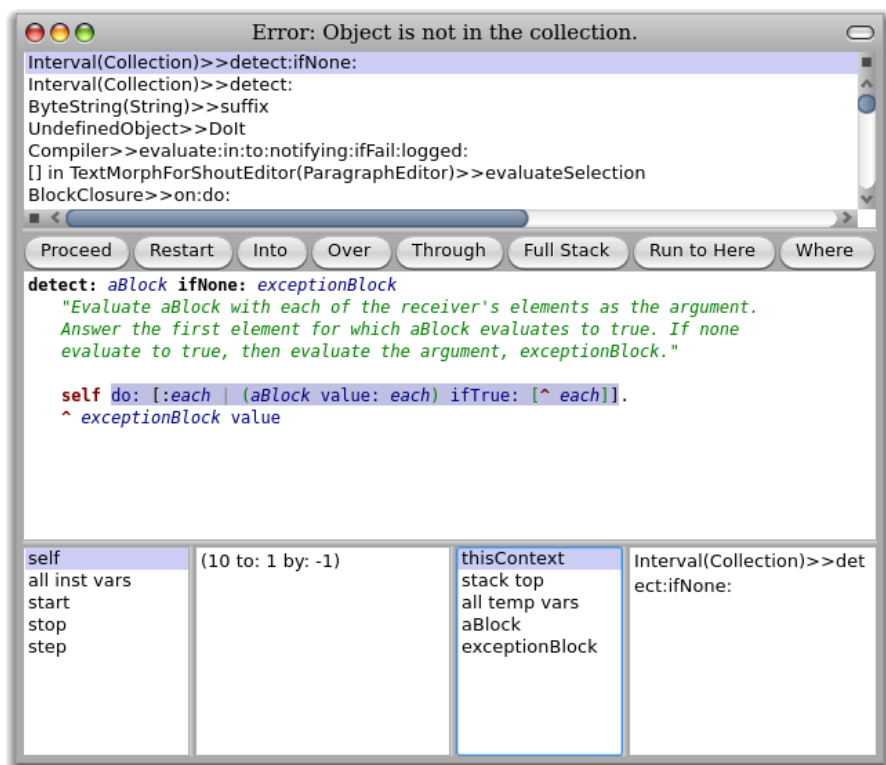





Figure 6.28: detect: ifNone: メソッドを再実行した後のデバッガ

ませんが、メソッドを再実行させることは可能で、オブジェクト自体を変化させず、新しいオブジェクトを生成するようなコードの場合にうまく働きます。


 **Restart** をクリックしてください。すると実行位置が現在のメソッドの最初の文に戻ります。次に送信されるメッセージ `do:` が青くハイライトして表示されています(図 6.28 参照)。

**Into** と **Over** ボタンは二つの異なる方法でステップ実行します。**Over** ボタンを押した場合、Pharo は現在のメッセージ送信(この場合では `do:`)を、エラーが発生しない限り 1 ステップだけ実行します。つまり **Over** ボタンによって、現在のメソッドの次のメッセージ送信へと移ります。この場合は `value` となります。— これは、まさしく最初にいた場所なので、大した助けにはなりません。必要なことは、なぜ `do:` メソッドが求めている文字を見つけられないのかを突き止めることです。

 **Over** ボタンをクリックした後で、**Restart** ボタンをクリックして図 6.28 の状態に戻ってください。

 **Into** ボタンをクリックしてください。Pharo はハイライトされていたメッセージ送信に対応するメソッドの中に入ります。この場合は、`Collection>>do:` です。


しかし、どちらにしてもこれはあまり助けにならないことがわかります。というのも `Collection>>do:` が壊れていないことはかなり確信が持てます。Pharo に対して何かをさせているところにバグがあるはずですが、この場合、**Through** ボタンを用いるのが適しています。do: メソッド自身の詳細を無視して、引数ブロックの実行に焦点を当てたいからです。

 再度 `detect:ifNone:` メソッドを選び、**Restart** ボタンで図 6.28 の状態に戻します。今度は **Through** ボタンを数回クリックします。このとき、コンテキストウィンドウの `each` 変数を選択しておきます。do: メソッドの実行に従って、`each` の内容が 10 からカウントダウンしていくことがわかります。

`each` の値が 7 のとき、`ifTrue:` ブロックが実行されることを期待しているのですが、そうなっていません。何が悪いのかを見るため、図 6.29 で示すように `value:` を **Into** ボタンでステップ実行します。

**Into** ボタンをクリックした後、図 6.30 で示した位置にすることがわかります。一見すると `suffix` メソッドに戻ってしまったように見えますが、`suffix` メソッドが `detect:` メソッドに引数として渡しているブロックを実行中なのです。コンテキストインスペクタで `dot` を選ぶと、その値が `'.'` であることがわかります。ここに至って、なぜイコールとならなかったかわかりました。`'readme.txt'` の 7 番目の文字は `Character` なのに、`dot` が `String` となっています。

バグが判明したので修正方法も明白です。`dot` を探す前に、文字へ変換する必要があります。

 `dot := FileDirectory dot first` のように、デバッガの中でコードを正しく修正し、変更を **accept** してください。

`detect:` の内側にあるブロック内のコードを実行しているため、この修正によっていくつかのスタックフレームが失われることとなります。それでよいかどうかを Pharo が確認します (図 6.31 参照)、**yes** をクリックして、新しいメソッドを保存 (およびコンパイル) します。

`'readme.txt'` `suffix` 式の評価は完了し、`'txt'` という答えを出力します。

この答えは正しいでしょうか？ 残念ながら、明確に答えることができません。サフィックスは `.txt` であるべきでしょうか？ それとも `txt` ? `suffix` メソッドのコメントは全然明確ではありません。こうした問題を避ける方法は、答えを定義する SUnit テストを書くことです。

### Method 6.2: `suffix` メソッドのための簡単なテスト

```
testSuffixFound
self assert: 'readme.txt' suffix = 'txt'
```

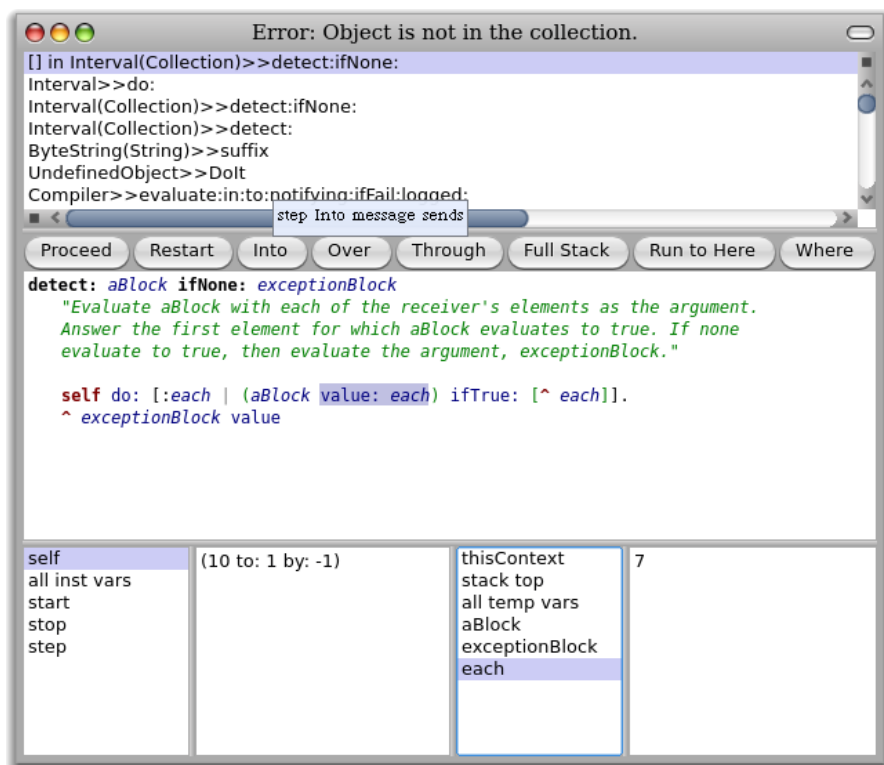


Figure 6.29: do: メソッドを数回 Through ボタンでステップ実行した後のデバッグ

テストの記述に必要な労力は、同じテストをワークスペースで実行するよりも若干多くなりますが、SUnitを使うことで、テストを実行可能な文書として保持し、他人が簡単にテストできるようになります。さらに、StringTestクラスに Method 6.2 を追加して SUnit でテストを実行すれば、エラーのデバッグに素早く戻ることができます。SUnit はアサーション違反があるとデバッグを開きますが、図 6.32 のように、スタックフレームを一つ遡り、Restart ボタンでテストを再実行し、Into ボタンで suffix メソッドに入ることでエラーを修正できます。その後、SUnit Test Runner で Run Failures ボタンをクリックして、テストが通過することを確認するだけです。

以下はより良いテストコードです。

#### Method 6.3: suffix メソッドのためのより良いテスト

```

testSuffixFound
self assert: 'readme.txt' suffix = 'txt'.
self assert: 'read.me.txt' suffix = 'txt'

```

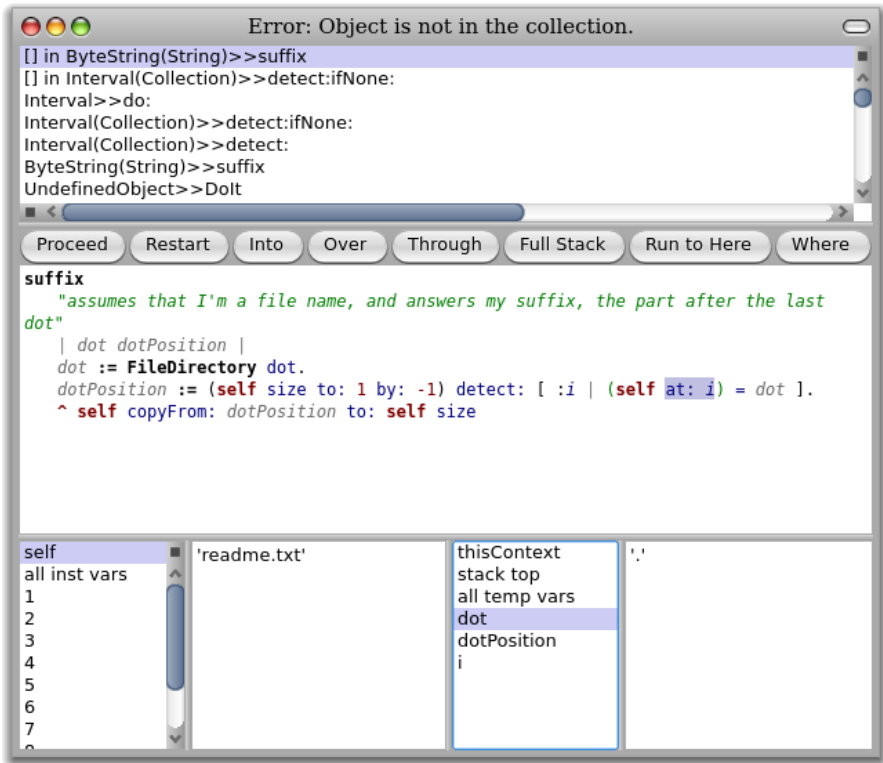


Figure 6.30: 'readme.txt' at: 7 が dot と等しくならない原因を示しているデバッガ

なぜこのテストがより良いものなのでしょう？ その理由は、対象となる文字列に一つ以上のドットがある場合に、このメソッドがどのように振る舞うかを読む人に教えてくれているからです。

エラーやアサーション違反の捕捉以外にも、デバッガに入るいくつかの方法があります。コードの実行によって無限ループに陥ってしまった場合、`CMD-`（あなたが英語を学んだ場所によりますが、終止符またはピリオドと呼ばれるもの）を入力することで、割り込みをかけてその計算状況におけるデバッガを開くことができます。<sup>11</sup> また、疑わしいコードの箇所に `self halt` と入れておくだけでもいいです。例えば、`suffix` メソッドであるならば、次のような感じです。

<sup>11</sup>同様に、どんなときでも `CMD-SHIFT-` の入力によって、緊急デバッガが起動することを覚えておくとう便利です。



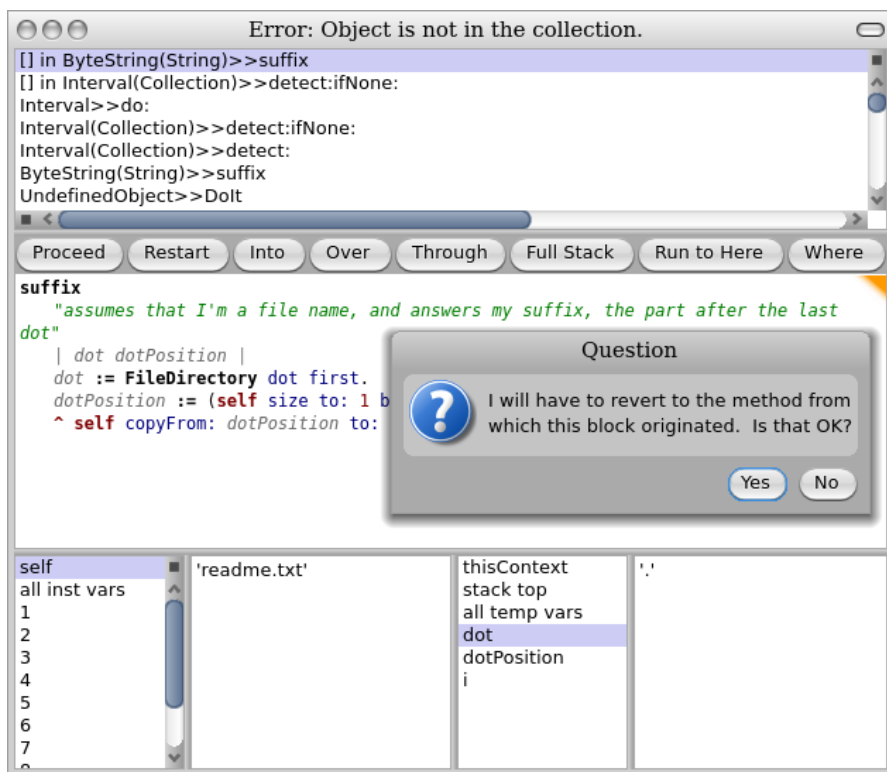


Figure 6.31: デバッガの中で suffix メソッドを修正する。内側のブロックから抜けだすかどうかの確認

#### Method 6.4: suffix メソッドに halt を入れる

```

suffix
"自分自身をファイル名と仮定して、最後のドットから後の部分をサフィックスとして返す。"
| dot dotPosition |
dot := FileDirectory dot first.
dotPosition := (self size to: 1 by: -1) detect: [:i | (self at: i) = dot ].
self halt.
↑ self copyFrom: dotPosition to: self size

```

このメソッドを走らせると、self halt の実行により pre-debugger が開きますので、そこから、実行を続行させたり、デバッガに入って変数を見たり、ステップ実行したり、コードを編集したりすることができます。

デバッガについては以上ですべてですが、suffix メソッドについてはまだです。最初のバグで気づいたでしょうが、対象となる文字列にドットがない場

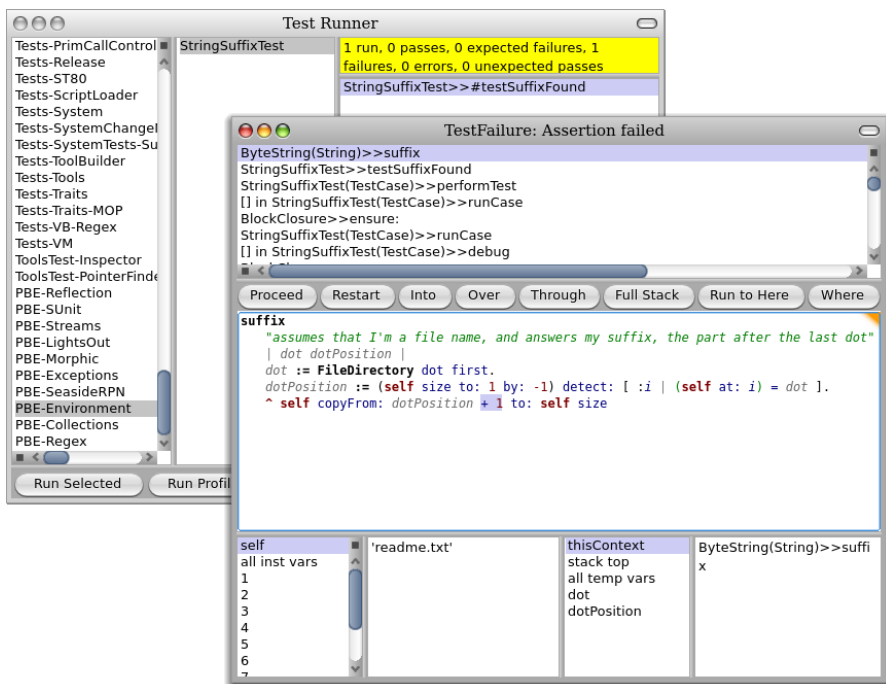


Figure 6.32: デバッガの中で suffix メソッドを編集する: SUnit のアサーション違反の後に Off-by-one エラーを修正する

合、suffix メソッドはエラーを発生します。これは望ましい振る舞いではないので、こうしたケースで起きることを明示するために二つ目のテストを追加しましょう。

Method 6.5: suffix メソッドのための二つ目のテスト: 対象がサフィックスを持たない

```
testSuffixNotFound
self assert: 'readme' suffix = ''
```

🕒 Method 6.5 を StringTest クラスのテストスイートに追加し、テストがエラーを起こすところを見ます。SUnit でエラーが起きたテストを選んでデバッガに入り、テストを通過させるためにコードを編集します。最も簡単で明解な方法は、detect: メッセージの代わりに detect:ifNone: を使い、2 番目の引数を単に文字列の大きさを返すブロックとすることです。

SUnit については第 7 章 でより詳しく学びます。

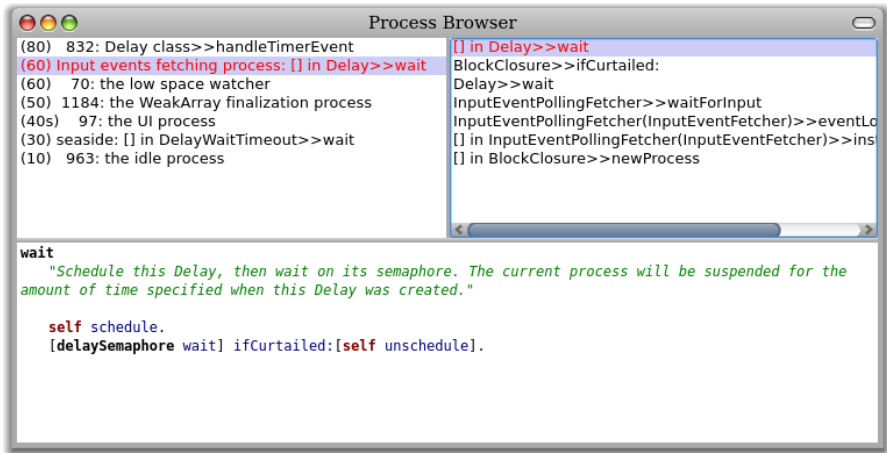


Figure 6.33: プロセスブラウザ

## 6.6 プロセスブラウザ

Smalltalk はマルチスレッドなシステムです。イメージ中では、たくさんの軽量プロセス (スレッドとして知られる) が並行して走っています。将来的には Pharo の仮想機械は可能ならばマルチプロセッサを利用するようになるかもしれませんが、現時点では、平行性は時分割処理で実装されています。

プロセスブラウザは、デバッガの親類で、Pharo の中で稼働している様々なプロセスを見せてくれます。図 6.33 にスクリーンショットを示します。左上のペインは Pharo の中のすべてのプロセスのリストで、優先度 80 の割り込みタイマー監視のプロセスから、優先度 10 のアイドルプロセスに至るまでが優先度順に並んでいます。もちろん、単一プロセッサ上ですので、見ているときに動いている唯一のプロセスは UI プロセスです。他のすべてのプロセスは、何らかのイベントを待っています。デフォルトでは、プロセスの表示は静止していますが、アクションクリック `ing` で `turn on auto-update(a)` を選ぶことで更新することができます。

左上のペインでプロセスを選ぶと、ちょうど、デバッガのように、右上のペインにそのプロセスのスタックトレースが表示されます。スタックトレースを選ぶと、対応するメソッドが下部のペインに表示されます。プロセスブラウザは `self` や `thisContext` のためのミニインスペクタを備えてはいませんが、スタックトレース上でアクションクリック `ing` することにより同様の機能を提供します。

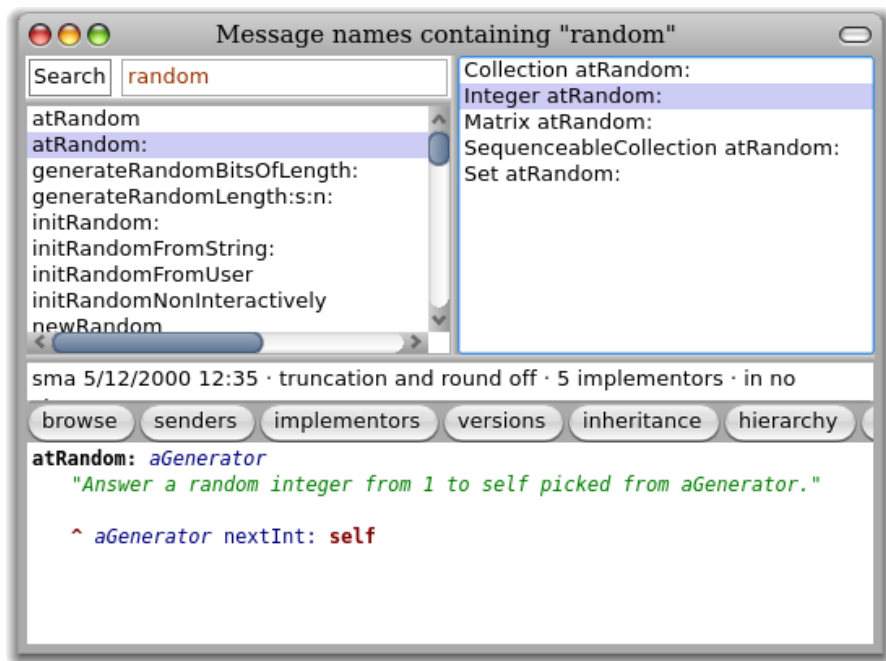


Figure 6.34: random を部分文字列としてセレクト名に持つすべてのメソッドを表示しているメッセージ名ブラウザ

## 6.7 メソッド検索

Pharo にはメソッド検索のための二つのツールがあります。両者はインターフェイスも機能性も異なっています。

メソッド・ファインダは 1.9 節 である程度説明したように、メソッドを名前や機能性から探すことができます。しかし、メソッドの本体を見ようとすると、メソッド・ファインダは新たにブラウザを開いてしまいます。これではすぐにウィンドウで溢れてしまいます。

メッセージ名ブラウザは、より限定的な検索機能を持っています。検索用の入力欄にメッセージ・セレクトの一部を入力すると、図 6.34 のように、名前にその一部を含んだすべてのメソッドがブラウザに表示されます。さらに、これは一人前のブラウザでもあります。左側のペインにある名前の一つを選択すると、その名前を持つすべてのメソッドが右側のペインに表示され、下部のペインでブラウズできます。ブラウザと同様に、メッセージ名ブラウザはボタン・バーを備えており、選択したメソッドやそのクラスを他のブラウザで開くことができます。

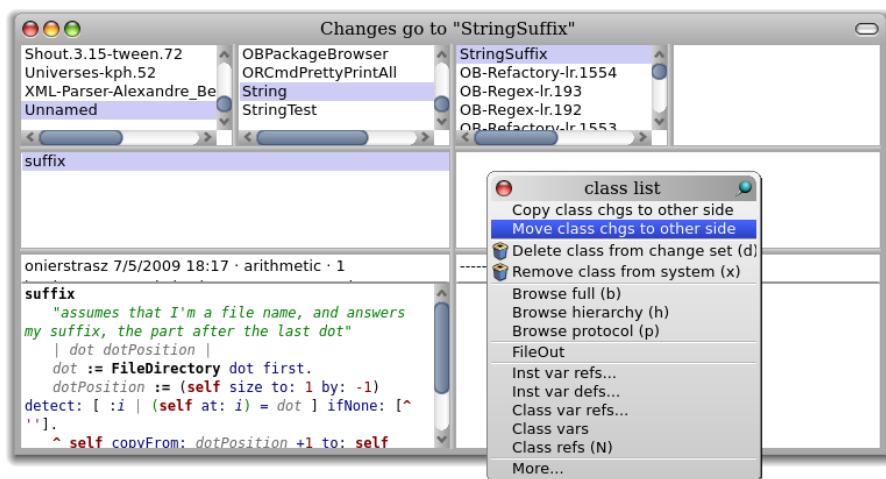


Figure 6.35: チェンジソーター

## 6.8 チェンジセットとチェンジソーター

Pharo で作業をしている間、メソッドやクラスに対するどのような変更もチェンジセットに記録されます。新しいクラスを作成したこと、クラス名を変更したこと、カテゴリを変更したこと、既存のクラスにメソッドを追加したことなど—重要な変更はすべて記録されます。しかし、きままな *doit* は含まれないので、例えばワークスペースで新しくグローバル変数を定義した場合には、変数の作成はチェンジセットに記録されません。

多くのチェンジセットが存在していますが、常にそのうちの一つだけがイメージに対する変更を収集します。—これを **カレントチェンジセット** と呼びます— **World ▸ Tools ... ▸ Change Sorter** を選ぶと現れるチェンジソーターを使えば、どれがカレントチェンジセットなのか確認したり、すべてのチェンジセットについて調べたりすることができます。

図 6.35 にチェンジソーターを示します。タイトルバーは、どのチェンジセットがカレントチェンジセットなのかを表すとともに、カレントチェンジセットが選択された状態でチェンジソーターが開きます。

左上のペインで他のチェンジセットを選ぶことができます。アクションクリックメニューで他のチェンジセットをカレントチェンジセットにしたり、新しいチェンジセットを作ったりすることができます。右隣のペインには、選択したチェンジセット (とそのカテゴリ) が影響を与えるすべてのクラスがリスト表示されます。クラスの一つを選ぶと、(クラスのすべてのメソッドではなく) そのチェンジセットに存在するメソッドの名前が左中央のペインに表示され、さらにメソッド名を選ぶと下のペインにメソッドの定義が表示されます。チェ

ンジソーターは、クラスの生成自体がチェンジセットの一部であるかどうかを示さないことに注意してください。なお、クラス生成に関する情報は、チェンジセットを表現するためのオブジェクト構造には格納されています。

チェンジソーターでは、クラスやメソッドの上でアクションクリックメニューを使うことで、チェンジセットから削除することができます。

チェンジソーターは、同時に二つのチェンジセットを扱え、一つは左側でもう一つは右側に表示されます。この配置により、チェンジソーターの主な特徴である、一方のチェンジセットからもう一方へと変更の移動やコピーが支援されます。図 6.35 にアクションクリックメニューを示します。個々のメソッドを一方からもう一方へコピーすることもできます。

なぜ、チェンジセットの合成に注意を払わなければならないのか、不思議に思うかもしれません。その答えは、チェンジセットが Pharo からファイルシステムにコードを書き出すシンプルな仕組みを提供するからです。つまり、コードを他の Pharo イメージに取り込んだり、Pharo 以外の Smalltalk に取り込んだりすることができるからです。チェンジセットの書き出しは“ファイルアウト”として知られているもので、どんなチェンジセットでも、どんなブラウザのクラスやメソッドでも、アクションクリックメニューにより行うことができます。ファイルアウトを繰り返すと、ファイルの新しいバージョンが生成されますが、チェンジセットは Monticello のようなバージョン管理ツールではないので、依存関係の追跡は行いません。

Monticello の出現以前、チェンジセットは Smalltalker の間でコードを交換する主な手段でした。チェンジセットは、単純さ (ファイルアウトしたものは単なるテキストファイルですが、テキストエディタで編集するようなことはお勧めしません) と可搬性の高さという利点を持っています。

チェンジセットの主な欠点は、Monticello パッケージに比べて依存関係の概念をサポートしていないということです。ファイルアウトしたチェンジセットは、読み込まれる際にイメージに変更を加えるアクションの集合です。チェンジセットを無事に読み込むためには、イメージはふさわしい状態にあることが求められます。例えば、チェンジセットがあるクラスへメソッドを追加するアクションを含んでいる場合、あらかじめイメージにそのクラスが定義されているときのみ、読み込みを完了できます。同様にチェンジセットが、クラス名の変更やカテゴリの変更を行う場合も、当然そのクラスがイメージに存在しているときのみ正しく動作します。ファイルアウト時には、クラスに定義されているインスタンス変数をメソッドが参照しているのに、それを取り込もうとしているイメージには変数が存在していないこともあり得ます。問題は、チェンジセットがファイルインできる条件を明示していないことです。ファイルインのプロセスはうまくいくことが望まれますが、うまくいかない場合、たいいていわけのわからないエラーメッセージとスタックトレースで終わることになります。たとえファイルインが成功しても、あるチェンジセットが別のチェンジセットによる変更を勝手に元に戻してしまうことすらあります。

これとは対照的に Monticello パッケージではコードを宣言的な流儀で表現します。つまり、読み込みの完了後にイメージがどのような状態であるべきか

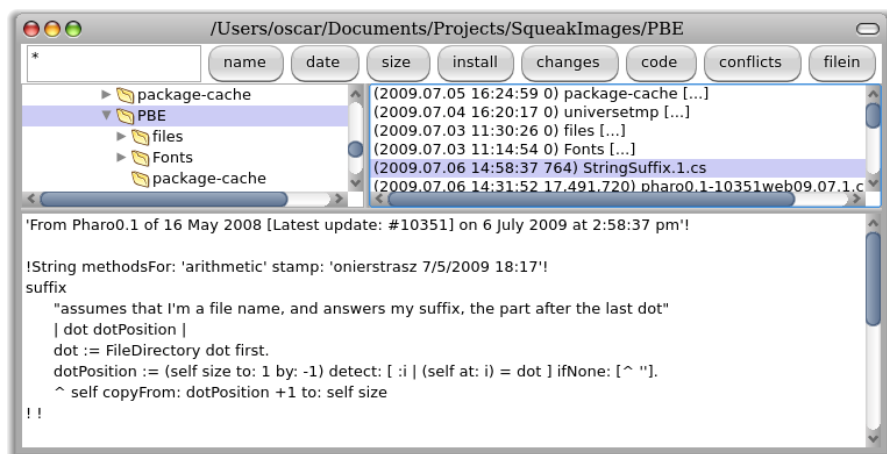


Figure 6.36: ファイルリストブラウザ

を記述します。これにより (二つのパッケージが最後の段階で矛盾した要求を持つ場合など) コンフリクトについて警告し、依存関係の順序で一連のパッケージをロードすることを提案します。

こういった欠点にも関わらず、チェンジセットは未だに使われています。特にインターネット上では、参考にしたリ場合によっては使いたいと思うチェンジセットが見つかるかもしれません。さて、チェンジソーターを使ってチェンジセットをファイルアウトするところを見てきたので、今度はファイルインの方法を説明しましょう。それには別のツール、ファイルリストブラウザが必要になります。

## 6.9 ファイルリストブラウザ

ファイルリストブラウザ は、実際のところ Pharo からファイルシステム (FTP サーバーも含む) をブラウズするための汎用的なツールです。**World ▸ Tools ... ▸ File Browser** メニューで開くことができます。もちろん、みなさんのローカルのファイルシステムの内容次第で見えるものは変わりますが、典型的な見え方を 図 6.36 に示します。

最初にファイルリストブラウザを開くと、カレントディレクトリ、つまり Pharo を起動したディレクトリが選択された状態になります。タイトルバーにはこのディレクトリへのパスが表示されます。左側の大きなペインは、ごく一般的なやり方でファイルシステムをたどるのに使えます。ディレクトリが選択されると、そのディレクトリ内のファイル (ディレクトリは含まれない<sup>12</sup>) のリ

<sup>12</sup>訳注：実際には表示されます

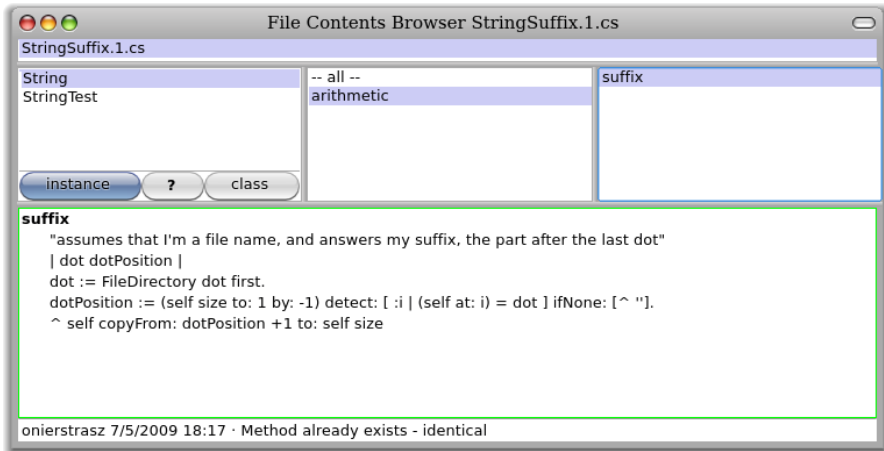


Figure 6.37: ファイルコンテンツブラウザ

ストが右側に表示されます。このファイルのリストは、Unix 流のパターンをウィンドウの左上にある小さな入力ボックスに入力することでフィルタをかけることができます。最初、このパターンは\*であり、すべてのファイル名にマッチしますが、異なる文字列を入力してアクセプトすると、パターンが変更されます。(入力したパターンの前後には、暗黙的に\*が付加されることに気をつけてください) ファイルの並び順は、`name`、`date` そして `size` ボタンで変更できます。その他のボタンはブラウザで選択したファイル名に依存します。図 6.36 では、ファイル名のサフィックスが `.cs` であると、ブラウザがチェンジセットだと推測し、`install` する (この場合、そのファイル名に基づいた名前前のチェンジセットにファイルインされます) ためのボタンや、ファイルの変更点を見るための `changes` ボタン、ファイル内のコードを調べるための `code` ボタン、カレントチェンジセットにコードを取り込むための `filein` ボタンを提供します。`conflicts` ボタンは、イメージに既にあるコードとコンフリクトするようなチェンジセットの変更を教えてくれるものと推測されるかもしれませんが、それは違います。その代わりに (改行の有無など) ファイルが正しく読み込まれるかといった潜在的な問題をチェックします。

ボタン表示の選択はファイル内容ではなくファイルの名前に依存していることから、時折期待するボタンが画面に現れないことがあります。しかし、すべてのオプションは常にアクションクリックの `more ...` から利用できるのです、簡単にこの問題へ対処できます。

`code` ボタンは、おそらくチェンジセットを扱う上で最も便利なもので、チェンジセットファイルの内容をブラウザで開きます。例を 図 6.37 に示します。このファイルコンテンツブラウザはブラウザによく似ていますが、クラスとプロトコルとメソッドだけを表示し、カテゴリを表示しません。個々のクラスご



とに、そのクラスが既にシステムに存在しているかどうか、ファイルで定義されているかどうか（定義が同一か否かは示しません）がわかるようになっています。個々のクラスのメソッドを表示した上で、さらに（図 6.37 のように）現在のバージョンとファイルのバージョンとの差分を表示します。上部の四つのペインのコンテキストメニューでは、チェンジセット全体をファイルインするの、それとも対応するクラスや、プロトコル、メソッドごとにファイルインするかどうかを指示できます。

## 6.10 Smalltalk では、コードを失うことはありません

Pharo がクラッシュすることは十分あり得ることで、実験的なシステムであり、Pharo が機能するために必要なものを含め、すべてが変更可能になっているからです！

🔔 悪意を持って *Pharo* をクラッシュさせましょう。Object become: nil を試してみてください

クラッシュしてイメージが何時間も前に保存したバージョンに戻ってしまったとしても、良い知らせがあります。それはみなさんがやったことは決して失われぬということです。というのも、実行したあらゆるコードが *.changes* ファイルに保存されているからです。あらゆるコードです！これにはワークスペースで評価した 1 行プログラムから、プログラミング中にクラスへ追加したコードまで含まれます。

ここでは、どのようにコードを回復するかという手順を説明します。必要などきまで、特に読む必要はありませんが、そうなったときに見つけられるよう、ここでみなさんをお待ちしています。

最悪の場合では、*.changes* ファイルをテキストエディタで開くこともできますが、数メガバイトに達するため遅くなりますし、お勧めできません。Pharo は、より良い方法を提供します。

### コードを回復する方法

最後のスナップショットの *Pharo* をリスタートして、**World▷Tools ...▷Recover lost changes** を選びます。

すると、履歴をどれくらい遡ってブラウズしたいかを選ぶことができます。通常、最後のスナップショットまで遡って変更をブラウズすることで十分です。（同様の効果を *ChangeList browseRecent: 2000* というコードを編集することで得ることができます。ただし 2000 という数値は試行錯誤で決めることになります）

*recent changes* ブラウザが表示されますが、これは最後のスナップショットの後の変更で、Pharo に対して行ったすべての操作のリストとなります。この

リストからアクションクリックメニューで項目を削除することができます。これで十分と判断したら、残ったものをファイルインできます。このように、新しいイメージに変更を加えます。ファイルインする前に、普通のチェンジセットブラウザを使って新しいチェンジセットを作るのは良いアイデアです。これによって復元したすべてのコードが新しいチェンジセットの中に入るからです。その後、チェンジセットをファイルアウトすることができます。


`recent changes` ブラウザの便利な機能に `remove dolts` があります。たいてい、`do it` の操作はファイルイン (つまり再実行) したくないでしょう。しかし、例外もあります。クラス定義は `dolt` として表現されるからです。あるクラスのメソッドをファイルインする前に、クラスは存在していなければなりません。そこで、新しいクラスを定義する際には、最初にクラスを定義する `do it` をファイルインし、その後 `remove dolts` してからメソッドをファイルインします。

私がりカバリが完了するときは、新しいチェンジセットをファイルアウトした後、イメージを保存することなく `Pharo` を終了します。再起動して、きれいな状態に戻ってから新しいチェンジセットを適用するようにしています。

## 6.11 まとめ

`Pharo` で開発作業を効果的に行うためには、開発環境が提供するツールについて学ぶ努力を惜しまないことが大切です。

- 標準的なブラウザは、既存のカテゴリやクラス、メソッドプロトコルやメソッドを調べたり、新しく定義したりするための中心的なインターフェースです。ブラウザは、メッセージのセンダやインプリメンタ、メソッドのバージョンへ直接ジャンプするといった、いろいろな便利なボタンを提供します。
- (OmniBrowser やリファクタリングブラウザといった) 様々なブラウザや、クラスやメソッドへの様々なビューを提供する (階層ブラウザなど) 特殊用途のブラウザがあります。
- どんなツールでも、クラスやメソッドの名前を選んだ後に `CMD-b` のキーボードショートカットを使えば、すぐにブラウザへジャンプできます。
- `SystemNavigation default` に対してメッセージを送ることで、`Smalltalk` システムをプログラミング的にブラウズできます。
- `Monticello` は、クラスやメソッドのパッケージを外部へ出力したり、取り込んだり、バージョン管理や共有するためのツールです。`Monticello` のパッケージは、カテゴリやサブカテゴリと他のカテゴリに属する関連メソッドプロトコルから構成されます。
- インспекタ と エクスプローラ の二つのツールは、イメージの中で生きているオブジェクトを調べたり対話したりするための便利なツールで

す。メタクリックすることにより **Morphic Halo** を出し、デバッグハンドル  を選ぶと、ツールをインスペクトすることもできます。

- デバッガは、エラーが起きた際にプログラムの実行スタックを見せるだけでなく、ソースコードを含め、アプリケーションのすべてのオブジェクトと対話することを可能にするツールです。多くの場合、デバッガでソースコードを修正し、実行を継続することが可能です。デバッガは **SUnit**(第7章) と並んでテストファースト開発を支援する効果的なツールです。
- プロセスブラウザは、イメージの中で現在実行中のプロセスを監視したり問い合わせたり対話したりできます。
- メソッド・ファインダ とメッセージネームブラウザは、メソッドの場所を突き止めるツールです。前者は、名前がはっきりわからないが期待される振る舞いがわかっている場合に便利です。後者は、名前の一部しかわからない場合に、ブラウズのための優れたインターフェースを提供します。
- チェンジセット は、イメージのソースコードに対するすべての変更を自動的に記録されるログのことです。主にソースコードのバージョンの保存や交換の手段としては、**Monticello** に取って代わられていますが、ごくまれに発生し得る壊滅的な故障からの復旧などでは未だに便利です。
- ファイルリストブラウザはファイルシステムをブラウズするツールです。これはファイルシステムからソースコードを **filein** することもできます。
- イメージを保存したり **Monticello** でソースコードをバックアップしたりする前にクラッシュしてしまった場合、チェンジリストブラウザを使うことで最新の変更から回復することができます。再実行したい変更を選んで、最新のイメージのコピーに取り込むことができます。



# Chapter 7

## SUnit

### 7.1 はじめに

SUnit は小さいながらも強力な、テストの作成と運用のためのフレームワークです。その名前からわかるように、SUnit はユニットテストを対象として設計されていますが、統合テストや機能テストにも使えます。初期のSUnit はケント・ベック (Kent Beck) が開発しました。その後ジョセフ・ペルリン (Joseph Pelrine) を始めとする開発者たちが、SUnit にテストリソース (7.6 節) の概念を取り入れました。

テストとテスト駆動開発のメリットは、Pharo や Smalltalk にとどまりません。テストの自動化はアジャイルソフトウェア開発の特徴であり、ソフトウェアの品質を上げたい開発者なら採用するでしょう。ユニットテストの効果を高く評価する開発者は多く、今では多くの言語に *xUnit* ライブラリがあります (Java、Python、Perl、.Net、Oracle など)。この章ではSUnit 3.3 (原文の執筆時のバージョン) について解説します。SUnit の最新版は、[sunit.sourceforge.net](http://sunit.sourceforge.net) で確認できます。

テストをすることやテストスイートを作ることは、新しい考えではありません。誰でもテストはエラーを発見する良い方法だと知っています。エクストリーム・プログラミング (eXtreme Programming) ではテストが中心的なプラクティスと位置付けられ、テストの自動化が重視されました。その結果、テストはプログラマにとって面倒な作業から生産的で楽しい作業に変わりました。Smalltalk コミュニティは、長らくテストを重視してきました。Smalltalk のプログラミング環境ではインクリメンタルな開発が容易だからです。Smalltalk の伝統的な開発では、プログラマはメソッドを実装するとすぐにワークスペースでテストします。メソッドのコメントにテストを書いたり、セットアップが必要なテストはサンプルメソッドとして実装したりする場合があります。しかしこれらの方法には問題があります。まずワークスペースにテストを書いた場合 (イメージファイルを共有しない限り) 他のプログラマから使

えません。コメントやサンプルメソッドにした場合、状況は多少改善されますがそれでも、メソッドの開発に合わせてテストを見直し、さらに自動的に実行させるのは簡単ではありません。テストが実行されなければバグも見つけれません! さらにサンプルメソッドは、何が期待された動作なのかについて何も語りません。サンプルを実行して—意外な—結果が得られるかもしれません、それが正しい振る舞いかどうかはわからないのです。

SUnit は、テストの結果が正しいかどうかをテスト自身で検証できるところに価値があります。またテストをグループ分けしたり、テストに必要なコンテキストを準備したり、グループ単位でテストを自動実行することも可能になります。もうワークスペースに小さなコードを書かなくても、2分もあれば SUnit でテストが書けるようになります。SUnit を使ってテストの自動化をしましょう。

この章では、テストを行う理由とより良いテストを書く方法の議論から始めます。小さな例を挙げながら SUnit の使い方を紹介し、最終的には SUnit の内部実装を見ていきます。SUnit を支えている Smalltalk のリフレクションの力がわかるはずです。

## 7.2 なぜテストは重要か

残念ながら、テストを時間の無駄と思い込んでいる開発者もたくさんいます。彼らはバグを作りません—他のプログラマだけがバグを作るのです。誰も一度くらいこうやってしまった経験があります: 「時間があればテストを書きますよ。」絶対にバグのないコードが書いて、そのコードが将来にわたって絶対に変更されないなら、確かにテストは時間の無駄です。しかし、そのようなコードはごく小さいか、あるいはあなたを含む誰も使わないようなものがほとんどです。テストは将来への投資だと考えてください。テストはすぐにでも役に立ちますが、将来アプリケーションや実行環境が変わったとき、絶大な効果を発揮します。

テストにはいくつかの役割があります。一つは生きたドキュメントです。すべてのテストに成功すれば、ドキュメントは最新版です。もう一つは、変更したコードが無事動くという確信です。もしシステムの一部が壊れたとしても、すぐに原因を見つけることができます。最後に、コードを書くとき—あるいはさらに進んでコードを書く前に—テストを書くことで、実装について考えるよりもむしろ設計しようとしている機能について、そしてそれをどうクライアントコードに見せるのかについて考えるようになります。コードを書く前にテストを書くことで、コードが動作するのに必要なコンテキスト、コードとクライアントとのやりとり、期待される結果をまず明確にしなければならなくなります。きっとコードの品質が向上します。ぜひ試してみてください。

現実のあらゆるアプリケーションのすべての側面をテストするのは不可能です。いくら良質のテストスイートを書いても、システムを壊す機会を窺っているバグが入り込みます。完全なテストを書くことを目標にするのではなく、バグを見つけたらすぐにあぶり出すテストを書き、実行し、失敗を確認し、修

正を始めるようにしましょう。テストが成功すれば修正作業は終わりです。

## 7.3 良いテストを書くには?

良いテストが書けるようになるには、練習するのが一番です。テストのメリットを最大限に活かすヒントを見ていきましょう。

1. テストは繰り返し実行可能であるべきです。実行したいときに実行でき、常に同じ結果を出すべきです。
2. テストは人の手を借りずに実行可能であるべきです。テストはたとえば夜中でも実行可能であるべきです。
3. テストは物語であるべきです。各テストの目的を一つに絞るべきです。テストをシナリオと考え、読めばアプリケーションの機能が理解できるようにすべきです。
4. テストの変更はなるべく控えるべきです。アプリケーションを変更するたびにテストを変更する必要があっては大変です。テストを変更せずにすませるには、主に公開インターフェースを使ってテストを書きましょう。プライベートな「ヘルパー」メソッドについては、それ自体が複雑な処理をするようならテストを書いてもいいでしょう。ただし、公開インターフェースの内部実装が変わったときには、そのテストを変更するか捨てる必要があるかもしれないことに気をつけましょう。

テストの目的を絞り (3)、テストすべき機能とテストの数とのバランスを取りましょう。システムの一部を変更したときにすべてのテストではなく、限られた数のテストだけが失敗するようにすべきです。なぜなら、100 のテストに失敗した方が 10 のテストの失敗ですんだことより重大なメッセージだからです<sup>1</sup>。ただし、いつもこの通りにできるとは限りません。例えば変更によってオブジェクトの初期化やテストのセットアップに問題が出れば、すべてのテストが失敗してしまいます。

エクストリーム・プログラミングでは、コードを書く前にテストを書くこと (テストファースト) を提唱しています。これはソフトウェア開発者の本能与相反するのに見えます。我々に言えるのは、「さあ一歩前に進んで試してみよう」ということです。わかっているのは、テストを先に書くことで、どんなコードを書くべきか、何をすべきか、どう要件をクラスに落とせばいいか、どんな API にすればいいのかを考える手助けになることです。さらにテストファースト開発は、開発速度を上げる自信にもつながります。開発を急ぐあまりに肝心な点を見落とすかもしれない、と言った心配をする必要がなくなるからです。

---

<sup>1</sup> 訳注: 一つの原因ですべてのテストに失敗するようでは、テストの失敗の数から問題の重大さがわかりません。

## 7.4 SUnit by example

SUnit の詳細に触れる前に、一つずつ例を挙げていきます。Set クラスのテストを例に進めます。例に従ってコードを入力してみましょう。

### ステップ 1: テストクラスを作る

🕒 まず `TestCase` のサブクラス `ExampleSetTest` を作りましょう。このクラスに二つのインスタンス変数 `full` と `empty` を追加しましょう。

#### Class 7.1: `ExampleSetTest` クラス

```
TestCase subclass: #ExampleSetTest
  instanceVariableNames: 'full empty'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'MySetTest'
```

`ExampleSetTest` クラスに `Set` クラスのテストをまとめます。このクラスにはテスト実行時のコンテキスト (テストの実行に必要な状況、状態) を定義します。ここではコンテキストは二つのインスタンス変数 `full` と `empty` で表します。`full` は要素を持つ `Set` のインスタンス、`empty` は空の `Set` のインスタンスを表します。

クラスの名前は SUnit にとって重要ではありませんが、慣例により `Test` で終わる名前にすべきです。Pattern クラスのテストクラスなら、`PatternTest` とすれば、ブラウザで見たとときに (どちらも同じカテゴリであれば) 両方のクラスがアルファベット順に並びます。テストクラスが `TestCase` のサブクラスであることが重要です<sup>2</sup>。

### ステップ 2: テストのコンテキストを初期化する

`setUp` メソッドで、テスト実行時のコンテキストを定義します。ちょっと `initialize` メソッドに似ていますね。`setUp` は各テストメソッド (後述) の実行前に実行されます。

🕒 `setUp` メソッドを次のように定義しましょう。インスタンス変数 `empty` に空の `Set` を代入し、`full` に二つの要素を持つ `Set` を代入しましょう。

#### Method 7.2: フィクスチャをセットアップする

```
ExampleSetTest>setUp
  empty := Set new.
  full := Set with: 5 with: 6
```


<sup>2</sup>訳注: 名前に関する慣例と違い、SUnit にとって重要です。



テスト用語では、コンテキストのことをテストのフィクスチャと呼びます。

### ステップ 3: テストメソッドを書く

それではテストを書いてみましょう。ExampleSetTest クラスにテスト用のメソッドを定義します。一つのメソッドで一つのテストを表します。テストメソッドは、名前を『test』で始めます。SUnit は『test』で始まるメソッドを集めて、テストスイートとしてまとめます。テストメソッドは引数を取りません。

 次のテストメソッドを定義しましょう。

最初のテスト testIncludes は、Set の includes: メソッドをテストします。このテストは、5 を要素に持つ Set にメッセージ includes: 5 を送信すると true が返ると言っています。明らかにこのテストは、setUp メソッドが先に実行されていることを前提にしています。

#### Method 7.3: 要素を含むことをテスト する

```
ExampleSetTest>>testIncludes
self assert: (full includes: 5).
self assert: (full includes: 6)
```

二つ目のテスト testOccurrences は、既に 5 が含まれている Set にさらにもう一つの 5 を追加しても、5 の数は 1 のままであることを検証しています。

#### Method 7.4: 要素の数をテスト する

```
ExampleSetTest>>testOccurrences
self assert: (empty occurrencesOf: 0) = 0.
self assert: (full occurrencesOf: 5) = 1.
full add: 5.
self assert: (full occurrencesOf: 5) = 1
```

最後に、5 を削除すると Set は 5 を含まなくなることをテストします。

#### Method 7.5: 削除をテスト する

```
ExampleSetTest>>testRemove
full remove: 5.
self assert: (full includes: 6).
self deny: (full includes: 5)
```

真でないことをアサート (assert) するのに deny: メソッドを使っていることに注意してください。aTest deny: anExpression は aTest assert: anExpression not と等価ですが、よりわかりやすくなります。

## ステップ 4: テストを実行する

テストを実行するにはブラウザを使うのが一番簡単です。パッケージ、クラス名、テストメソッドのいずれかをアクションクリックし、**run the tests (t)** を選択するだけです。テストが成功したメソッドには緑のマークが、失敗したメソッドには黄のマークが付きます。クラスにもマークが付きます。つまり、すべてのテストに成功すると緑のマークが、いくつか失敗すると緑と赤の混じったマークが、全部失敗すると赤のマークがクラスに付きます<sup>3</sup>。

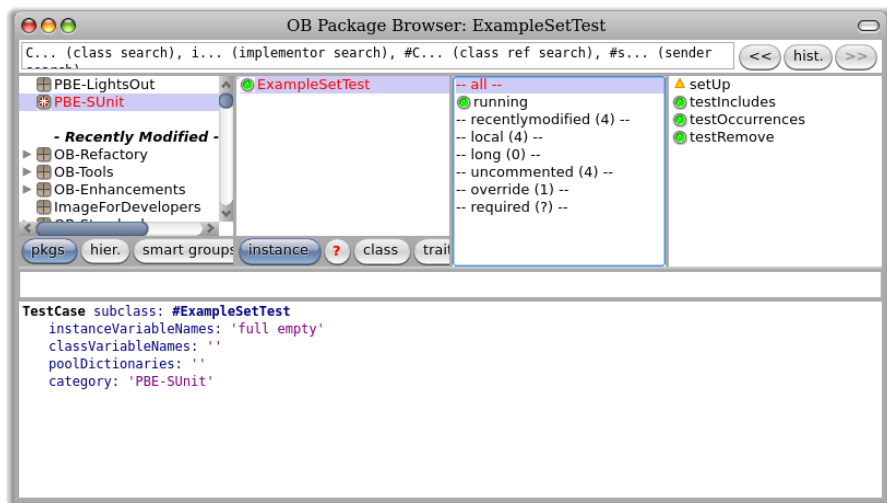


Figure 7.1: ブラウザから SUnit のテストを実行する

SUnit テストランナー (図 7.2) を使えば、実行するテストスイートを複数選択してテスト結果の詳細なログを見ることができます。テストランナーは **World ▸ Test Runner** で開きます。テストランナーは、まとまったテストを簡単に実行できるように設計されています。最も左にあるペインには、テストクラス (すなわち `TestCase` のサブクラス) を含むすべてのカテゴリが表示されます。カテゴリを選択すると、そのカテゴリに含まれるテストクラスのクラス階層が隣のペインに表示されます。抽象クラス (テストメソッドを含まないテストクラス) はイタリック体で表示され、テストクラスの階層はインデントによって示されます。例えば、`ClassTestCase` のサブクラスは `TestCase` のサブクラスより深くインデントされます<sup>4</sup>。

🔍 テストランナーを開いて **MySetTest** カテゴリを選択し、**Run Selected** ボタンをクリックしましょう。

<sup>3</sup> 訳注: PBE.image の場合。

<sup>4</sup> 訳注: `TestCase` が表示されるのは `SUnit-Kernel` カテゴリを選んだ場合か、どのカテゴリも選ばない場合です。

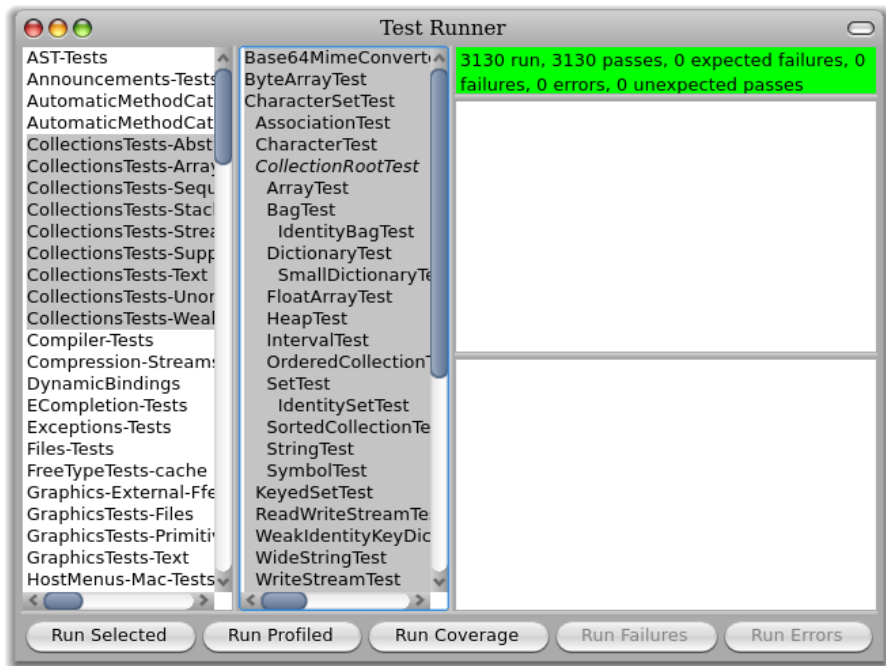


Figure 7.2: Pharo SUnit テストランナー

🕒 **ExampleSetTest**»testRemove にバグを入れて、再度テストを実行してみましょう。例えば5を4にします。

失敗したテスト(あれば)は、テストランナーの右のペインに表示されます。失敗の原因を知りたいければ、表示されたテストメソッドをクリックしてください。ここからデバッグもできます。


## ステップ 5: 結果を分析する

TestCase クラスの assert: メソッドは、真偽値の引数を取ります。通常はテスト対象となる式を指定します。引数が真ならテストに成功、偽なら失敗です。

テストの実行結果には3通りあります。すべてのテストに成功する場合と、テストに失敗する場合と、テストの実行中にエラーが発生してテストが実行できない場合です。すべてのテストに成功すると、テストランナーの右上のバーが緑に変わります。テストに失敗するとバーが黄色に変わり、中段のペインに失敗したテストが表示されます<sup>5</sup>。最後が、テストの実行中にエラー(例外)が発

<sup>5</sup>訳注: PBE.image では、エラーの発生とは無関係に、失敗したテストがあった場合必ずバーが黄色になります。

生ずる場合です。例えば *message not understood* や *index out of bounds* と書いたエラーが発生する可能性があります。エラーが発生するとバーが赤に変わり、エラーの発生したテストが下段のペインに表示されます。

 エラーが発生したり失敗したりするようにテストを変更してみましょう。

## 7.5 SUnit クックブック


この節では、SUnit のより詳しい使い方を解説します。JUnit<sup>6</sup> などの他のテストフレームワークの経験があるなら、SUnit は親しみやすいでしょう。どのフレームワークも SUnit にルーツがあります。通常 SUnit では GUI でテストを実行しますが、GUI を使いたくない場合もあるでしょう。

### 様々なアサーション

`assert:` や `deny:` の他にもアサーション用のメソッドがあります。

一つ目は `assert:description:` と `deny:description:` です。二つ目の引数には、テストが失敗した理由を示す文字列を指定します。失敗の原因がわかりにくい場合に使うといいでしょう。7.7 節でこれらのメソッドについて触れます。

次は `should:raise:` と `shouldnt:raise:` です。これらは、テストの実行時に適切な例外が発生するかどうかをテストします。例えば `self should: aBlock raise: anException` を実行した場合、`aBlock` の実行中に例外 `anException` が発生すればテストは成功です。Method 7.6 に `should:raise:` を使った例を示します。

 このテストを実行してみましょう。

`should:` と `shouldnt:` の最初の引数は、評価する式を含むブロックです。

#### Method 7.6: エラーの発生をテストする

```
ExampleSetTest»testIllegal
```

```
self should: [empty at: 5] raise: Error.
self should: [empty at: 5 put: #zork] raise: Error
```

SUnit は移植性が確保されていて、どの Smalltalk の処理系でも使えます。SUnit の開発者は、処理系依存のコードを括り出すことで移植性を確保しました。例えば `TestResult class»error` クラスメソッドは、処理系によるシステムエラーのクラスの違いを吸収します。これを活かして Method 7.6 のコードを次のように書き換えると、同じテストを Smalltalk のあらゆる処理系で動かすことができます。

<sup>6</sup><http://junit.org>

## Method 7.7: 移植性の高いエラー処理

```
ExampleSetTest»testIllegal
```

```
self should: [empty at: 5] raise: TestResult error.
```

```
self should: [empty at: 5 put: #zork] raise: TestResult error
```

 試してみましょう。

## テストを一つだけ実行する

通常はテストランナーを使ってテストを実行しますが、もしテストランナーを `open ...` メニューから開くのが好みであれば、`TestRunner open` を `print it` することもできますね。

テストを一つだけ実行したい場合には、次のようにします。

```
ExampleSetTest run: #testRemove → 1 run, 1 passed, 0 failed, 0 errors
```

## 一つのテストクラスに定義されたすべてのテストを実行する

`TestCase` のサブクラスに `suite` メッセージを送信すると、名前が「test」で始まるメソッドを集めてテストスイートを組み立てます。このテストスイートに `run` メッセージを送信してテストを実行します。次に例を示します。

```
ExampleSetTest suite run → 5 run, 5 passed, 0 failed, 0 errors
```

## 必ず `TestCase` クラスを継承しなければならないのでしょうか？

`TestCase` クラスを継承しなくても `TestSuite` の要素として使うことはできます。ただし、自分で `TestCase` クラスの主要なメソッド (`assert`: など) を実装したり、テストスイートを自分で作ったりする必要が出てきます。`TestCase` クラスを継承することをお勧めします。フレームワークが既にあるのですから、それを使いましょう。

## 7.6 SUnit フレームワーク

SUnit は次の四つのクラス: `TestCase`、`TestSuite`、`TestResult`、`TestResource` (図 7.3) から構成されています。テストリソースは SUnit 3.1 で導入された概念で、セットアップのコストが高く、かつ一連のテスト全体で使われるようなリソースに使われます。`TestResource` の `setUp` メソッドは、`TestCase»setUp` と異なり、一連のテストの前に一度だけ実行されます。`TestCase»setUp` は各テストの前に実行されます。

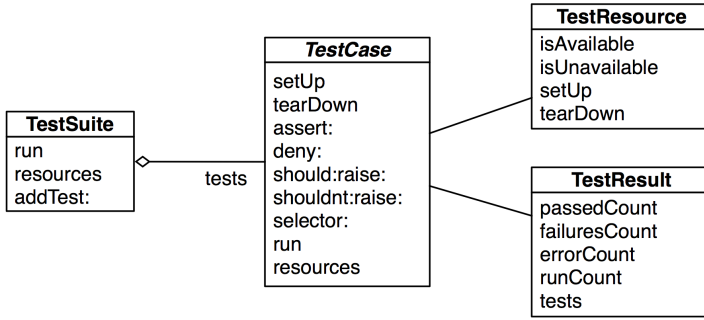


Figure 7.3: SUnit を構成する四つのクラス

## TestCase

TestCase は抽象クラスです。TestCase を継承して具体的なテストクラスを定義します。テストメソッド 1 個につき 1 個、テストクラスのインスタンスが生成されます。インスタンスにはまず setUp が送信され、次にテストメソッド自身が実行され、最後に tearDown が送信されます。

テストのコンテキストは、テストクラスのインスタンス変数が保持します。コンテキストを初期化するには setUp メソッドをオーバーライドします。tearDown メソッドをオーバーライドすることもできます。tearDown はテストの終了後に呼ばれるので、setUp で割り当てられたオブジェクトをこのメソッドで開放することができます。

## TestSuite

TestSuite のインスタンス (テストスイート) は、テストや他のテストスイートの集合です。つまり、テストスイートは TestCase と TestSuite のインスタンスを要素として含みます。TestCase と TestSuite の間に継承関係はありませんが、どちらも同じプロトコルを実装しているのと同じように扱えます。例えば、どちらも run メッセージを理解します。実際のところこれは、TestSuite を枝、TestCase を葉とする Composite パターンの応用になっています — 詳しくは デザインパターン を参照してください<sup>7</sup>。

## TestResult

TestResult クラスはテストの実行結果を扱います。成功したテストの数、失敗したテストの数、エラーが発生したテストの数を記録します。

<sup>7</sup>Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison Wesley, 1995, ISBN 0-201-63361-2-(3).

## TestResource

一連のテストでは、各テストが互いに独立していることが重要です。一つのテストの失敗が他のテストに波及しないように、そしてテストの実行順序が結果に影響のないようにしなければなりません。setUp と tearDown がこの独立性を高めます。

しかし、テストのたびにコンテキストを設定しては時間がかかりすぎる場合があります。さらに、一連のテストが環境を壊さずにすむことが事前にわかっているのなら、一度作った環境を使い回せば十分です。例えば、一連のテストがデータベースに問い合わせを行うなら、事前にセットアップしておいたデータベースに接続するだけで準備できます。コンパイルされたコードを分析するテストなら、同様に事前にソースコードをコンパイルしておけば十分です。

こうした環境 — リソースをどこにキャッシュすれば、一連のテストで共有できるようになるでしょうか? テストクラスのインスタンス変数はキャッシュとして使えません。テストクラスのインスタンスは1回のテストごとに消滅してしまうからです。グローバル変数ならできますが、多用すると名前空間が汚染されてしまいますし、各テストとグローバル変数の関係がわかりにくくなります。必要なリソースを何らかのクラスのシングルトンオブジェクトに持たせるのが良い方法です。そのための抽象クラスとして、TestResource が用意されています。つまり TestResource のサブクラスのインスタンスをリソースとして使います(テストリソース)。具体的なテストリソースを得るには TestResource のサブクラスに current メッセージを送信します。するとサブクラスのシングルトンインスタンスが得られます。リソースの初期化と終了処理を行うには、setUp と tearDown をオーバーライドします。

もう一つ、どうにかしてリソースとテストスイートを関連づける必要があります。これにはまず、テストクラスで resources クラスメソッドをオーバーライドし、テストクラスで使うリソース<sup>8</sup>を返すようにします。TestSuite»resources は、テストスイートに含まれるすべての TestCase のリソースのクラスの和集合を返します。

例を示しましょう。TestResource のサブクラス MyTestResource を定義します。次に MyTestCase のクラスメソッド resources をオーバーライドし、このクラスのテストで使うリソースのクラスの配列を返すようにします。

### Class 7.8: TestResource のサブクラスの例

```
TestResource subclass: #MyTestResource
  instanceVariableNames: "

MyTestCase class»resources
  "リソースとテストクラスを関連づける"
  ↑{ MyTestResource }
```

<sup>8</sup>訳注: テストクラスで使うテストリソースのクラスのコレクション。

## 7.7 SUnit のより高度な機能

現在の SUnit は `TestResource` に加えて、アサーションの詳細、ロギング、失敗したテストの実行再開に対応しています。

### アサーションの詳細

`TestCase` のアサーションプロトコルには、アサーションの詳細を文字列 (`String`) で知らせるメソッドが含まれています。テストに失敗すると、テストランナーにアサーションの詳細が表示されます。もちろん詳細を表す文字列を動的に生成することもできます。

```
| e |  
e := 42.  
self assert: e = 23  
  description: 'expected 23, got ', e printString
```

関連する `TestCase` のメソッドを次に示します:

```
#assert:description:  
#deny:description:  
#should:description:  
#shouldnt:description:
```

### ロギング

前節で触れたアサーションの詳細は、`Transcript` やファイルストリームなどの `Stream` にも出力できます。出力の有無を設定するには、`TestCase»isLogging` をオーバーライドして真偽値を返すようにします。出力するなら真を、しないなら偽を指定します。出力先を指定するには `TestCase»failureLog` をオーバーライドし、出力したいストリームを返すようにします<sup>9</sup>。

### テストの失敗後も実行を続ける

SUnit では、テストの失敗後に実行を続けるかどうかも指定できます。Smalltalk の例外処理の仕組みを利用した非常に強力な機能です。どのように動くのでしょうか。まず次のテストコードを見てください。

```
aCollection do: [:each | self assert: each even]
```

このコードではコレクションの要素を順に調べていますが、偶数 (`even`) でない要素があるとすぐにテストが終わってしまいます。しかし通常はテストを継続

<sup>9</sup>訳注: `isLogging` のデフォルト値は `false`、`failureLog` のデフォルト値は `Transcript` です。



して、偶数ではない要素がいくつあるのか調べたり、どれが偶数でないのかを把握したりしたいでしょう。またログを残したいかもしれません。それには次のようにします<sup>10</sup>。

```
aCollection do:
  [:each |
  self
    assert: each even
    description: each printString , ' is not even'
    resumable: true]
```

これで、失敗した要素についてのメッセージを `failureLog` に出力できます。実行を続けても失敗回数は増えないので、たとえ 1 回のテストを実行するうち 10 回以上アサーションに失敗しても、失敗の総数は 1 回と数えられます<sup>11</sup>。これまで見てきた他のアサーション用のメソッドは、失敗後に実行を続けません。assert: p description: s は、assert: p description: s resumable: false と等価です。

## 7.8 SUnit の内部実装

SUnit の内部実装は、Smalltalk のフレームワークの良い事例でもあります。テストの実行過程を通して、内部実装の鍵となる側面を見てみましょう。

### 一つのテストを実行する

一つのテストを実行するには、(aTestClass selector: aSymbol) run を評価します。

TestCase»run メソッドは、テスト結果を集計するために TestResult のインスタンスを生成し、それを引数として自身に run: メッセージを送信します (図 7.4 を参照)。

#### Method 7.9: テスト を実行する

```
TestCase»run
  | result |
  result := TestResult new.
  self run: result.
  ↑result
```

TestCase»run: メソッドは、引数 (TestResult のインスタンス) に runCase: メッセージを送信します。

#### Method 7.10: TestCase を TestResult に渡す

```
TestCase»run: aResult
```

<sup>10</sup> 訳注: PBE.image ではうまくいきません。

<sup>11</sup> 訳注: 1 回の TestResult»runCase: (7.8 節) の実行で記録される失敗の回数は、たかだか 1 回です。

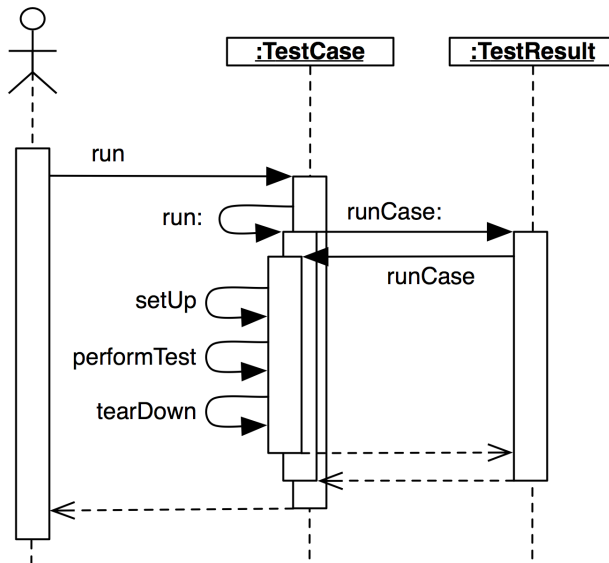


Figure 7.4: 一つのテストを実行する

```
aResult runCase: self
```

TestResult>runCase: メソッドは、引数として渡された TestCase に runCase メッセージを送信して、テストでエラーが発生した回数と失敗した回数、成功した回数を数えます。例外ハンドラを設定して、例外の発生とアサーションの失敗に備えます。

#### Method 7.11: TestCase のエラーと失敗を捕捉する

```
TestResult>runCase: aTestCase
| testCasePassed |
testCasePassed := true.
[[aTestCase runCase]
on: self class failure
do:
[:signal |
failures add: aTestCase.
testCasePassed := false.
signal return: false]]
on: self class error
do:
[:signal |
errors add: aTestCase.
testCasePassed := false.
signal return: false].
```

```
testCasePassed ifTrue: [passed add: aTestCase]
```

TestCase»runCase は、自身に setUp と tearDown を送信します。

Method 7.12: テンプレートメソッドとしての TestCase»runCase<sup>12</sup>

```
TestCase»runCase
|self setUp.
self performTest] ensure: [self tearDown]
```

## TestSuite の実行

関連する複数のテストを実行するには、TestSuite に run メッセージを送信します。TestCase クラスには、サブクラスの方法セレクタのコレクションからテストスイートを組み立てる機能があります。MyTestCase buildSuiteFromSelectors を評価すると、MyTestCase に定義したすべてのテストを含むテストスイートを返します。この処理の中心になるのは、以下のメソッドです。

Method 7.13: テストスイートを自動的に組み立てる

```
TestCase class»testSelectors
↑self selectors asSortedCollection asOrderedCollection select: [:each |
('test*' match: each) and: [each numArgs isZero]]
```

TestSuite»run は TestResult のインスタンスを生成し、すべてのテストリソースが問題なく使えるかどうかを検証します。続けて自身に run: を送信し、テストスイートの中のすべてのテストを実行します (TestSuite»run:.)。そしてテストの実行後にすべてのテストリソースを解放 (reset) します。

Method 7.14: テストスイートを実行する

```
TestSuite»run
| result |
result := TestResult new.
self resources do: [:res |
res isAvailable ifFalse: [↑res signalInitializationError]].
[self run: result] ensure: [self resources do: [:each | each reset]].
↑result
```

Method 7.15: TestResult を TestSuite に渡す

```
TestSuite»run: aResult
self tests do: [:each |
self changed: each.
each run: aResult].
```

<sup>12</sup>訳注: self performTest は、self が知っているテストメソッドを間接的に呼び出します (perform)。

`TestResource` クラスとそのサブクラスにはシングルトンインスタンスがあって、`current` クラスメソッドで取得できます。このインスタンスはテストリソースを保持しており、テストの終了後に `reset` されます。

`TestResource class»isAvailable` クラスメソッドを使えばテストリソースが使えるかどうか確認できますが、このとき必要に応じてテストリソースが再生成されます。再生成されたインスタンスには、`setUp` が送信されます。

#### Method 7.16: テストリソースが使えるかどうかの確認

```
TestResource class»isAvailable
  ↑self current notNil and: [self current isAvailable]
```

#### Method 7.17: テストリソースの生成

```
TestResource class»current
  current isNil ifTrue: [current := self new].
  ↑current
```

#### Method 7.18: テストリソースの初期化

```
TestResource»initialize
  super initialize.
  self setUp
```

## 7.9 テストのポイント

テストの仕組みは簡単ですが、良いテストを書くことは簡単ではありません。テストを設計する上でのいくつかのポイントを紹介します。

**Feathers** のユニットテストの法則。アジャイル開発コンサルタントであり「レガシーコード改善ガイド」などの著作でも知られる TBD (Michael Feathers) は、次のように書いています<sup>13</sup>。

次のテストはユニットテストではありません:

- データベースと対話したり、
- ネットワークに接続したり、
- ファイルシステムに触ったり、
- 他のユニットテストと並行して実行できなかったり、
- 特別な環境 (設定ファイルの編集が必要など) を準備しなければならない。

<sup>13</sup><http://www.artima.com/weblogs/viewpost.jsp?thread=126923> を参照。2005/9/9

このようなテストも悪くはありません。書く価値はありますし、ユニットテストの補助にもなります。しかしこれらのテストを本当のユニットテストとは分けておくことが大事です。そうすることで、コードを変更したら即刻走らせることができるテストを確保することができるのです。

決して「ユニットテストするのに時間がかかるからやらないですませたいな」と思ってしまうような状況を作ってははいけません。

**ユニットテスト vs. 承認テスト** ユニットテストは機能の一部を単体でテストし、機能に関するバグの特定を容易にします。各メソッドのユニットテストを失敗するまで可能な限り実行し、クラス単位で結果をまとめます。しかし、複雑な環境をセットアップする必要があるれば、より大きなアプリケーションの中のシナリオに沿ってテストを書く方が簡単です。このようなテストは承認テストまたは機能テストと呼ばれます。Feathersの法則を守らないテストは、承認テストとしては良いテストである場合があります。テストする機能に沿って承認テストをまとめましょう。例えばコンパイラを書く場合、その言語のステートメントごとの生成コードを検査する承認テストを書くことになるでしょう。たくさんのクラスがテストの実行に関与するかもしれません。そして、ファイルシステムにアクセスするので時間がかかるかもしれません。そのようなテストをSUnitで記述することはできますが、ほんのちょっとの変更のたびに実行したいとは思わないでしょう。ならば、そのようなテストは本当のユニットテストとは分けておくべきなのです。

**Black** のテストの法則。システムのテストごとに、テストを行うことによってシステムのどの性質についての確信が深まるのか、明らかにしなければなりません。もちろん重要な性質は必ずテストしなければなりません。その一方、システムの性質についての確信を深めないようなテストはしてはいけません。そのようなテストはシステムに何ら価値を付加しません。例えば、同じ性質に関するテストを複数書いてもいいことはありません。かえって二つのデメリットがあります。一つは、テストからクラスの振る舞いが読み取りにくくなります。もう一つは、コードにいくつバグが残っているのか見積りにくくなります。たった一つのバグが複数のテストを失敗させるからです。テストを書くときは、システムのどの性質についての確信を得るためなのか、常に考えましょう。

## 7.10 まとめ

この章では、なぜテストが将来のコードへの重要な投資であるのかを説明しました。Setクラスのテストを定義する方法について、一つ一つ手順を追って説明しました。SUnitフレームワークの全体像を示すため、TestCase、TestResult、TestSuite、TestResourceの四つのクラスを紹介しました。テストとテストスイートの実行過程の深層を見てきました。

- ユニットテストの効果を最大限に引き出すポイント: 高速、繰り返し可能、人と直接やりとりしない、一つの機能のみをテストする。
- テストクラス名は「被テストクラス名 + Test」にします (MyClass のテストクラスなら MyClassTest)。テストクラスは TestCase のサブクラスとして定義しましょう。
- setUp メソッドでテストデータを初期化します。
- 各テストメソッド名は「test」で始めます。
- TestCase の assert: や deny: などのメソッドでアサーションを定義します。
- SUnit のテストランナーを使ってテストを実行します。

# Chapter 8

## 基本的なクラス

Smalltalk の魔法の大部分は、言語そのものではなくクラスライブラリにあります。Smalltalk で効率的にプログラミングするには、クラスライブラリがどのように言語や環境を支えているのかを学ばなければなりません。クラスライブラリはすべてが Smalltalk で書かれていて、パッケージ (クラス定義を含んでいる必要はありません) を使えば新しい機能をクラスに追加できるので、簡単に拡張することができます。

この章の目的は、Pharo のクラスライブラリ全体について詳細で退屈な話をするものではありません。代わりにここでは、効率よくプログラムするために利用したりオーバーライドしたりすべき、主要なクラスやメソッドを示します。この章では、ほぼすべてのアプリケーションで必要になる、以下の基本的なクラスを取り上げます: Object、Number とそのサブクラス、Character、String、Symbol、Boolean。

### 8.1 Object

何をするにも、Object は継承階層のルートです。実際は、Pharo では ProtoObject が階層の真のルートです。ProtoObject は、オブジェクトのふりをすることができる最小の実体を定義するのに使用されます。しかし、今このことは気にしなくていいでしょう。

Object は *Kernel-Objects* カテゴリの中にあります。驚いたことに、そこには (拡張も含めると) 400 ものメソッドがあります。これはつまり、新しいクラスを定義すると、あなたがそれらについて知っていようといまいと、すべてに自動的に 400 個のメソッドが提供されるということです。ただ、削除されるべきメソッドもいくつかあって、Pharo の新しいバージョンではそれらの不要なメソッドのいくつかはおそらく削除されることを覚えておきましょう。

Object のクラスコメントには次のように書かれています:

Object はクラス階層内のほぼすべてのクラスのルートクラスである。例外は ProtoObject (Object のスーパークラス) とその (直接の) サブクラスである。Object クラスは、通常のオブジェクトのアクセス、コピー、比較、エラー処理、メッセージ送信、リフレクションなどのデフォルトの振る舞いを提供する。また、すべてのオブジェクトが反応する必要があるユーティリティメッセージもここで定義される。Object はインスタンス変数を持っておらず、追加すべきでもない。これは、Object を継承するクラスには特殊な実装を持つもの (例えば SmallInteger や UndefinedObject) がいくつかあって、これらや VM が、ある標準的なクラスの構造とレイアウトを知っていてそれに依存しているためである。

本節では、Object のインスタンスボタン側のメソッドカテゴリ (プロトコル) をブラウズしていきます。それに伴い、Object が提供するいくつかの主だった振る舞いが理解できるようになっていくでしょう。

## printing プロトコル

Smalltalk のすべてのオブジェクトは、自分自身の表示用の形式を返すことができます。ワークスペースで好きな式を選び、`print it` メニューを選択してください: `print it` は式を実行し、返されたオブジェクトに「あなたを表示してください」と頼みます。実際には、返されたオブジェクトに `printString` メッセージを送信します。`printString` はテンプレートメソッドで、その中で `printOn:` メッセージをレシーバに送信します。`printOn:` はフックメソッドで、特殊化することができます。

おそらく Object`»printOn:` は、最も頻繁にオーバーライドされるメソッドの一つになるでしょう。このメソッドは引数に Stream (ストリーム) を受け取り、そこにオブジェクトの String 表現を書き込みます。`printOn:` のデフォルト実装は、単に「a」または「an」の後にクラス名を書き込むだけです。Object`»printString` は、ストリームに書き込まれたものを String にして返します。

例えば、Browser クラスは `printOn:` メソッドを再定義していないので、`printString` メッセージを Browser のインスタンスに送信すると Object で定義されたメソッドが実行されます。

```
Browser new printString → 'a Browser'
```

Color クラスでは `printOn:` を特殊化した例が見られます。Color`»printOn:` は、「クラス名 色名 (例えば Color red)」と言った色を表すメッセージ式をプリントします。



Method 8.1: *printOn*: の再定義。

```
Color»printOn: aStream
| name |
(name := self name) ifNotNil:
[↑ aStream
 nextPutAll: 'Color ';
 nextPutAll: name ].
self storeOn: aStream
```

```
Color red printString → 'Color red'
```

*printOn*: メッセージは *storeOn*: とは異なる、ということに注意してください。*storeOn*: メッセージは、レシーバを再生成するのに使える式を引数 (ストリーム) に流します。そしてその式は、ストリームが *readFrom*: メッセージによって読み込まれるときに評価されます。一方 *printOn*: は、単にレシーバのテキストバージョンを返すだけです。もちろんそのテキスト表現が、レシーバを表す自己評価型の式になっていることもあり得ます。

表現と自己評価型表現という単語について。関数型プログラミングでは、式は実行されると値を返します。一方 Smalltalk では、メッセージ (式) はオブジェクト (値) を返します。オブジェクトの中には、値と自分自身が同じという好ましい性質を持つものがあります。例えば、*true* オブジェクトの値はそれ自身、すなわち *true* オブジェクトです。このようなオブジェクトを自己評価型オブジェクトと呼びます。ワークスペースでオブジェクトを *print it* すると、オブジェクトの値の表示用バージョンを見ることができます。次のものが自己評価型表現の例です。

```
true      → true
3@4       → 3@4
$a        → $a
#(1 2 3)  → #(1 2 3)
Color red → Color red
```

配列のようなオブジェクトは、そこに含まれるオブジェクトによって自己評価型であったりそうではなかったりするに注意してください。例えば、真偽値の配列は自己評価型ですが、*person* オブジェクトの配列はそうではありません。次の例は、動的配列が自己評価型になるのは、要素も自己評価型の場合だけであることを示しています:

```
{10@10. 100@100} → {10@10. 100@100}
{Browser new . 100@100} → an Array(a Browser 100@100)
```

リテラル配列はリテラルだけを保持できることに注意してください。そのため次の配列は、二つの座標を保持するのではなく6個のリテラル要素を持つこととなります。

```
 #(10@10 100@100)  →  #(10 #@ 10 100 #@ 100)
```

printOn: メソッドの多くは、自己評価型の振る舞いを実装するように特殊化されています。例えば Point»printOn: や Interval»printOn: の実装は、自己評価型です。

#### Method 8.2: Point の自己評価<sup>1</sup>

```
Point»printOn: aStream
  "レシーバ(の表現形式)を中置記法で aStream に書き込む"
  x printOn: aStream.
  aStream nextPut: $@.
  y printOn: aStream
```

#### Method 8.3: Interval の自己評価

```
Interval»printOn: aStream
  aStream nextPut: $(;
    print: start;
    nextPutAll: ' to: ';
    print: stop.
  step ~ = 1 ifTrue: [aStream nextPutAll: ' by: '; print: step].
  aStream nextPut: $)
```

```
1 to: 10  →  (1 to: 10)  "Interval オブジェクトは自己評価型"
```

## comparing プロトコル (同一性と同値性)

Smalltalk では、= メッセージはオブジェクトの同値性をテストします (すなわち二つのオブジェクトが同じ値を持つかどうか)。一方、== はオブジェクトの同一性をテストします (すなわち二つの式が同じオブジェクトを示すかどうか)。

オブジェクトの同値性テストのデフォルト実装では、オブジェクトの同一性をテストしています:

#### Method 8.4: オブジェクトの同値性

```
Object»= anObject
  "レシーバと引数が同じオブジェクトを示すかどうか答える。
  もし = をサブクラスで再定義するなら、hash メッセージの再定義も検討すること。"
  ↑ self == anObject
```

これは頻繁にオーバーライドすることになるメソッドです。Complex (複素数) の場合を見てみましょう<sup>2</sup>:

<sup>1</sup> 訳注: Pharo 1.3 ではコードが変更されています。

<sup>2</sup> 訳注: Complex クラスは Pharo 1.3 に存在しません。

```
(1 + 2 i) = (1 + 2 i)  → true  "同じ値"
(1 + 2 i) == (1 + 2 i) → false "だけど違うオブジェクト"
```

このように動作するのは、Complex が = を次のようにオーバーライドしているからです:

#### Method 8.5: 複素数の同値性

```
Complex>>= anObject
anObject isComplex
  ifTrue: [↑ (real = anObject real) & (imaginary = anObject imaginary)]
  ifFalse: [↑ anObject adaptToComplex: self andSend: #=]
```

Object>>~= は、単に = を反転しているだけですが、通常は変更すべきではありません。

```
(1 + 2 i) ~= (1 + 4 i)  → true
```

もし = をオーバーライドしたら、hash をオーバーライドしなければなりません。クラスのインスタンスが Dictionary のキーとして使用された場合に、同値であるとみなされるインスタンスは、ハッシュ値が同じになるようにしておく必要があります:

#### Method 8.6: 複素数の hash は再実装されなければならない

```
Complex>>hash
  "= を実装したので hash も再実装する。"
  ↑ real hash bitXor: imaginary hash.
```

= と hash は同時にオーバーライドしなければなりません、== は決してオーバーライドしてはいけません (オブジェクトの同一性に関する意味論は、すべてのクラスで同じです)。== は ProtoObject のプリミティブメソッドです。

Pharo には、他の Smalltalk と比べていくつか奇妙な振る舞いがあることに注意してください: 例えばシンボルと文字列は同値とみなされることがあります (これは機能ではなくバグではないかと著者らは考えています)。

```
#'lulu' = 'lulu'  → true
'lulu' = #'lulu' → true
```

## class membership プロトコル

いくつかのメソッドを使って、オブジェクトのクラスに関する問い合わせをすることができます。

**class.** 任意のオブジェクトに `class` メッセージを送信して、それが属するクラスを答えさせることができます。

```
1 class → SmallInteger
```

反対に、オブジェクトが特定のクラスのインスタンスであるかどうかを尋ねることもできます:

```
1 isMemberOf: SmallInteger → true "まさにこのクラスに属していなければ  
ならない"  
1 isMemberOf: Integer → false  
1 isMemberOf: Number → false  
1 isMemberOf: Object → false
```

Smalltalk のクラスライブラリは Smalltalk 自身によって書かれているので、スーパークラスとクラスメッセージを正しく組み合わせて、クラスライブラリの構造を自在にたどることができます (第 13 章を参照)。

**isKindOf:** `Object`»`isKindOf:` は、レシーバのクラスが引数のクラスと等しいか、またはそのサブクラスであるかを答えます。

```
1 isKindOf: SmallInteger → true  
1 isKindOf: Integer → true  
1 isKindOf: Number → true  
1 isKindOf: Object → true  
1 isKindOf: String → false  
  
1/3 isKindOf: Number → true  
1/3 isKindOf: Integer → false
```

1/3 は `Fraction` で `Number` でもあります。これは、`Number` クラスが `Fraction` クラスのスーパークラスだからです。しかし 1/3 は `Integer` ではありません。

**respondsTo:** `Object`»`respondsTo:` は、レシーバがメッセージセレクトア (`respondsTo:` のシンボル引数) を理解するかどうか答えます。

```
1 respondsTo: #, → false
```

通常は、オブジェクトにクラスを問い合わせたり、オブジェクトがどのようなメッセージに応えられるかを聞いたりするのは良いやり方ではありません。オブジェクトのクラスに基づいて判断する代わりに、オブジェクトにメッセージだけを送信し、どう振る舞うかはオブジェクトの決定に任せるべきです。

## copying プロトコル

オブジェクトのコピーには、いくつかちょっとした問題があります。インスタンス変数は参照によってアクセスされるので、あるオブジェクトのシャローコピーは元のオブジェクトとインスタンス変数の参照を共有することになります。

```
a1 := { { 'harry' } }.
a1  →  #(#('harry'))
a2 := a1 shallowCopy.
a2  →  #(#('harry'))
(a1 at: 1) at: 1 put: 'sally'.
a1  →  #(#('sally'))
a2  →  #(#('sally'))  "内側の配列が共有されている"
```

Object>shallowCopy は、オブジェクトのシャローコピーを作成するためのプリミティブメソッドです。a2 は a1 の単なるシャローコピーなので、二つの配列はネストされた配列の参照を共有しています。

Object>shallowCopy は Object>copy の「公開インターフェース」で、インスタンスに一意性があればオーバーライドする必要があります。例えば Boolean、Character、SmallInteger、UndefinedObject などのクラスがそれに当たります。

Object>copyTwoLevel は、単純なシャローコピーではうまくいかないときに、見たままのことは行ってくれます:

```
a1 := { { 'harry' } }.
a2 := a1 copyTwoLevel.
(a1 at: 1) at: 1 put: 'sally'.
a1  →  #(#('sally'))
a2  →  #(#('harry'))  "状態は完全に独立している"
```

Object>deepCopy は、任意の深さまでオブジェクトをディープコピーします。

```
a1 := { { { 'harry' } } }.
a2 := a1 deepCopy.
(a1 at: 1) at: 1 put: 'sally'.
a1  →  #(#('sally'))
a2  →  #(#('#('harry'))))
```

deepCopy の問題は、相互再帰的な構造に対して使うと終了しなくなることです:

```
a1 := { 'harry' }.
a2 := { a1 }.
a1 at: 1 put: a2.
a1 deepCopy  →  ... does not terminate!
```

正しく動くように deepCopy をオーバーライドすることもできますが、Object>>copy がもっと良い解決策を提供してくれます:

## Method 8.7: オブジェクトをコピーするテンプレートメソッド

Object»copy

“レシーバと同じようなもう一つのインスタンスを返す。  
サブクラスは通常、postCopy をオーバーライドする。  
通常は shallowCopy はオーバーライドしない。”  
↑self shallowCopy postCopy

共有すべきではないインスタンス変数をコピーするには、postCopy をオーバーライドしましょう。そして、postCopy の中では常に super postCopy しましょう。

## debugging プロトコル

デバッグする上で最も重要なメソッドは halt です<sup>3</sup>。メソッドにブレークポイントを設定するには、メソッド本文の好きな場所に self halt というメッセージ送信を挿入するだけでかまいません。このメッセージが送信されると、プログラムのその位置で実行が中断され、デバッガが起動します (デバッガの詳細については第 6 章を参照してください)。

次に重要なメソッドは assert: です<sup>4</sup>。これは引数としてブロックを取ります。ブロックが true を返すと実行は継続されます。そうでなければ AssertionError 例外が発生します。この例外がキャッチされなければ、例外の発生した位置でデバッガが開きます。assert: は、契約による設計をサポートするのに特に便利です。オブジェクトのパブリックなメソッドの重要な事前条件をチェックする、というのが最もよくある利用方法でしょう。Stack»pop は、次のように簡単に実装できます:

## Method 8.8: 事前条件のチェック

Stack»pop

“最初の要素を返し、スタックからそれを削除する。”  
self assert: [ self isEmpty not ].  
↑self linkedList removeFirst element

Object»assert: と TestCase»assert: を混同しないでください。後者は SUnit テスティングフレームワーク (第 7 章を参照) の中で出てきます。前者は引数としてブロックを期待する<sup>5</sup>のに対し、後者は Boolean を期待します。共にデバッグのために有用ですが、それぞれ目的は大きく異なります。

## error handling プロトコル

このプロトコルには、ランタイムエラーを通知するのに便利なメソッドがいくつか含まれます。

<sup>3</sup>訳注: halt は、実際には error handling プロトコルに分類されています。

<sup>4</sup>訳注: assert: は、実際には error handling プロトコルに分類されています。

<sup>5</sup>実際には value メッセージを理解するあらゆる引数 — Boolean を含む — を受け取れます。

`self deprecated: anExplanationString` を送信すると、そのメソッドは今後は奨励されないということが通知できます (プリファレンス・ブラウザの `debug` プロトコルで `deprecation` をオンにしているとき)。今後奨励される選択肢は、String 引数の形で提示しましょう。

```
1 dolfNotNil: [ :arg | arg printString, ' is not nil' ]
   → SmallInteger(Object)»dolfNotNil: has been deprecated. use ifNotNilDo:
```

メソッド探索に失敗すると `doesNotUnderstand:` が送信されます。`doesNotUnderstand:` のデフォルト実装、すなわち `Object»doesNotUnderstand:` では、その位置でデバッガが起動します。`doesNotUnderstand:` をオーバーライドして他の振る舞いをさせると便利かもしれません。

`Object»error` と `Object»error:` は、例外を発生させるのに使える汎用的なメソッドです (ただし独自に定義した例外を発生させる方が、一般的には望ましいでしょう。自分のコードとカーネルクラスのどちらで発生したエラーなのかが区別できるからです)。

`Smalltalk` の抽象メソッドは、規約によりメソッドの本文に `self subclassResponsibility` と書いて実装します。従って抽象クラスが間違っってインスタンス化され抽象メソッドが呼び出されたとしても、`Object»subclassResponsibility` が評価されるだけです。

#### Method 8.9: メソッドが抽象メソッドであることを通知する<sup>6</sup>

```
Object»subclassResponsibility
"このメッセージはサブクラスの振る舞いに関する仕組みを支えています。
サブクラスはこのメッセージを実装すべきだと忠告します。"
self error: 'My subclass should have overridden ', thisContext sender selector
printString
```

`Magnitude` と `Number` と `Boolean` が、抽象クラスの古典的な例です。この章の少し先で説明します。

```
Number new + 1 → Error: My subclass should have overridden #+
```

規約により、サブクラスが継承すべきではないメソッドを通知するには、サブクラス側で `self shouldNotImplement` を送信します。これは、一般的にはクラス階層の設計が何かうまくいっていないことのサインです。しかし単一継承の制限があるため、どうしてもこのような回避策を取らざるを得ない場合もあります。

典型的な例が `Dictionary»remove:` です。ここでは、`Collection»remove:` を継承しないで `shouldNotImplement` フラグを立てています (`Dictionary` は代わりに `removeKey:` を用意しています)。

<sup>6</sup>訳注: この例は Pharo 1.3 では変更されています。

## testing プロトコル

*testing* プロトコルに分類されるメソッド (*testing* メソッド) は、SUnit のテストとは何の関係もありません! *testing* メソッドは、レシーバの状態を聞かれて Boolean を答えるメソッドです。

たくさんの *testing* メソッドが、Object によって提供されています。isComplex は既に見ました。他にも isArray、isBoolean、isBlock、isCollection などがあります。一般的には、これらのメソッドの使用は避けるべきです。オブジェクトに自身のクラスを問い合わせるのは、カプセル化を破ることだからです。オブジェクトのクラスをテストする代わりに要求だけをオブジェクトに送信し、後の処理はオブジェクトに任せるべきです。

とは言っても、これらの *testing* メソッドのいくつかは間違いなく有用です。最も有用なのは、おそらく ProtoObject»isNil と Object»notNil でしょう (Null Object<sup>7</sup> デザインパターンを使ってこれらのメソッドを不要にすることもできますが)。

## initialize-release プロトコル

Object ではなく ProtoObject にある最後のキーメソッドは、initialize です。

### Method 8.10: 空のフックメソッドとしての initialize

ProtoObject»initialize

”サブクラスはこのメソッドを再定義して、インスタンス生成時に初期化を実行する”

これが重要なのは、Pharo ではすべてのクラスの new メソッドのデフォルト実装が、新しく生成されたインスタンスに initialize を送信するように書かれているからです。

### Method 8.11: クラス側のテンプレートメソッドとしての new

Behavior»new

”添字付きの変数を持たないレシーバ(クラス)の初期化された新しいインスタンスを返す。  
クラスが添字付きの場合は失敗する。”  
↑self basicNew initialize

これは、initialize フックメソッドをオーバーライドしさえすれば、あなたのクラスの新しいインスタンスが自動的に初期化されるということを意味しています。通常 initialize メソッドは、継承されたすべてのインスタンス変数についてクラスの不変条件を確立するために、super initialize を実行すべきです (これは他の Smalltalk では標準の振る舞いではないことに注意してください)。

<sup>7</sup>Bobby Woolf, Null Object. In Robert Martin, Dirk Riehle and Frank Buschmann, editors, Pattern Languages of Program Design 3. Addison Wesley, 1998.



## 8.2 Number

驚くべきことに、Smalltalk では数値はプリミティブなデータではなく、本物のオブジェクトです。もちろん数値はバーチャルマシンで効率的に実装されていますが、Number の階層は、Smalltalk のクラス階層の他の部分と同じく、完全にアクセス可能で拡張可能です。

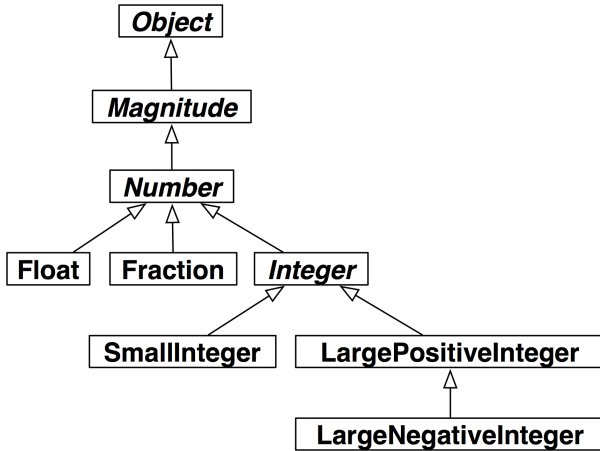


Figure 8.1: Number のクラス階層

Number は *Kernel-Numbers* カテゴリの中にあります。この階層の抽象的なルートは、Magnitude です。このクラスは、比較演算子をサポートするあらゆる種類のクラスを表します。Number は、様々な算術演算子やその他の演算子を追加します。これらの演算子のほとんどが抽象メソッドです。Float と Fraction は、それぞれ浮動小数点数と分数を表します。Integer も抽象クラスです。抽象クラスの Integer は、従って、具象サブクラスの SmallInteger、LargePositiveInteger、LargeNegativeInteger とは区別されます。ただし、必要に応じて自動的に値が変換されるので、ほとんどのユーザは三つの Integer クラスの違いを気にする必要はありません。

### Magnitude

Magnitude は Number の親であるだけでなく、Character、Duration、Timespan などのような比較演算をサポートするその他のクラスの親クラスでもあります (Complex は比較できません。従って Complex は Number を継承していません<sup>8</sup>)。

< と = は抽象メソッドです。それ以外の演算子は汎用的な形で定義されています。例えば:

<sup>8</sup>訳注: Complex クラスは Pharo 1.3 に存在しません。

## Method 8.12: 比較用の抽象メソッド

```
Magnitude» < aMagnitude
"レシーバが引数より小さいかどうか答える"
↑self subclassResponsibility
```

```
Magnitude» > aMagnitude
"レシーバが引数より大きいかどうか答える"
↑aMagnitude < self
```

## Number

同様に `Number` では、`+`、`-`、`*`、`/` などが抽象メソッドとして定義されており、それ以外のすべての算術演算子は汎用的な形で定義されています。

すべての `Number` オブジェクトは、`asFloat` や `asInteger` のような様々な変換演算子をサポートします。また、それ以外にもたくさんのショートカットコンストラクタメソッド<sup>9</sup>つまり、`Number` を実部がゼロの `Complex`<sup>10</sup> のインスタンスに変換する `i` や、`Number` から `Duration` オブジェクトを生成する `hour`、`day`、`week` などがあります。

`Numbers` は、`sin`、`log`、`raiseTo:`、`squared`、`sqrt` などのよくある数学関数を直接サポートします。

`Number»printOn:` は、抽象メソッドの `Number»printOn:base:` を使って基数 10 の場合として実装されています。

`testing` メソッドとしては、`even`、`odd`、`positive`、`negative` などがあります。`Number` が `isNumber` をオーバーライドしているのは驚くほどのことではありません。もっと興味深い点として、`isInfinite` は `false` を返すように定義されています。

切り捨て系のメソッドとしては、`floor`、`ceiling`、`integerPart`、`fractionPart` などがあります。

<code>1 + 2.5</code>	→	<code>3.5</code>	"二つの数の和"
<code>3.4 * 5</code>	→	<code>17.0</code>	"二つの数の積"
<code>8 / 2</code>	→	<code>4</code>	"二つの数の商"
<code>10 - 8.3</code>	→	<code>1.7</code>	"二つの数の差 <sup>11</sup> "
<code>12 = 11</code>	→	<code>false</code>	"二つの数の同値性"
<code>12 ~= 11</code>	→	<code>true</code>	"二つの数が異なるかどうかをテスト"
<code>12 &gt; 9</code>	→	<code>true</code>	"より大きい"
<code>12 &gt;= 10</code>	→	<code>true</code>	"以上"
<code>12 &lt; 10</code>	→	<code>false</code>	"より小さい"
<code>100@10</code>	→	<code>100@10</code>	"座標の生成"

<sup>9</sup> 訳注: 便利コンストラクタとも呼ばれます。


<sup>10</sup> 訳注: `Complex` クラスは `Pharo 1.3` に存在しません。

<sup>11</sup> 訳注: `Pharo 1.3` では表示が異なる場合があります。

次の例は Smalltalk では驚くほど正しく動作します:

```
1000 factorial / 999 factorial  →  1000
```

1000 factorial は、他の多くの言語では計算するのが非常に難しいのですが、実際に計算できていることに注目してください。これは、数値に関する「自動型変換と正確な演算」の非常に良い例になっています。

 1000 factorial の結果を表示してみましょう。計算よりも表示に長い時間がかかります!

## Float

Float は、Number の抽象メソッドを浮動小数点数用に実装します。

さらに興味深いことに、Float class (すなわち Float のクラスボタン側) は、次の定数を返すメソッドを提供しています: e、infinity、nan、pi。

```
Float pi           →  3.141592653589793
Float infinity     →  Infinity
Float infinity isInfinite →  true
```

## Fraction

Fraction は、分子と分母に当たるインスタンス変数によって表されます。分子と分母は Integer であることが期待されます。Fraction は、通常は Integer の割り算によって生成されます (Fraction>>numerator:denominator: というコンストラクタメソッドによってではなく)。

```
6/8           →  (3/4)
(6/8) class   →  Fraction
```

Fraction と Integer または Fraction と他の Fraction を掛け合わせると、Integer になることがあります:

```
6/8 * 4  →  3
```

## Integer

Integer は三つの具象整数クラスの親に当たる抽象クラスです。Integer は、多くの Number の抽象メソッドの具体的な実装を提供するだけでなく、factorial、atRandom、isPrime、gcd: などの整数特有のメソッドやその他多くのメソッドを追加します。

`SmallInteger` は、インスタンスがコンパクトに表現される点で特殊です — `SmallInteger` は、値を参照として格納する代わりに、通常は参照を保持するために使われるビット領域を使って直接値を表現します。この領域の最初のビットが、オブジェクトが `SmallInteger` かどうかを示しています。

`minVal` と `maxVal` というクラスメソッドで、`SmallInteger` の範囲がわかります:

```
SmallInteger maxVal = ((2 raisedTo: 30) - 1)    → true
SmallInteger minVal = (2 raisedTo: 30) negated  → true
```

`SmallInteger` がその範囲に収まらなくなると、必要に応じて自動的に `LargePositiveInteger` または `LargeNegativeInteger` に変換されます:

```
(SmallInteger maxVal + 1) class → LargePositiveInteger
(SmallInteger minVal - 1) class → LargeNegativeInteger
```

大きな整数も同様に、適切なタイミングで小さな整数に変換されます。

多くのプログラミング言語同様、整数は繰り返しを指定するのに便利なことがあります。ブロックを繰り返し評価することに特化した `timesRepeat:` というメソッドもあります。第3章で既に同様の例を見ました:

```
n := 2.
3 timesRepeat: [ n := n*n ].
n → 256
```

## 8.3 Character

`Character` は、*Collections-Strings* カテゴリ内で `Magnitude` のサブクラスとして定義されています。表示できる文字は、Pharo では `$(char)` と表します。例えば:

```
$(a) < $(b) → true
```

表示できない文字は、様々なクラスメソッドを使って得ることができます。`Character class>>value:` は、Unicode (または ASCII) の整数値を引数として受け取って、対応する文字を返します。*accessing untypeable characters* プロトコルには、`backspace`、`cr`、`escape`、`euro`、`space`、`tab` などのたくさんの便利コンストラクタ<sup>12</sup>が含まれています。

```
Character space = (Character value: Character space asciiValue) → true
```

`printOn:` メソッドは十分に賢く、文字を表現する三つの方法のうちどれが最も適切かを知っています:

<sup>12</sup>訳注: 正確にはこれらはコンストラクタではありません。

```
Character value: 1   → Character home
Character value: 2   → Character value: 2
Character value: 32  → Character space
Character value: 97  → $a
```

Character には、様々な便利な *testing* メソッドが組み込まれています: `isAlphaNumeric`、`isCharacter`、`isDigit`、`isLowercase`、`isVowel` など。

Character を、その文字だけを含む文字列に変換するには、`asString` を送信します。この場合 `asString` と `printString` とでは結果が異なります。

```
$a asString   → 'a'
$a           → $a
$a printString → '$a'
```

8 ビットの範囲内の Character はそれぞれ一意のインスタンスで、クラス変数である `CharacterTable` の中に保持されています:

```
(Character value: 97) == $a → true
```

しかし、0 から 255 の範囲外の Character は一意ではありません:

```
Character characterTable size → 256
(Character value: 500) == (Character value: 500) → false
```

## 8.4 String

String クラスも *Collections-Strings* カテゴリで定義されています。String は、Character だけを保持する添字付きの Collection です。

実際は、String は抽象クラスです。本当は Pharo の String は、具象クラスである `ByteString` か `WideString` のインスタンスです。

```
'hello world' class → ByteString
```

String のもう一つの重要なサブクラスは、`Symbol` です。主な違いは、`Symbol` のインスタンスは一つの値につきたった一つしか存在しないということです (これはしばしば「一意インスタンス性」と呼ばれます)。一方、別々に生成されたたまたま同じ文字の並びを持つ String は、異なるオブジェクトであることもよくあります。

```
'hel','lo' == 'hello' → false
```

```
('hel','lo') asSymbol == #hello → true
```

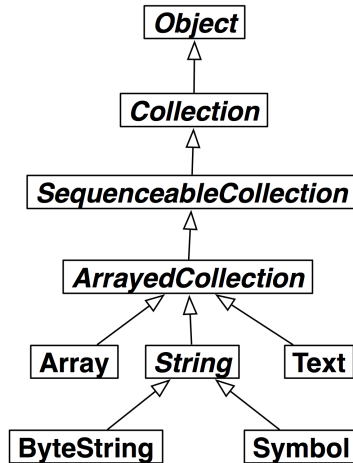


Figure 8.2: String のクラス階層

もう一つの重要な違いは、String が変更可能 (mutable) なものに対して Symbol は変更不能 (immutable) なことです。

```
'hello' at: 2 put: $u; yourself → 'hullo'
```

```
#hello at: 2 put: $u → エラー
```

見落としがちなのは、文字列はコレクションなので他のコレクションと同じメッセージを理解することです:

```
#hello indexOf: $o → 5
```

String は Magnitude を継承していませんが、<、= などの通常の *comparing* メソッドをサポートしています。さらに String#match: は、いくつかの基本的な glob スタイルのパターンマッチを行うのに便利です。

```
**or* match: 'zorro' → true
```

さらに進んだ正規表現のサポートが必要なときは、Vassili Bykov による *Regex* パッケージを検討してください。

String は比較的多くの変換メソッドをサポートしています。それらの多くは、asDate、asFileName などのような他のクラスへのショートカットコンストラクタメソッドです。また capitalized や translateToLowercase などのような、文字列から他の文字列に変換する便利なメソッドもたくさんあります。

文字列とコレクションに関するさらに進んだ話題については、第 9 章を参照してください。

## 8.5 Boolean

Boolean クラスを見れば、Smalltalk の言語機能のどれほど多くがクラスライブラリ側に押しやられているかということについて、非常に興味深い洞察を得ることができます。Boolean は、Singleton クラスである True と False の抽象スーパークラスです。

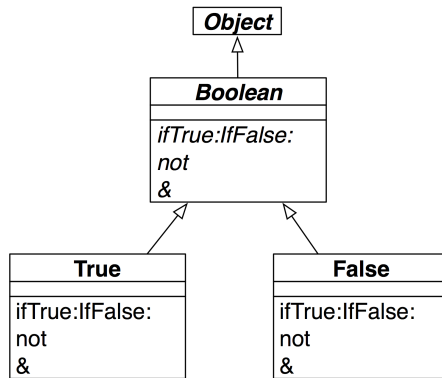


Figure 8.3: Boolean のクラス階層

Boolean の振る舞いのほとんどは、ifTrue:ifFalse: メソッドを注意深く調べれば理解できるでしょう。このメソッドは、引数として二つの Block (ブロック) を取ります。

```
(4 factorial > 20) ifTrue: [ 'bigger' ] ifFalse: [ 'smaller' ] → 'bigger'
```

このメソッドは、Boolean の中では抽象メソッドです。具象サブクラスでの実装は、いずれも簡単なものです：

### Method 8.13: ifTrue:ifFalse: の実装

```
True>>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
  ↑trueAlternativeBlock value
```

```
False>>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
  ↑falseAlternativeBlock value
```

実際、これが OOP の本質です：メッセージがオブジェクトに送信されたら、オブジェクト自身が、どのメソッドを使って反応するかを決めるのです。ifTrue:ifFalse: では、True のインスタンスは *true* 側の選択肢 (ブロック) を評価し、反対に False のインスタンスは *false* 側の選択肢 (ブロック) を評価します。Boolean のすべての抽象メソッドは、True と False の中でこのような方法で実装されています。例えば：


## Method 8.14: 否定を実装する

```
True>>not
"否定--レシーバが true なので false と答える。"
↑false
```

Boolean には、ifTrue:、ifFalse:、ifFalse:ifTrue などのような便利メソッドがいくつかあります。また強欲・怠惰な論理積・論理和から好きなものを選ぶことができます。

```
(1>2) & (3<4)      →  false  "両辺を評価する"
(1>2) and: [ 3<4 ] →  false  "左辺(レシーバ)だけを評価する"
(1>2) and: [ (1/0) > 0 ] →  false  "右辺(引数ブロック)は評価されないで、例外は発生しない"
```

最初の例では、& の両辺の Boolean の部分式が評価されます。& は (評価済みの) Boolean 引数を受け取ります。二つ目と三つ目の例では、and: は引数として Block を取り、結果的に左辺 (レシーバ) しか評価されません。右辺 (Block) は、レシーバが true のときにだけ評価されるからです。

 and: と or: がどのように実装されているか想像してみましょう。そして Boolean と True と False での実装を確認してみましょう。

## 8.6 まとめ

- = をオーバーライドしたら hash もオーバーライドしましょう。
- 自作のオブジェクトのコピーを正しく実装するには、postCopy をオーバーライドします。
- ブレークポイントを設定するには self halt を送信します。
- メソッドを抽象メソッドにするには、本体に self subclassResponsibility と書きます。
- オブジェクトの String 表現を提供するには、printOn: をオーバーライドしましょう。
- インスタンスを適切に初期化するには、フックメソッドである initialize をオーバーライドします。
- Number の一連のメソッドは、自動的に Float、Fraction、Integer 間の変換を行います。
- Fraction は、浮動小数点数ではなく本物の有理数を表します。
- (0 から 255 の範囲の) Character のインスタンスは一意です。



- `String` は変更可能 (`mutable`) ですが、`Symbol` はそうではありません。しかし文字列リテラルを変更しないように気をつけましょう!
- `Symbol` は一意ですが、`String` はそうではありません。
- `String` と `Symbol` は `Collection` なので、通常の `Collection` のメソッドをサポートします。



# Chapter 9

## コレクション

### 9.1 はじめに

コレクションクラスは、`Collection` と `Stream` の汎用のサブクラスのグループで大まかに定義されます。「ブルーブック」<sup>1</sup>には、こうしたグループの要素として `Collection` の 17 個のサブクラスと `Stream` の 9 個のサブクラス、合計 28 個のクラスが取り上げられています。これらは、Smalltalk-80 システムがリリースされる前に何度か再設計されたものでした。これらのクラスのグループは、しばしばオブジェクト指向設計の模範的な実例としてみなされます。

Pharo では、抽象クラスである `Collection` には 101 個のサブクラスが、同じく抽象クラスの `Stream` には 50 個のサブクラスがありますが、これらのうち多く (`Bitmap`、`FileStream`、`CompiledMethod` など) はシステムやアプリケーションで使われるように作られた専用のクラスなので、「`Collection`」カテゴリ内にはありません。この章では「コレクション階層」という言葉は、`Collection` と特に `Collections-*` というカテゴリ内にあるその 47 個のサブクラスを指します。同様に「ストリーム階層」という言葉は、`Stream` と特に `Collections-Streams` カテゴリ内にあるその 9 個のサブクラスを指します。これら 56 個のクラスは、982 個のメッセージに反応し、合計 1609 個のメソッドを定義しているのです<sup>2</sup>!

この章では、図 9.1 に示したコレクションクラスのサブセットに焦点を当てます。ストリームについては別に第 10 章で論じます。

---

<sup>1</sup>Adele Goldberg and David Robson, *Smalltalk 80: the Language and its Implementation*. Reading, Mass.: Addison Wesley, May 1983, ISBN 0-201-13688-0.

<sup>2</sup>訳注: Pharo 1.3 では `Collection` には 75 個のサブクラス、`Stream` には 39 個のサブクラスがあります。また「コレクション階層」と「ストリーム階層」の合計 54 個のクラスは 1660 個のメソッドを定義しており、993 個のメッセージに反応します。

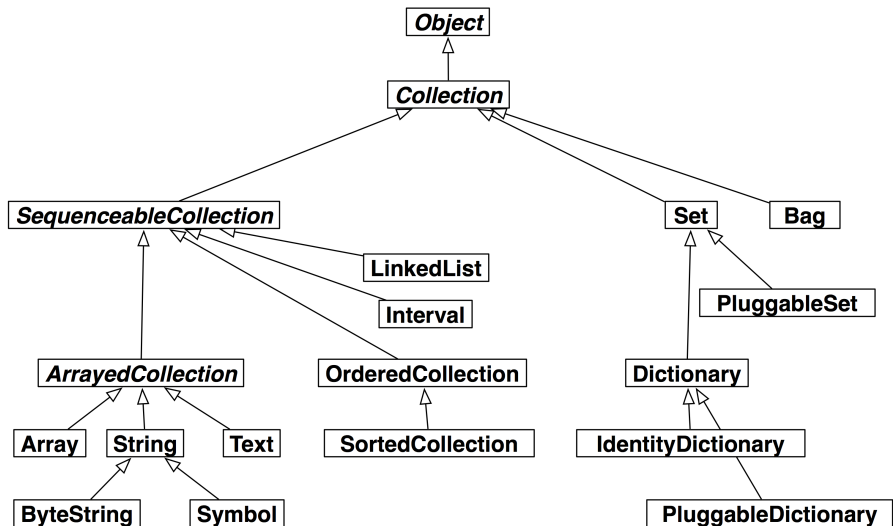


Figure 9.1: Pharo における主要なコレクションクラス

## 9.2 コレクションの種類

コレクションクラスをうまく使うには、幅広い種類のコレクションの実装とそれぞれの共通点と違いについて、少なくとももうわべだけは理解しておかなければなりません。

個別の要素についてではなくコレクションについてプログラムすることは、プログラムの抽象度を高める上で重要な手法の一つです。Lisp の `map` 関数は、このようなスタイルを実現した初期の例です。`map` は、関数とリストを引数に取り、リストの各要素に対して関数を適用してその結果をリストにして返します。Smalltalk-80 はこのようなスタイルを推し進め、「コレクションに基づく」プログラミングを中心に据えたのです。ML や Haskell のようなモダンな関数型プログラミング言語は、Smalltalk の後継者であると言えます。

このアイデアがいいのはなぜでしょう？例えば学生についてのレコードの集合と言ったデータ構造があったとして、何かの基準に一致するすべての学生のレコードについて同じアクションを取りたかったとします。命令型言語に慣れたプログラマなら、ループを使うことをすぐに思いつくでしょう。しかし、Smalltalk プログラマならこう書くでしょう：

```
students select: [ :each | each gpa < threshold ]
```

これは「students の要素の中で、ある条件を満たすもの」からなる新しいコレ

プロトコル	メソッド
<i>accessing</i>	size, capacity, at: <i>anIndex</i> , at: <i>anIndex</i> put: <i>anElement</i>
<i>testing</i>	isEmpty, includes: <i>anElement</i> , contains: <i>aBlock</i> , occurrencesOf: <i>anElement</i>
<i>adding</i>	add: <i>anElement</i> , addAll: <i>aCollection</i>
<i>removing</i>	remove: <i>anElement</i> , remove: <i>anElement</i> ifAbsent: <i>aBlock</i> , removeAll: <i>aCollection</i>
<i>enumerating</i>	do: <i>aBlock</i> , collect: <i>aBlock</i> , select: <i>aBlock</i> , reject: <i>aBlock</i> , detect: <i>aBlock</i> , detect: <i>aBlock</i> ifNone: <i>aNoneBlock</i> , inject: <i>aValue</i> into: <i>aBinaryBlock</i>
<i>converting</i>	asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection: <i>aBlock</i>
<i>creation</i>	with: <i>anElement</i> , with:with:, with:with:with:, with:with:with:with:, withAll: <i>aCollection</i>

Figure 9.2: 標準的なコレクションプロトコル

クシオンを返します。角括弧で囲まれた式が true を返すことが条件です<sup>3</sup>。この Smalltalk コードには、ドメイン特化言語<sup>4</sup>的なシンプルさとエレガンスがあります。

select: メッセージは Smalltalk のすべてのコレクションが理解します。学生のデータ構造が配列なのか連結リストなのかを調べる必要はありません。どちらも select: メッセージを理解します。ループを使った場合、students が配列なのか連結リストなのかをあらかじめ知っておかなければなりません、それは大きく異なるということに注意しましょう。

Smalltalk では、特定のコレクションを指定せずに単に「コレクション」と言った場合には、要素に関する次のプロトコルが明確に定義されたオブジェクトを指します: ある要素が含まれているかテスト (test) するプロトコル、要素の列挙 (enumerate) を行うプロトコル。すべてのコレクションは、testing メッセージ: includes: 、isEmpty、occurrencesOf: を理解します。すべてのコレクションは、enumeration メッセージ: do: 、select: 、reject: (select: の反対)、collect: (Lisp の map と同じ)、detect:ifNone: 、inject:into: (左畳込みを行う) などを理解します。このプロトコルの普遍性とバラエティが、コレクションをとっても強力なものにしています。

図 9.2 は、コレクション階層内のほとんどのクラスが標準でサポートしているプロトコルをまとめたものです。それぞれのメソッドは、Collection のサブクラス内で定義または再定義され、最適化され、時にはその使用を禁じられることもあります。

このような基本的な一貫性を前提として、サポートするプロトコルが異なっ

<sup>3</sup>角括弧内の式は無名関数  $\lambda x. x \text{ gpa} < \text{threshold}$  を定義する入式だと考えることができます。

<sup>4</sup>訳注: domain-specific query language

たり、同じリクエストに対しての振る舞いが異なる様々なコレクションが存在しています。これらの違いのうち重要なものについて見ていきましょう。

- **順序付き (Sequenceable):** `SequenceableCollection` のすべてのサブクラスのインスタンスは、`first` 要素から始まり、`last` 要素まで明確に決まった順序で並んでいます (`sequenceable`)。一方で、`Set`、`Bag`、`Dictionary` のインスタンスは順序付きではありません。
- **ソート済み:** `SortedCollection` は、その要素を常にソートされた状態に保ちます。
- **添字参照/添字付き:** ほとんどの順序付きコレクションは添字付きです。つまり `at:` で要素を取り出せます。`Array` (配列) は添字付きのデータ構造の中でもとても身近なもので、固定されたサイズを持ちます。`anArray at: n` は `anArray` の `n` 個目の要素を取り出し、`anArray at: n put: v` は `n` 個目の要素の値を `v` に変えます。`LinkedList` と `SkipList` は、順序付きですが添字付きではありません。つまり `first` と `last` は理解しますが、`at:` は理解しません<sup>5</sup>。
- **キー参照:** `Dictionary` とそのサブクラスのインスタンスは、添字の代わりにキーで要素を参照できます。
- **変更可能性 (Mutable):** ほとんどのコレクションは変更可能 (`mutable`) ですが、`Interval` と `Symbol` は例外です。`Interval` クラスは、`Integer` が扱える範囲の大きさを持つ変更不能 (`immutable`) なコレクションです。例えば `5 to: 16 by: 2` は、要素 5、7、9、11、13、15 を含む `Interval` オブジェクトです。これは `at:` を使って添字で参照できますが、`at:put:` で値を変更することはできません。
- **動的拡張:** `Interval` や `Array` のインスタンスは、常にサイズが固定されています。他の種類のコレクション (`SortedCollection`、`OrderedCollection`、`LinkedList` は、生成後にサイズを拡張することができます。  
`OrderedCollection` クラスは、`Array` より汎用的です。`OrderedCollection` は動的にサイズが拡張され、`addFirst:` や `addLast:` と呼ぶメソッドが、`at:` や `at:put:` メソッドの他にあります。
- **重複の許容:** `Set` は要素の重複を取り除きますが、`Bag` はそういったことをしません。重複の判定に、`Dictionary`、`Set`、`Bag` は、各要素の `=` メソッドを用います。これらのクラスの `Identity` な変種は、二つの引数が同じオブジェクトであるかをテストする `==` メソッドを用い、`Pluggable` な変種であれば、コレクションの作成時に与えられた任意の同値関係を用います。
- **異種性 (Heterogeneous):** ほとんどのコレクションは、どんな種類の要素も格納できます。一方で `String`、`CharacterArray`、`Symbol` は、`Character` だけしか入れられません。`Array` には様々なオブジェクトが混在できますが、`ByteArray` はバイトだけを、`IntegerArray` は `Integer` だけを、`FloatArray`

<sup>5</sup>`LinkedList>at:put:` はありませんが、`LinkedList>at:` はあります。`SkipList` というクラスはありません。

Arrayed Implementation	Ordered Implementation	Hashed Implementation	Linked Implementation	Interval Implementation
Array String Symbol	OrderedCollection SortedCollection Text Heap	Set IdentitySet PluggableSet Bag IdentityBag Dictionary IdentityDictionary PluggableDictionary	LinkedList SkipList	Interval

Figure 9.3: 実装テクニックにより分類されたコレクションクラス

は Float だけを格納できます。LinkedList の要素には、必ず *Link* の *accessing* プロトコルを受け付けなければならないという制約があります。

### 9.3 コレクションの実装

機能による分類だけではなく、どのようにコレクションクラスが実装されているかについても知っておく必要があります。図 9.3 に示すように、主要な実装テクニックとしては五つものがあります。

1. 配列は、要素をコレクションオブジェクト自身の (添字付きの) インスタンス変数に格納します。結果として配列のサイズは固定され、その代わりに配列は、一回のメモリ割り当てで生成できます。
2. OrderedCollection と SortedCollection は要素を配列に格納し、インスタンス変数でその配列を参照します。こうすることで、内部の配列のサイズよりもコレクションが大きくなったらより大きい配列と入れ替えることができます。
3. Set や Dictionary の類もまた、ストレージへのアクセスに補助的な配列を参照しますが、これをハッシュテーブルとして用います。Bag は、内部の要素そのものとその出現回数をキーと値のペアとして持つ補助的な Dictionary を使います。
4. LinkedList は、標準的な片方向リンクの内部形式を取ります。
5. Interval は、開始点、終了点、ステップサイズの三つの整数を格納します。

これらの基本的なクラスの他に、Array、Set、それから辞書の多くには、「Weak」な変種があります。これらのコレクションは、要素を「弱く」保持します。すなわち要素のガーベージコレクションを禁止しないということです。Pharo のバーチャルマシンはこれらのクラスを判別し、特別に扱います。

Smalltalk のコレクションについてより詳しく知りたい場合は、LaLonde と Pugh の素晴らしい本<sup>6</sup> を参照してください。

## 9.4 主要なクラスの例

よく使う、あるいは重要なコレクションクラスについて、簡単なコード例で説明しましょう。コレクションの主なプロトコルは以下の通りです: `at`、`at:put`: — 要素へのアクセス、`add`、`remove`: — 要素の追加または削除、`size`、`isEmpty`、`include`: — コレクションについての情報の取得、`do`、`collect`、`select`: — コレクションについての繰り返し。それぞれのコレクションによって、これらのプロトコルを実装したりしなかったりしますし、実装しているときは、コレクションの意味論に沿うようにプロトコルを解釈します。それぞれのクラスをブラウズしてみて、具体的でより上級のプロトコルを探してみるといいでしょう。

以下では、最もよく使う次のクラスを取り上げます: `OrderedCollection`、`Set`、`SortedCollection`、`Dictionary`、`Interval`、`Array`。

よく使う生成プロトコル。コレクションのインスタンスを作る方法はいくつかあります。`new`: メソッドと `with`: メソッドを使うのが一番汎用的なやり方でしょう。`new: anInteger` は、サイズが `anInteger` ですべての要素が `nil` であるコレクションを作ります。`with: anObject` はコレクションを作り、それに `anObject` を追加します。コレクションごとにこの振る舞いの実装は異なるでしょう。

初期要素を持つコレクションを作るには、`with`、`with:with` などのメソッドを使うことができます。`with:with:...` 方式では、最大 6 個の要素を持ったコレクションまで作ることができます。

```
Array with: 1 → #(1)
Array with: 1 with: 2 → #(1 2)
Array with: 1 with: 2 with: 3 → #(1 2 3)
Array with: 1 with: 2 with: 3 with: 4 → #(1 2 3 4)
Array with: 1 with: 2 with: 3 with: 4 with: 5 → #(1 2 3 4 5)
Array with: 1 with: 2 with: 3 with: 4 with: 5 with: 6 → #(1 2 3 4 5 6)
```

あるコレクションのすべての要素を別の種類のコレクションに加えるには、`addAll` が使えます:

```
(1 to: 5) asOrderedCollection addAll: '678'; yourself → an OrderedCollection(1 2 3
4 5 $6 $7 $8)
```

`addAll`: は引数を返します。レシーバを返すわけではないのに注意してください!

`withAll`: や `newFrom`: で、いろいろなコレクションを作ることができます。

<sup>6</sup>Wilf LaLonde and John Pugh, *Inside Smalltalk: Volume 1*. Prentice Hall, 1990, ISBN 0-13-468414-1.



```
Array withAll: #(7 3 1 3)      → # (7 3 1 3)
OrderedCollection withAll: #(7 3 1 3) → an OrderedCollection(7 3 1 3)
SortedCollection withAll: #(7 3 1 3) → a SortedCollection(1 3 3 7)
Set withAll: #(7 3 1 3)       → a Set(7 1 3)
Bag withAll: #(7 3 1 3)      → a Bag(7 1 3 3)
Dictionary withAll: #(7 3 1 3) → a Dictionary(1->7 2->3 3->1 4->3)
```

```
Array newFrom: #(7 3 1 3)      → # (7 3 1 3)
OrderedCollection newFrom: #(7 3 1 3) → an OrderedCollection(7 3 1 3)
SortedCollection newFrom: #(7 3 1 3) → a SortedCollection(1 3 3 7)
Set newFrom: #(7 3 1 3)       → a Set(7 1 3)
Bag newFrom: #(7 3 1 3)      → a Bag(7 1 3 3)
Dictionary newFrom: {1 -> 7. 2 -> 3. 3 -> 1. 4 -> 3} → a Dictionary(1->7 2->3 3->1 4->3)
```

この二つのメソッドは同じではないことに注意してください。具体的には、Dictionary class»withAll: では引数を値のコレクションと解釈しますが、Dictionary class»newFrom: はアソシエーションのコレクションが与えられることを期待しています。

## Array

Array (配列) は、サイズが固定のコレクションです。また配列の要素には、整数の添字でアクセスできます。C の流儀とは異なり、Smalltalk の配列の先頭要素は位置 1 です。0 ではありません。配列の要素にアクセスする主なプロトコルは、at: と at:put: です。at: anInteger は、添字 anInteger にある要素を返します。at: anInteger put: anObject は、anObject を添字 anInteger の位置に入れます。配列はサイズ固定のコレクションなので、配列の末尾に要素を追加したり、末尾の要素を削除したりはできません。以下のコードは、サイズ 5 の配列を作り、最初の三つの要素にそれぞれ値を入れ、先頭要素の値を返します。

```
anArray := Array new: 5.
anArray at: 1 put: 4.
anArray at: 2 put: 3/2.
anArray at: 3 put: 'ssss'.
anArray at: 1 → 4
```

Array クラスのインスタンスを作るにはいくつか方法があります。new:, with:, それから #( ) 構文と { } 構文を使うこともできます。

**new:** による生成。 new: anInteger は、サイズ anInteger の配列を生成します。Array new: 5 は、サイズ 5 の配列を生成します。

**with:** による生成。 `with:` メソッドは、要素の値を引数として与えることができます。次のコードでは、数値 4、分数 3/2、文字列 'lulu' の三つの要素を持つ配列を生成します。

```
Array with: 4 with: 3/2 with: 'lulu'  → {4. (3/2). 'lulu'}
```

**#()** を用いたリテラル生成。 `#()` は、静的な (または「リテラル」) 要素を持つリテラル配列を生成します。静的というのは、要素の値が実行時ではなく式がコンパイルされたときにわかっているなければならない、ということです。以下のコードは、最初の要素が (リテラル) 数値 1 で次の要素が (リテラル) 文字列 'here' であるサイズ 2 の配列を生成します。

```
#(1 'here') size  → 2
```

`#(1+2)` を評価した場合、数値 3 だけを要素として持つ配列ではなく、`#(1 #+ 2)` すなわち数値 1、シンボル `#+`、数値 2 の三つの要素を持つ配列になります。

```
#(1+2)  → #(1 #+ 2)
```

これは、`#()` 構文によって、配列に格納されている式がリテラルとしてコンパイラに解釈されるからです。式はスキャンされて結果として得られた要素が新しい配列に格納されます。リテラル配列は、数値、`nil`、`true`、`false`、シンボル、文字列を格納できます。

**{}** を用いた動的生成。最後に `{}` 構文を使って動的配列を作ることができます。`{ a . b }` は `Array with: a with: b` と等価です。つまり、`{}` と `}` で囲まれた式が実行されるということです。

```
{ 1 + 2 }  → #(3)
{(1/2) asFloat} at: 1  → 0.5
{10 atRandom . 1/3} at: 2  → (1/3)
```

**要素へのアクセス。** すべての配列は、`at:` および `at:put:` で要素にアクセスできます。

```
anArray := #(1 2 3 4 5 6) copy.
anArray at: 3  → 3
anArray at: 3 put: 33.
anArray at: 3  → 33
```

リテラル配列を変更するコードには十分注意してください! リテラル配列について、コンパイラは領域を最初の一回だけ確保します。そのため配列をコピーしない限り、上の例を二回目に評価したとき、「リテラル」配列の値は期待通りにはならないでしょう。(コピーを取らなければ、二回目の実行時にはリテラ

ル配列 #(1 2 3 4 5 6) は実は #(1 2 33 4 5 6) になってしまっているのです! 動的配列にはこのような問題はありませ<sup>7</sup>ん。

## OrderedCollection

OrderedCollection は、自動でサイズを拡張し、順序付きで要素を追加できるコレクションです。OrderedCollection は、add:、addFirst:、addLast:、addAll: などの多様なメソッドを提供します。

```
ordCol := OrderedCollection new.
ordCol add: 'Seaside'; add: 'SqueakSource'; addFirst: 'Monticello'.
ordCol → an OrderedCollection('Monticello' 'Seaside' 'SqueakSource')
```

要素を削除する。メソッド remove: anObject は、anObject と一致する最初のオブジェクトをコレクションから削除します。もしそのオブジェクトが存在しない場合はエラーを発生させます。

```
ordCol add: 'Monticello'.
ordCol remove: 'Monticello'.
ordCol → an OrderedCollection('Seaside' 'SqueakSource' 'Monticello')
```

remove: の変種に、remove:ifAbsent: があります。remove:ifAbsent: の第二引数には、削除しようとした要素がコレクションに存在しないときの動作をブロックとして渡すことができます。

```
res := ordCol remove: 'zork' ifAbsent: [33].
res → 33
```

変換。Array に対して (またはその他どんなコレクションに対しても)、asOrderedCollection メッセージを送信して、そのコレクションを OrderedCollection に変換することができます:

```
#(1 2 3) asOrderedCollection → an OrderedCollection(1 2 3)
'hello' asOrderedCollection → an OrderedCollection($h $e $l $l $o)
```

## Interval

Interval クラスは数値の並びを表現します。例えば 1 から 100 までの数の並びは、以下のように定義できます:

<sup>7</sup>訳注: 同一コンパイル単位内に字面がまったく同じリテラル配列が出現すると、それらは同一のオブジェクトとして扱われます。

```
Interval from: 1 to: 100  → (1 to: 100)
```

この Interval オブジェクトを `printString` した結果を見ると、Number クラスに `to:` という便利メソッド (便利コンストラクタ) があって、これを使って Interval オブジェクトを生成することもできます。

```
(Interval from: 1 to: 100) = (1 to: 100)  → true
```

Interval class`»from:to:by:` または Number`»to:by:` によって、以下のように二つの値の間のステップを指定することができます。

```
(Interval from: 1 to: 100 by: 0.5) size  → 199
(1 to: 100 by: 0.5) at: 198  → 99.5
(1/2 to: 54/7 by: 1/3) last  → (15/2)
```

## Dictionary

Dictionary (辞書) は、要素がキーによってアクセスされる重要なコレクションです。Dictionary のメッセージの中でよく使うものとしては、`at:`、`at:put:`、`at:ifAbsent:`、`keys`、`values` があります。

```
colors := Dictionary new.
colors at: #yellow put: Color yellow.
colors at: #blue put: Color blue.
colors at: #red put: Color red.
colors at: #yellow  → Color yellow
colors keys         → a Set(#blue #yellow #red)
colors values       → {Color blue. Color yellow. Color red}8
```

辞書は、キーを同値性によって比較します。= で比較したときに真であるならば、二つのキーは等しいとみなされます。よくあるけれど見つけにくいバグは、キーとして使うオブジェクトで `=` メソッドを再定義しているにもかかわらず、`hash` メソッドを再定義していないというものです。これら二つのメソッドは、どちらも辞書の実装の中でオブジェクトの比較に用いられます。

コレクション階層はサブクラスの考え方に基づいており、サブタイプに基づいているわけではありません。そのことは、Dictionary クラスに明確に示されています。Dictionary は Set のサブクラスですが、普通は Dictionary を Set のように使おうとは思わないでしょう。しかし実装を見ると、Dictionary はアソシエーション (キーと値。→ メッセージにより生成される) の集合であることが明らかです。そのため、Dictionary はアソシエーションのコレクションから作ることもできますし、辞書をアソシエーションの配列に変換することもできます。

<sup>8</sup>訳注: Pharo 1.3 では rgb 値が返ります。

```
colors := Dictionary newFrom: { #blue->Color blue . #red->Color red . #yellow->Color
  yellow }.
colors removeKey: #blue.
colors associations  → {#yellow->Color yellow. #red->Color red}9
```

**IdentityDictionary**。辞書が二つのキーが同じであるかを判定するのに = メッセージおよび hash メッセージを用いるのに対し、IdentityDictionary は、キーの同一性 (== メッセージ) を用います。すなわち二つのキーが同じオブジェクトであるときにだけ、これらのキーは等しいとみなされます。

しばしば Symbol はキーとして用いられますが、そのような場合に IdentityDictionary を使うのはごく自然なことです。なぜなら、Symbol はグローバルに一意であることが保証されているからです。一方、キーが String の場合は、普通の Dictionary クラスを使う方が賢明です。そうしないと問題が生じるでしょう。

```
a := 'foobar'.
b := a copy.
trouble := IdentityDictionary new.
trouble at: a put: 'a'; at: b put: 'b'.
trouble at: a      → 'a'
trouble at: b      → 'b'
trouble at: 'foobar' → 'a'
```

a と b はそれぞれ別のオブジェクトなので、異なるオブジェクトとして扱われます。興味深いことですが、リテラル 'foobar' は一度しか確保されないの、a とまったく同じオブジェクトになります<sup>10</sup>。このような微妙な振る舞いに依存したコードを書きたいなんて思わないでしょう! 普通の Dictionary は、'foobar' と同値 (equal) なキーに対して同じ値を返します。

IdentityDictionary のキーとしては、グローバルに一意であるオブジェクト (Symbol や SmallInteger など) だけを使ってください。String (やその他のオブジェクト) は、普通の Dictionary のキーとして使いましょう。

特筆すべきは、グローバル変数 Smalltalk<sup>11</sup> が SystemDictionary のインスタンスであり、SystemDictionary は IdentityDictionary のサブクラスなので、Smalltalk のすべてのキーは Symbol (実際は 8 ビット文字しか格納できない ByteSymbol) だということです。

```
Smalltalk keys collect: [ :each | each class ]  → a Set(ByteSymbol)12
```

<sup>9</sup>訳注: Pharo 1.3 では rgb 値が返ります。

<sup>10</sup>訳注: 同一コンパイル単位内に限ります。

<sup>11</sup>訳注: Pharo 1.3 では Smalltalk globals

<sup>12</sup>訳注: Pharo 1.3 では長い Array が返ります。

keys または values メッセージを Dictionary に送信すると結果は Set になります  
が、このクラスについては次に説明します<sup>13</sup>。

## Set

Set は、数学的な「集合」のように振る舞うコレクションです。すなわち重複した要素を持たず、要素間には順序がありません。Set に要素を加えるには add: メッセージを用いますが、at: によって要素にアクセスすることはできません。Set に格納されるオブジェクトは、メソッド hash と = を実装していなければいけません。

```
s := Set new.  
s add: 4/2; add: 4; add:2.  
s size  → 2
```

Set を作るには、Set class»newFrom: または変換メッセージ Collection»asSet を用いることができます。

```
(Set newFrom: #(1 2 3 1 4)) = #(1 2 3 4 3 2 1) asSet  → true
```

asSet は、コレクションから重複を取り除くのに便利に使えます。

```
{ Color black. Color white. (Color red + Color blue + Color green) } asSet size  → 2
```

Color red + Color blue + Color green = Color white であることに注意してください。

Bag は Set ととても似ていますが、重複を許すという違いがあります。

```
{ Color black. Color white. (Color red + Color blue + Color green) } asBag size  → 3
```

集合演算である *union* (和集合)、*intersection* (積集合)、要素を含むかのテストは、Collection メソッドの union: 、 intersection: 、 includes: として実装されています。レシーバが最初に Set に変換されるので、これらの演算はすべてのコレクションでちゃんと動くのです!

```
(1 to: 6) union: (4 to: 10)  → a Set(1 2 3 4 5 6 7 8 9 10)  
'hello' intersection: 'there'  → 'he'  
#Smalltalk includes: $k  → true
```

後で説明するように、Set の各要素はイテレータでアクセスできます (9.5 節参照)。

## SortedCollection

OrderedCollection とは対照的に SortedCollection は、ソート順を保ちつつ要素を格納します。デフォルトでは、SortedCollection はソート順を決めるのに <= メッ

<sup>13</sup>訳注: Pharo 1.3 では Array が返ります。

セージを用いるので、抽象クラス `Magnitude` のサブクラスのインスタンスをソートできます。`Magnitude` は、比較プロトコル (`<`, `=`, `>`, `>=`, `between:and:...`) を定義しています (第 8 章参照)。

`SortedCollection` を作るには、まず新しいインスタンスを生成し、要素を追加していきます。

```
SortedCollection new add: 5; add: 2; add: 50; add: -10; yourself.  →  a
SortedCollection(-10 2 5 50)
```

しかし普通は、既にあるコレクションに変換メッセージ `asSortedCollection` を送信して `SortedCollection` を作ります:

```
#(5 2 50 -10) asSortedCollection  →  a SortedCollection(-10 2 5 50)
```

この例は、次の FAQ に対する答えとなります:

FAQ: コレクションをソートするにはどうすればいいでしょうか?  
 答え: `asSortedCollection` メッセージを送信します。

```
'hello' asSortedCollection  →  a SortedCollection($e $h $l $l $o)
```

ではソートした結果を `String` に戻すにはどうすればいいでしょうか? 残念ながら `asString` が返す `printString` 表現は、思った通りのものではありません。

```
'hello' asSortedCollection asString  →  'a SortedCollection($e $h $l $l $o)'
```

正解は、`String class>>newFrom:`、`String class>>withAll:`、`Object>>as:` のいずれかを用いることです。

```
'hello' asSortedCollection as: String  →  'ehllo'
String newFrom: ('hello' asSortedCollection)  →  'ehllo'
String withAll: ('hello' asSortedCollection)  →  'ehllo'
```

すべての要素が互いに比較可能である限り、`SortedCollection` は異なった種類の要素を持つことができます。例えば整数、浮動小数点数、分数と言った異なった種類の数を混ぜることができます。

```
{ 5. 2/-3. 5.21 } asSortedCollection  →  a SortedCollection((-2/3) 5 5.21)
```

もしかしたら、`<=` メッセージを理解しないオブジェクトをソートしたかったり、`<=` 以外の基準でソートしたかったりするかもしれません。この場合は、`SortedCollection` の生成時にソートブロックと呼ばれる 2 引数のブロックを与えることで、ソートが可能になります。例えば、`Color` クラスは `Magnitude` ではありませんし `<=` メソッドも実装していませんが、次のようにソートブロックを指定することで、色を明度 (明るさの尺度) によってソートすることができます。

```
col := SortedCollection sortBlock: [:c1 :c2 | c1 luminance <= c2 luminance].
col addAll: { Color red. Color yellow. Color white. Color black }.
col → a SortedCollection(Color black Color red Color yellow Color white)
```

## String

Smalltalk の String (文字列) は、Character のコレクションとして表現されます。このコレクションは順序付きで、添字付きで、変更可能で、同じ種類の要素すなわち Character のインスタンスしか持つことができません。Array のように、String は専用の文法を持ち、普通は String リテラルを一重引用符で囲むことにより直接生成しますが、通常のコレクション作成メソッドもちゃんと動きます。

```
'Hello' → 'Hello'
String with: $A → 'A'
String with: $h with: $i with: $! → 'hi!'
String newFrom: #($h $e $! $! $o) → 'hello'
```

実際は、String は抽象クラスです。String をインスタンス化したときに実際に得られるのは、8 ビット文字の ByteString か 32 ビット文字の WideString です。話を簡単にするために、その差異は無視して単に String のインスタンスについて述べます。

String の二つのインスタンスは、カンマによって連結できます。

```
s := 'no', ', ', 'worries'.
s → 'no worries'
```

文字列は変更可能なコレクションなので、at:put: メソッドを用いて変更することもできます。

```
s at: 4 put: $h; at: 5 put: $u.
s → 'no hurries'
```

カンマメソッドは Collection で定義されているので、どんな種類のコレクションでも使えることに注意してください!

```
(1 to: 3), '45' → #(1 2 3 $4 $5)
```

以下に示すように、既にある文字列を replaceAll:with: または replaceFrom:to:with: を使って変更することもできます。後者の文字数と指定した間隔の長さが異なるとエラーが発生するので、注意してください。

```
s replaceAll: $n with: $N.
s → 'No hurries'
s replaceFrom: 4 to: 5 with: 'wo'.
s → 'No worries'
```



上のメソッドと異なり、`copyReplaceAll:` は新たな文字列を生成します (興味深いことに、引数は個別の文字というよりも部分文字列なので、サイズが一致している必要はありません)。

```
s copyReplaceAll: 'ries' with: 'mbats' → 'No wombats'
```

実装をちょっと見てみれば、これらのメソッドは `String` 専用ではなくどんな `SequenceableCollection` にも使えることがわかるので、例えば次のような例もちゃんと動きます。

```
(1 to: 6) copyReplaceAll: (3 to: 5) with: { 'three'. 'etc.' } → #(1 2 'three' 'etc.' 6)
```

文字列のマッチ。 `match:` メッセージを送信することによって、パターンと文字列が一致するかどうかを調べることができます。パターンでは、`*` を任意の長さの文字列へのマッチ、`#` を 1 文字へのマッチの意味で使うことができます。注意したいのは、`match:` はパターンに送信されるのであって、調べたい文字列に送信されるのではないということです。

```
'Linux *' match: 'Linux mag' → true
'GNU/Linux #ag' match: 'GNU/Linux tag' → true
```

もう一つ便利なメソッドは `findString:` です。

```
'GNU/Linux mag' findString: 'Linux' → 5
'GNU/Linux mag' findString: 'linux' startingAt: 1 caseSensitive: false → 5
```

Perl 互換のより高度のパターンマッチングの機能が必要ななら、`Regex` パッケージを使うこともできます。

文字列に対するテスト。 以下の例は、`String` だけでなく他の一般的なコレクションにも定義されているメッセージ: `isEmpty:`、`includes:`、`anySatisfy:` を文字列に使ったときの動作を示しています。

```
'Hello' isEmpty → false
'Hello' includes: $a → false
'JOE' anySatisfy: [:c | c isLowercase] → false
'Joe' anySatisfy: [:c | c isLowercase] → true
```

文字列のテンプレート化。 文字列のテンプレート化 に便利なメッセージが三つあります: `format:`、`expandMacros`、`expandMacrosWith:`。

```
{1} is {2} format: {'Pharo' . 'cool'} → 'Pharo is cool'
```

`expandMacros` 系のメッセージは、`<n>` によるキャリッジリターン展開、`<t>` によるタブ展開、`<1s>`、`<2s>`、`<3s>` による引数展開 (`<1p>`、`<2p>`) もほぼ同様で

すが、これらは文字列を一重引用符で囲みます)、`<1?value1:value2>` による条件付き展開などをサポートしています。

```
'look-<t>-here' expandMacros           → 'look- -here'
'<1s> is <2s>' expandMacrosWith: 'Pharo' with: 'cool' → 'Pharo is cool'
'<2s> is <1s>' expandMacrosWith: 'Pharo' with: 'cool' → 'cool is Pharo'
'<1p> or <1s>' expandMacrosWith: 'Pharo' with: 'cool' → "'Pharo' or Pharo'
'<1?Quentin:Thibaut> plays' expandMacrosWith: true   → 'Quentin plays'
'<1?Quentin:Thibaut> plays' expandMacrosWith: false  → 'Thibaut plays'
```

その他のユーティリティメソッド。 `String` クラスは他にもたくさんのユーティリティメソッドを提供しています。例えば、`asLowercase`、`asUppercase`、`capitalized`。

```
'XYZ' asLowercase   → 'xyz'
'xyz' asUppercase   → 'XYZ'
'hilaire' capitalized → 'Hilaire'
'1.54' asNumber     → 1.54
'this sentence is without a doubt far too long' contractTo: 20 → 'this sent...too long'
```

`printString` メッセージを送信してオブジェクトにその文字列表現を問い合わせたときと、`asString` メッセージを送信してオブジェクトを文字列に変換したときの結果は、一般的に異なるということに注意してください。例えば次の通り。

```
#ASymbol printString → '#ASymbol'
#ASymbol asString    → 'ASymbol'
```

シンボルは文字列と似ていますが、グローバルに一意であることが保証される点で異なります。このため辞書の、とりわけ `IdentifyDictionary` のインスタンスのキーとして、シンボルは文字列より適しています。 `String` と `Symbol` については、第 8 章も参照してください。

## 9.5 コレクションイテレータ

`Smalltalk` では、ループや条件分岐は、コレクションや整数やブロックのようなオブジェクトへの単なるメッセージ送信です (第 3 章も参照のこと)。 `todo` のような低水準のメッセージ — 初期値から最終値まで変化する数値を引数としてブロックを評価する — に加え、 `Smalltalk` のコレクション階層は様々な高水準のイテレータを提供しています。このようなイテレータを使うことで、コードがより堅牢でコンパクトになります。

## 繰り返し (do:)

do: メソッドは、基本的なコレクションイテレータです。do:は、引数 (1 引数ブロック) をレシーバの各要素に順に適用していきます。次の例では、レシーバに含まれるすべての文字列をトランスクリプトにプリントします。

```
#('bob' 'joe' 'toto') do: [:each | Transcript show: each; cr].
```

変種。 do: にはたくさんの変種が存在します。例えば do:without:、doWithIndex:、reverseDo: などです: 添字付きのコレクション (Array、OrderedCollection、SortedCollection) 用に doWithIndex: メソッドがあって、これを使うとブロック内から現在の添字にもアクセスできます。このメソッドは、Number クラスで定義されている to:do: に関係しています。

```
#('bob' 'joe' 'toto') doWithIndex: [:each :i | (each = 'joe') ifTrue: [↑i]] → 2
```

OrderedCollection の reverseDo: は、要素を逆順に処理します。

以下のコードは、do:separatedBy: という興味深いメッセージについて示しています。do:separatedBy: は、二つの要素の間においてだけ 2 番目のブロックを実行します。

```
res := "".
#('bob' 'joe' 'toto') do: [:e | res := res, e ] separatedBy: [res := res, '.'].
res → 'bob.joe.toto'
```

ただしこのコードは、特に効率的だとは言えません。このように一時文字列を作る代わりに、書き込みストリームを用いて結果をバッファした方がよいということに注意しましょう (第 10 章参照):

```
String streamContents: [:stream | #('bob' 'joe' 'toto') asStringOn: stream delimiter: '.']
→ 'bob.joe.toto'
```

**Dictionary.** do: メッセージが辞書に送信されたときにチェックされる要素は、「値」であって「アソシエーション」ではありません。keysDo:、valuesDo:、associationsDo: を使う方がより適当です。それぞれキー、値、アソシエーションについて繰り返します<sup>14</sup>。

```
colors := Dictionary newFrom: { #yellow -> Color yellow. #blue -> Color blue. #red ->
    Color red }.
colors keysDo: [:key | Transcript show: key; cr].           "キーの一覧を表示する"
colors valuesDo: [:value | Transcript show: value;cr].    "値の一覧を表示する"
colors associationsDo: [:value | Transcript show: value;cr]. "アソシエーションの一覧
    を表示する"
```

<sup>14</sup>訳注: do: と valuesDo: は、ほとんど同じ意味です。従って両者は同じ程度不便です。

## 結果を集める (collect:)

コレクションの要素に何かの処理を行い、その結果を新たなコレクションにしたい場合、`do:` を用いるより `collect:` またはその他のイテレータメソッドを用いた方がいいでしょう。これらのメソッドのほとんどは、`Collection` やそのサブクラスの *enumerating* プロトコルに分類されています。

例えば各要素を倍にしたコレクションを新たに作る場合を考えてみましょう。`do:` メソッドを使うと、次のように書かなければなりません:

```
double := OrderedCollection new.
#(1 2 3 4 5 6) do: [:e | double add: 2 * e].
double  →  an OrderedCollection(2 4 6 8 10 12)
```

`collect:` メソッドは、引数で渡されたブロックを各要素に対し実行し、その結果を格納した新たなコレクションを返します。`collect:` を用いるとコードはとてもシンプルになります。

```
#(1 2 3 4 5 6) collect: [:e | 2 * e]  →  #(2 4 6 8 10 12)
```

`do:` に比べた `collect:` の利点は、次の例でよりいっそう劇的になります。整数のコレクションを用意して、その絶対値からなるコレクションを生成します:

```
aCol := #(2 -3 4 -35 4 -11).
result := aCol species new: aCol size.
1 to: aCol size do: [:each | result at: each put: (aCol at: each) abs].
result  →  #(2 3 4 35 4 11)
```

上の例と、ずっとシンプルな以下の式を比べてみてください。

```
#(2 -3 4 -35 4 -11) collect: [:each | each abs]  →  #(2 3 4 35 4 11)
```

二つ目の解のさらに良い点は、`Set` や `Bag` でも同じように動くことです。

一般的に、コレクションの要素それぞれにメッセージを送信したいのであれば、`do:` の使用は避けるべきです。

`collect:` メッセージを送信すると、レシーバと同じ種類のコレクションが返ることに注意しましょう。以下のコードが失敗するのはこれが原因です (`String` は整数を保持できません)。

```
'abc' collect: [:ea | ea asciiValue]  "エラー!"
```

こうする代わりに、最初に文字列を `Array` または `OrderedCollection` に変換しておかなければなりません:

```
'abc' asArray collect: [:ea | ea asciiValue]  →  #(97 98 99)
```

実は、`collect:` がレシーバと完全に同じクラスのコレクションを返すという保証はなく、単に同じ「種類 (*species*)」を返すだけです。例えば、`Interval` のイ

インスタンスに `collect:` を送信すると、実際には `Array` のインスタンスが返ります。`Interval»species` が `Array` を返すからです!

```
(1 to: 5) collect: [:ea | ea * 2]  →  #(2 4 6 8 10)
```

## 要素の選別 (`select:` と `reject:`)

`select:` メソッドは、レシーバの要素の中で与えられた条件を満たすものを返します:

```
(2 to: 20) select: [:each | each isPrime]  →  #(2 3 5 7 11 13 17 19)
```

`reject:` は、その反対です:

```
(2 to: 20) reject: [:each | each isPrime]  →  #(4 6 8 9 10 12 14 15 16 18 20)
```

## `detect:` による要素の特定

`detect:` メソッドは、レシーバの要素の中で与えられたブロック引数を満たす最初のものを返します。

```
'through' detect: [:each | each isVowel]  →  $o
```

`detect:ifNone:` メソッドは `detect:` の変種です。`detect:ifNone:` の二つ目のブロックは、最初のブロックを満たす要素が存在しないときに評価されます。

```
Smalltalk allClasses detect: [:each | '*cobol*' match: each asString] ifNone: [ nil ]
→ nil
```

## `inject:into:` を用いて結果を累積する

関数型言語はしばしば、*fold* や *reduce* と言った高階関数を用意しています。これらはコレクションのすべての要素に二項演算子を繰り返し適用して、結果を累積します。Pharo でこのようなことをするには、`Collection»inject:into:` が使えます。

第一引数は初期値です。第二引数は 2 引数ブロックで、これまでの累積結果と今回処理する要素を受け取ります。

`inject:into:` の簡単な例として、数のコレクションの総和を求めてみましょう。ガウス少年のように、Pharo では 1 から 100 までの自然数の和を以下のように書けるのです:

```
(1 to: 100) inject: 0 into: [:sum :each | sum + each ]  →  5050
```

別の例として、階乗を計算する 1 引数ブロックは以下のように書けます。

```
factorial := [:n | (1 to: n) inject: 1 into: [:product :each | product * each ]].
factorial value: 10 → 3628800
```

## その他のメッセージ

**count:** count: メッセージは、与えられた条件を満たす要素の数を返します。条件は、真偽値を返すブロックで表現します。

```
Smalltalk allClasses count: [:each | 'Collection*' match: each asString ] → 315
```

**includes:** includes: メッセージは、引数がコレクションの中に含まれているかどうかを調べます。

```
colors := {Color white . Color yellow . Color red . Color blue . Color orange}.
colors includes: Color blue. → true
```

**anySatisfy:** anySatisfy: メッセージは、引数で渡された条件を満足する要素がコレクション中に一つでもあれば true を返します。

```
colors anySatisfy: [:c | c red > 0.5] → true
```

## 9.6 コレクションを使うときのヒント

**add:** を使うときのよくある間違い。 次のエラーは、Smalltalk で最も頻繁に見られる間違いです。

```
collection := OrderedCollection new add: 1; add: 2.
collection → 2
```

変数 `collection` は、新たに作られたコレクションではなく最後に追加された値を保持しています。これは、`add:` メソッドが追加された要素を返し、レシーバを返すわけではないことによるものです。

次のコードによって期待した結果を得ることができます。

```
collection := OrderedCollection new.
collection add: 1; add: 2.
collection → an OrderedCollection(1 2)
```

<sup>15</sup>訳注: Pharo 1.3 では 5 が返ります。

また、`yourself` メッセージを用いることもできます。`yourself` はカスケードされたメッセージのレシーバを返します。

```
collection := OrderedCollection new add: 1; add: 2; yourself → an
OrderedCollection(1 2)
```

繰り返しを行っているコレクションから要素を削除する。 もう一つよくある間違いは、繰り返しを行っているコレクションから要素を削除しようとする事です。

```
range := (2 to: 20) asOrderedCollection.
range do: [:aNumer | aNumber isPrime ifFalse: [ range remove: aNumber ]].
range → an OrderedCollection(2 3 5 7 9 11 13 15 17 19)
```

9 と 15 が除外されなければいけないので、この結果は明らかに間違っています!

解決策は、処理の前にコレクションをコピーしておくことです。

```
range := (2 to: 20) asOrderedCollection.
range copy do: [:aNumer | aNumber isPrime ifFalse: [ range remove: aNumber ]].
range → an OrderedCollection(2 3 5 7 11 13 17 19)
```

`=` と `hash` は、両方再定義しなければならない。 発見が難しいのは、`=` を再定義して `hash` を再定義していないというエラーです。 `Set` に入れたはずの要素がいつの間になくなっていたりとか、そういうおかしな振る舞いが症状として現れます。 ケント・ベックが提案した一つの解決策は、`xor:` を用いて `hash` を再定義することです。 例えば、著者もタイトルも同じ二つの本は「等しい」と考えられるでしょうから、そのときは次のように、`=` だけでなく `hash` も再定義します:

#### Method 9.1: `=` と `hash` を再定義する

```
Book»= aBook
self class = aBook class ifFalse: [↑ false].
↑ title = aBook title and: [ authors = aBook authors]

Book»hash
↑ title hash xor: authors hash
```

また、変更可能なオブジェクト すなわちハッシュ値が時間と共に変わっていくようなオブジェクトを `Set` の要素、あるいは `Dictionary` のキーとして用いると、別のたちの悪い問題が生じます。 デバッグがお好きでなければ、そういうことは止めておきましょう!

## 9.7 まとめ

Smalltalk のコレクション階層は、異なった種類のコレクションを同じように扱える共通の語彙を持っています。

- 主な違いは、`SequenceableCollection`が要素を与えられた順序で保持するのに対し、`Dictionary`とそのサブクラスはキーと値のアソシエーションを保持し、`Set`と`Bag`は順序を持たないことです。
- ほとんどのコレクションは、`asArray`、`asOrderedCollection`などのメッセージを送信することで、他のコレクションに変換できます。
- コレクションをソートするには、`asSortedCollection`メッセージを送信します。
- リテラルの `Array` は `#(...)` という特別な文法で作ることができます。動的な `Array` は `{...}` という文法で作ることができます。
- `Dictionary` はキーを同値性で比較します。この性質は、キーとして `String` のインスタンスを用いるときに最も便利です。一方 `IdentifyDictionary` はオブジェクトの同一性でキーを判別します。この性質は、キーとして `Symbol` を用いるときか、オブジェクトの参照を値にマップするときなどに適しています。
- `String` は、通常のコレクションメッセージを受け取ることができます。加えて `String` は、簡単な形式のパターンマッチングもサポートしています。より高度なパターンマッチングが必要な場合は、`Regex` パッケージを検討してください。
- 繰り返し用の基本的なメッセージは、`do:` です。`do:` は命令調のコード、例えばコレクションの各要素を変更するとか、各要素に同じメッセージを送信するようなコードで有用です。
- Smalltalk では、`do:` より `collect:`、`select:`、`reject:`、`includes:`、`inject:into:` その他の高水準なメッセージを用いる方が普通です。これらによって、コレクションを統一的な方法で処理することができます。
- 繰り返しを行っているコレクションの要素を削除してはいけません。繰り返し中にコレクションを変更したいなら、コピーに対して繰り返します。
- `=` の定義をオーバーライドしたなら、`hash` のオーバーライドもお忘れなく!



# Chapter 10

## ストリーム

ストリームは、要素の並びを繰り返し処理するために用いられます。ここで並びとは、順序付きコレクション、ファイル、ネットワークストリームなどです。ストリームは読み込み可能か、書き込み可能か、もしくはその両方です。読み込みまたは書き込みは、常にストリーム内の現在の位置に対して相対的に行われます。ストリームは簡単にコレクションに変換可能ですし、その逆も同様です。

### 10.1 二つの「要素の並び」

ストリームを理解するには、以下のような例えが有効です: ストリームは二つの「要素の並び」すなわち、過去の要素の並びと未来の要素の並びとして表現することができます。ストリームはこの二つの並びの間に位置します。このモデルを理解しておくことは大切です。Smalltalk のすべてのストリーム操作はこのモデルによるからです。このような理由で、ほとんどの Stream クラスは PositionableStream のサブクラスになっています。図 10.1 は、5 個の文字を含むストリームを表しています。このストリームは最初の位置にあり、すなわち、過去の要素は一つもありません。reset メッセージを使えば、いつでもこの位置に戻ることができます。

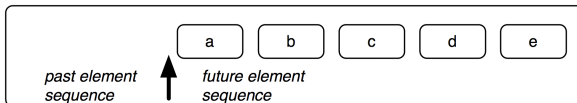


Figure 10.1: 元の位置にあるストリーム。

概念的に、要素の読み込みは、未来の要素の並びから先頭を取り除き、過去となる要素の並びの最後に付け加えることです。next メッセージを使って一

つの要素を読むと、ストリームの状態は図 10.2 のようになります。

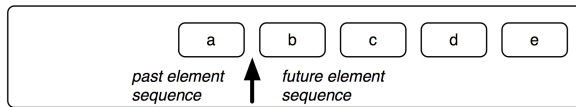


Figure 10.2: 図 10.1 のストリームに `next` を送信した直後の状態: 文字 `a` は「過去」に属するが `b`、`c`、`d`、`e` は「未来」に属する。

要素の書き込みは、未来の要素の並びの先頭を新しいものに置き換えて、置き換えられた要素を過去に移動させることです。図 10.3 は、`nextPut: anElement` メッセージを使って図 10.2 のストリームに `x` を書き込んだ直後の状態を示しています。

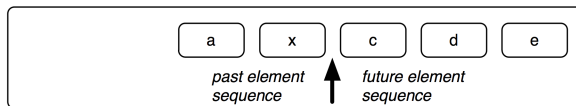


Figure 10.3: 図 10.2 のストリームに `x` を書き込んだ直後の状態。

## 10.2 ストリーム対コレクション

コレクションのプロトコルは、要素の格納と除去と列挙をサポートしますが、これらの操作を混ぜることはできません。例えば `OrderedCollection` の要素を `do:メソッド` によって処理する場合、`do:` ブロック内で要素の追加や削除はできません。あるいはコレクションのプロトコルには、二つのコレクションについて同時に繰り返し、1 回ごとに片方を選んで前へ進めながらもう片方は止めておくようなものはありません。このような処理を行うには、走査用の添字か位置の参照を、コレクション自身の外部で管理する必要があります: これはまさに `ReadStream`、`WriteStream` そして `ReadWriteStream` の役割です。

これら三つのクラスは、コレクションをストリーム処理するために定義されています。例えば、次のコードは `Interval` オブジェクトに対するストリームを作り、ストリームから二つの要素を読み込みます。

```
r := ReadStream on: (1 to: 1000).
r next.   → 1
r next.   → 2
r atEnd.  → false
```

`WriteStream` を使えばコレクションにデータを書き込むことができます:

```
w := WriteStream on: (String new: 5).  
w nextPut: $a.  
w nextPut: $b.  
w contents.  →  'ab'
```

読み込みと書き込みの両方のプロトコルをサポートする `ReadWriteStream` を使うこともできます。

`Pharo` の `WriteStream` と `ReadWriteStream` の主な問題は、配列と文字列しかサポートしていないことです。現在開発中の `Nile` という新しいライブラリがこの状況を変えつつありますが、今のところそれ以外のコレクションをストリーム処理すると、エラーになります<sup>1</sup>:

```
w := WriteStream on: (OrderedCollection new: 20).  
w nextPut: 12.  →  エラー
```

ストリームはコレクション専用ではありません。ファイルやソケットに対しても使えます。次の例は `test.txt` というファイルを作り、二つの文字列を改行で区切って書き込み、ファイルを閉じます。

```
StandardFileStream  
fileNamed: 'test.txt'  
do: [:str | str  
    nextPutAll: '123';  
    cr;  
    nextPutAll: 'abcd'].
```

次の節では、ストリームプロトコルについてさらに詳しく解説します。

## 10.3 コレクションに対するストリーム処理

ストリームはコレクションを扱うのにとっても便利で、コレクションの要素の読み込みと書き込みに使うことができます。コレクションに対するストリーム機能について探っていきましょう。

### コレクションの読み込み

この節ではコレクションの読み込みに使われる機能について見ていきます。コレクションの読み込みにストリームを使うことは、本質的にはコレクションへのポインタを得ることを意味します。ポインタは読み込みによって前へ進み、また希望する場所へ自由に移動させることができます。`ReadStream` クラスを使ってコレクションの要素を読み込みます。

<sup>1</sup>訳注: このコードはエラーにはなりません。

`next` メソッドでコレクションから要素を 1 個読み取り、`next: メソッド` で任意個の要素を読み取ります。

```
stream := ReadStream on: #(1 (a b c) false).
stream next.    → 1
stream next.    → #(a #b #c)
stream next.    → false
```

```
stream := ReadStream on: 'abcdef'.
stream next: 0. → ""
stream next: 1. → 'a'
stream next: 3. → 'bcd'
stream next: 2. → 'ef'
```

`peek` メッセージは、前へ進まずにストリームの次の要素を調べたいときに使います。

```
stream := ReadStream on: '-143'.
negative := (stream peek = $-). "前へ進まずに最初の要素を調べる。"
negative. → true
negative ifTrue: [stream next]. "マイナス文字を無視する"
number := stream upToEnd.
number. → '143'
```

このコードは、ストリームに数の符合があるかどうかで変数 `negative` に真偽値を設定し、数の絶対値を `number` に設定します。`upToEnd` メソッドは、現在の位置から最後までストリームのすべてを返し、ストリームを最後の位置に設定します。このコードは、`peekFor: メソッド` を使うことで単純化できます。`peekFor:` は、ストリームの次の要素がパラメータと同じときは前へ進み、異なるときは進みません。

```
stream := '-143' readStream.
(stream peekFor: $-) → true
stream upToEnd → '143'
```

`peekFor:` はまた、パラメータと次の要素が等しかったかどうかを示す真偽値を返します。

上の例を見て、ストリームを作る新しい方法に気づいたかもしれません: 順序付きコレクションに対して単に `readStream` を送信すれば、そのコレクションに対する読み込みストリームを作ることができます。

**位置決め。** ストリームのポインタの位置を設定するメソッドがあります。もし添字がわかっているならば、`position:` で直接その位置へ移動できます。`position` で現在の位置を求めることができます。ストリームは要素そのものを指しているわけではなく、要素と要素の間を指していることを覚えておいてください。ストリームの先頭に対応する添字は 0 です。

図 10.4 に描かれたストリームの状態を次のコードで得ることができます:

```
stream := 'abcde' readStream.
stream position: 2.
stream peek  →  $c
```

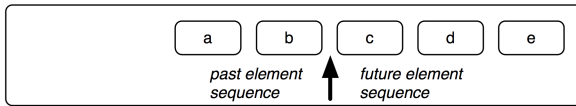


Figure 10.4: 位置 2 にあるストリーム

ストリームを先頭か末尾に移動させるには、それぞれ `reset`、`setToEnd` を使うことができます。`skip`: と `skipTo`: は、現在の位置を基準として前へ進む場合に使います。`skip`: は引数に数を取ることができ、その数だけ要素をスキップします。一方 `skipTo`: は、パラメータと同じ要素が見つかるまでストリームの要素をすべてスキップします。このときストリームは、見つかった要素の後ろに位置決めされることに注意してください。

```
stream := 'abcdef' readStream.
stream next.      →  $a  "ストリームは今、a の真後ろに位置している"
stream skip: 3.   →      "ストリームは今、d の後ろ"
stream position.  →  4
stream skip: -2.  →      "ストリームは b の後ろ"
stream position.  →  2
stream reset.
stream position.  →  0
stream skipTo: $e. →     "ストリームは今、e の真後ろ"
stream next.      →  $f
stream contents.  →  'abcdef'
```

ご覧のように文字 e はスキップされています。

`contents` メソッドは、常にストリーム全体のコピーを返します。

テスト。現在のストリームの状態をテストするためのメソッドがあります。`atEnd` は、これ以上読める要素がない場合のみ真を返します。`isEmpty` は、コレクションにまったく要素がない場合のみ真を返します。

`atEnd` を使ったアルゴリズムの実装例を示します。これは二つのソート済みのコレクションをパラメータとして取り、それらをマージして新たなソート済みのコレクションにします:

```
stream1 := #(1 4 9 11 12 13) readStream.
stream2 := #(1 2 3 4 5 10 13 14 15) readStream.
```

”変数 *result* にはソート済みのコレクションが入る。”

```
result := OrderedCollection new.
[stream1 atEnd not & stream2 atEnd not]
whileTrue: [stream1 peek < stream2 peek
  ”両方のストリームの要素のうち小さい方を取り除き、result に加える。”
  ifTrue: [result add: stream1 next]
  ifFalse: [result add: stream2 next]].
```

”どちらかのストリームは終端に来ていないかもしれない。残りすべてをコピーする。”

```
result
  addAll: stream1 upToEnd;
  addAll: stream2 upToEnd.
```

```
result.  →  an OrderedCollection(1 1 2 3 4 4 5 9 10 11 12 13 13 14 15)
```

## コレクションへの書き込み

`ReadStream` を使ってコレクションの要素を繰り返し読み取る方法は、既に見てきました。次に、`WriteStream` を使ってコレクションを作る方法を学びましょう。

`WriteStream` は、コレクションの様々な場所にたくさんのデータを加えるとき便利です。次の例のように、静的な部分と動的な部分からなる文字列を生成するときしばしば用いられます：

```
stream := String new writeStream.
stream
  nextPutAll: 'This Smalltalk image contains: ';
  print: Smalltalk allClasses size;
  nextPutAll: ' classes.';
  cr;
  nextPutAll: 'This is really a lot.'
```

```
stream contents.  →  'This Smalltalk image contains: 2322 classes.
This is really a lot.'
```

このテクニック<sup>2</sup>は例えば、様々なクラスの `printOn:` メソッドの実装でも使われています。

単に文字列を生成するためだけにストリームを用いる場合、より単純で効率的な方法があります：

<sup>2</sup>訳注: 書き込みストリームにメッセージのカスケードを送信するテクニック。

```
string := String streamContents:
  [:stream |
    stream
      print: #(1 2 3);
      space;
      nextPutAll: 'size';
      space;
      nextPut: $=;
      space;
      print: 3. ].
string.  →  '#(1 2 3) size = 3'
```

**streamContents:** メソッドは、コレクションとそのコレクションへの書き込みストリームを作ります。このメソッドは与えられたブロックを実行しますが、このときこのストリームをパラメータとしてブロックに渡します。ブロックが終了すると **streamContents:** は、コレクションの中身を返します。

このようなコンテキスト<sup>3</sup>では、以下の **WriteStream** のメソッドが特に便利です:

**nextPut:** パラメータをストリームに追加する;

**nextPutAll:** パラメータとして渡されたコレクションの要素をそれぞれストリームに追加する;

**print:** パラメータのテキスト表現をストリームに追加する。

特殊な文字をストリームに表示するのに便利な **space**、**tab**、**cr** (carriage return) などのメソッドもあります。もう一つ便利なメソッドは、**ensureASpace** です。これはストリームの最後の文字がスペースをあることを保証します; ストリームの最後の文字がスペースでない場合にはスペースを追加します。

文字 (列) の連結について。 **WriteStream** に **nextPut:**、**nextPutAll:** を送信することは多くの場合、文字 (列) を連結する最良の方法になります。カンマ演算子 (,) による結合は、これらに比べるとはるかに非効率的です:

```
[] temp |
temp := String new.
(1 to: 100000)
do: [:i | temp := temp, i asString, ' '] timeToRun  →  115176 "(milliseconds)"
```

```
[] temp |
temp := WriteStream on: String new.
(1 to: 100000)
do: [:i | temp nextPutAll: i asString; space].
temp contents] timeToRun  →  1262 "(milliseconds)"
```

<sup>3</sup> 訳注: 書き込みストリームにアクセス可能なコンテキスト。'This Smalltalk image contains: ...' のコード例もその一つ。

ストリームを使った方がずっと効率的になり得るのは、カンマ演算子はレシーバと引数を連結した新たな文字列を作成するので、レシーバと引数の両方をコピーしなければならないからです。もし同じレシーバに対してカンマ演算子で連結を繰り返すと、連結するごとに文字列が長くなり、コピーしなければならない文字数は指数関数的に増えてしまいます。またこの方法は、メモリ上に回収しなければならないゴミをたくさん残します。カンマ演算子の代わりにストリームを使うことは、最適化の技法としてよく知られています。実際、`streamContents: メソッド` (215 ページで述べました) が、ストリームを用いた文字列の連結作業を助けてくれます:

```
String streamContents: [ :tempStream |
  (1 to: 100000)
  do: [:i | tempStream nextPutAll: i asString; space]]
```

## 読み込みと書き込みを同時に行う

ストリームを介してコレクションへの読み込みアクセス、書き込みアクセスを同時に行うこともできます。ウェブブラウザの進む・戻るボタンの動作を管理する `History` (履歴) クラスが作りたかったとしましょう。履歴は 10.5 から 10.11 のような動作をします。



Figure 10.5: 新しい履歴は空。ウェブブラウザには何も表示されていない。

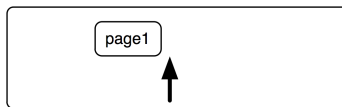


Figure 10.6: ユーザはページ 1 を開く。



Figure 10.7: ユーザはページ 2 へのリンクをクリックする。





Figure 10.8: ユーザはページ 3 へのリンクをクリックする。

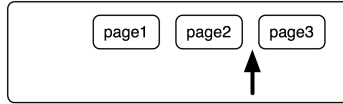


Figure 10.9: ユーザは戻るボタンをクリックする。今、再びページ 2 を見ている。

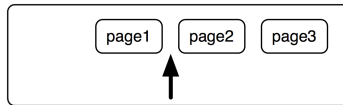


Figure 10.10: ユーザは再び戻るボタンをクリックする。ページ 1 が表示されている。



Figure 10.11: ユーザはページ 1 からページ 4 へのリンクをクリックする。ページ 2、ページ 3 の履歴はなくなる。

この振る舞いは `ReadWriteStream` を使って実装できます。

```
Object subclass: #History
  instanceVariableNames: 'stream'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE-Streams'
```

```
History»initialize
  super initialize.
  stream := ReadWriteStream on: Array new.
```

ここでは特に難しいことはありません。ストリームを持つ新しいクラスを定義します。このストリームは `initialize` メソッドで作られます。

次に、進むと戻るのメソッドが必要です:

```
History»goBackward
self canGoBackward ifFalse: [self error: 'Already on the first element'].
stream skip: -2.
↑ stream next.
```

```
History»goForward
self canGoForward ifFalse: [self error: 'Already on the last element'].
↑ stream next
```

ここまでのコードは実に単純明快です。次に、ユーザがリンクをクリックしたときに動作する `goTo:` メソッドに取りかかりましょう。例えば次のようなコードを思いつくかもしれません:

```
History»goTo: aPage
stream nextPut: aPage.
```

しかしこれは完全ではありません。ユーザがリンクをクリックした後は、それ以上は進めるページがあってはいけないからです。すなわち、進むボタンは無効にならなければいけません。これを実現する最も簡単な方法は、すぐ後ろに `nil` を書き込んで履歴の最後を示すことです:

```
History»goTo: anObject
stream nextPut: anObject.
stream nextPut: nil.
stream back.
```

もはや実装しなければならないメソッドは、`canGoBackward` と `canGoForward` だけになりました。

ストリームは常に二つの要素の間に位置しています。戻るためには現在の位置より前に二つのページがなければいけません: 一つは現在のページ、もう一つは戻りたいページです。

```
History»canGoBackward
↑ stream position > 1

History»canGoForward
↑ stream atEnd not and: [stream peek notNil]
```

ストリームの中身を確認するメソッドを追加しましょう:

```
History»contents
↑ stream contents
```

これで履歴が予告通りに動きます:

```
History new
goTo: #page1;
goTo: #page2;
goTo: #page3;
goBackward;
goBackward;
goTo: #page4;
contents  →  (#page1 #page4 nil nil)
```

## 10.4 ストリームをファイルアクセスに使う

コレクションをストリーム処理する方法については、既に見てきた通りです。同じように、ハードディスクのファイルもストリーム処理することができます。いったん作成されると、ファイルに対するストリームはコレクションに対するストリームとほとんど変わりません: 同じプロトコルでストリームの読み込み、書き込み、位置決めができます。主な違いはストリームの作成方法にあります。ファイルストリームの作成について、いくつかの異なる方法を見ていきましょう。

### ファイルストリームを作成する

ファイルストリームを作るには、以下のインスタンス作成メソッドのいずれかを用いなければなりません。これらのメソッドは、`FileStream` クラスが提供します:

**fileNamed:** 与えられた名前のファイルを、読み込みと書き込みのために開きます。ファイルが既に存在していた場合、その内容は変更されたり置き換えられたりしますが、ファイルを閉じるときに切り捨てられません。ファイル名にディレクトリ部が明示されていない場合、ファイルはデフォルトディレクトリ内に作られます。

**newFileNamed:** 与えられた名前の新しいファイルを作り、そのファイルに書き込むためのストリームを返します。ファイルが既に存在していた場合、ユーザにどうするか尋ねます。

**forceNewFileNamed:** 与えられた名前の新しいファイルを作り、そのファイルに書き込むためのストリームを返します。ファイルが既に存在していた場合、黙ってそのファイルを削除した後に新しいファイルを作ります。

**oldFileNamed:** 与えられた名前の既存のファイル<sup>4</sup>を、読み込みと書き込みのために開きます。ファイルが既に存在していた場合、その内容は変更され

<sup>4</sup>訳注: ファイルが存在しなかった場合、ユーザにどうするか尋ねます。

たり置き換えられたりしますが、ファイルを閉じるときに切り捨てられません。ファイル名にディレクトリ部が明示されていない場合、ファイルはデフォルトディレクトリ内に作られます。

**readOnlyFileNamed:** 与えられた名前の既存のファイルを、読み込みのために開きます。

開いたファイルストリームは忘れずに閉じなければなりません。これには `close` メソッドを使います。

```
stream := FileStream forceNewFileNamed: 'test.txt'.
stream
  nextPutAll: 'This text is written in a file named ';
  print: stream localName.
stream close.

stream := FileStream readOnlyFileNamed: 'test.txt'.
stream contents. → 'This text is written in a file named "test.txt"'
stream close.
```

ファイルストリームに `localName` を送信すると、パス名の最後の要素が返ります。 `fullName` を送信してフルパス名を得ることもできます。

すぐに、ファイルストリームを手作業で閉じるのが苦痛でエラーの元であることに気づくでしょう。 `forceNewFileNamed:do:` は、この問題を解消するために用意されています。このメソッドは、書き込みストリームをパラメータとするブロックを評価してすべての書き込みをすませた後、自動的にこのストリームを閉じます。同様に読み込みストリーム用に `readOnlyFileNamed:do:` があります。

```
FileStream
  forceNewFileNamed: 'test.txt'
  do: [:stream |
    stream
      nextPutAll: 'This text is written in a file named ';
      print: stream localName].
string := FileStream
  readOnlyFileNamed: 'test.txt'
  do: [:stream | stream contents].
string → 'This text is written in a file named "test.txt"'
```

ファイルストリーム作成メソッドのうちブロックを引数に取るものは、まずストリームを作成し、次にストリームを引数としてブロックを実行し、最後にストリームを閉じます。これらのメソッドは、ブロックが返すものを返します。つまりブロックの最後にある式の値を返します。前の例では、ブロックの最後の式: `stream contents` を評価してファイルの中身を得、その値を変数 `string` に代入しています。

## バイナリストリーム

デフォルトでファイルストリームは、テキストベースつまり文字の読み書き用に作られます。ストリームでバイナリデータを扱う場合は、ストリームに `binary` メッセージを送信しなければなりません。

ストリームがバイナリモードの場合、0 から 255 (1 バイト) の数値しか書き込めません。 `nextPutAll`: メソッドを使って一度に二つ以上の数値を書きたいときは、 `ByteArray` オブジェクトを引数として渡さなければなりません。

```
FileStream
forceNewFileNamed: 'test.bin'
do: [:stream |
    stream
        binary;
        nextPutAll: #(145 250 139 98) asByteArray].

FileStream
readOnlyFileNamed: 'test.bin'
do: [:stream |
    stream binary.
    stream size.      → 4
    stream next.      → 145
    stream upToEnd.  → #[250 139 98]
].
```

次の例では「`test.pgm`」という画像ファイルを作成します (PGM: portable graymap)。作成したファイルは、お気に入りのドロッププログラムで開くことができます。

```
FileStream
forceNewFileNamed: 'test.pgm'
do: [:stream |
    stream
        nextPutAll: 'P5'; cr;
        nextPutAll: '4 4'; cr;
        nextPutAll: '255'; cr;
        binary;
        nextPutAll: #(255 0 255 0) asByteArray;
        nextPutAll: #(0 255 0 255) asByteArray;
        nextPutAll: #(255 0 255 0) asByteArray;
        nextPutAll: #(0 255 0 255) asByteArray
    ]
```

これは図 10.12 のような 4x4 のチェッカーボード (市松模様) を作ります。

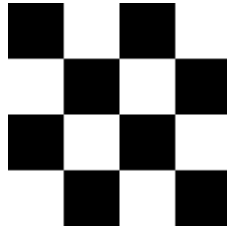


Figure 10.12: バイナリストリームで描くことができる 4x4 のチェッカーボード (市松模様)

## 10.5 まとめ

ストリームは、要素の並びをインクリメンタルに読み書きする場合に、コレクションよりも良い方法を提供します。ストリームとコレクションを相互に変換する簡単な方法があります。

- ストリームは読み込み可能、書き込み可能、あるいは読み書きともに可能です。
- コレクションをストリームに変換する場合、コレクション「に対する (on)」ストリームを定義します (例えば `ReadStream on: (1 to: 1000)`)。あるいはコレクションに `readStream` などのメッセージを送信します。
- ストリームをコレクションに変換する場合、`contents` メッセージを送信します。
- 大きなコレクションを連結する場合は、カンマ演算子を使うよりもストリームを使った方が効率的です。つまりストリームを作り、`nextPutAll:` でコレクションをストリームに追加し、`contents` を送信して結果を取り出します。
- ファイルストリームはデフォルトで文字型です。これを明示的にバイナリにするには、`binary` メッセージを送信します。


# Chapter 11

## Morphic

Morphic は、Pharo の GUI に付けられた名前です。Morphic は Smalltalk で書かれており、様々な OS の間で完全な互換性があります。その結果、Pharo は Unix, MacOS, Windows のいずれにおいてもまったく同じ見た目となっています。Morphic と他の多くのユーザ・インターフェース・ツールキットとの違いは、インターフェースを「構成すること」と「実行すること」とのモードを分けていないということです。画面上のすべてのグラフィック要素 (graphical elements) は、いつでもユーザが組み立てたり分解したりできます。<sup>1</sup>

### 11.1 Morphic の歴史


Morphic は、プログラミング言語 Self 向けとして、John Maloney と Randy Smith によって 1993 年頃から開発が始められました。Maloney は Squeak のために新しく Morphic を書き直しましたが、Self 版の基本的な考え方は今でも Pharo の Morphic に受け継がれています。その基本的な考え方とは、直接性と生命感です。直接性とは、画面上の図形がマウスでクリックして直接調べたり変更したりできるオブジェクトであるということです。生命感とは、ユーザインターフェースがいつでもユーザの操作に反応できることです。つまり、画面上の情報は、それが表している世界が変わるたびに自動的に更新されます。簡単な例を挙げれば、メニューから項目を取り外してボタンとして使うことができます。

 *World* メニューを出します。その上でメタクリックを一回すると、*Morphic* ハローが現れます。<sup>2</sup>その後、取り外したいメニュー項目の上でさらにメタク

---

<sup>1</sup>Hilaire Fernandes に謝意を表します。彼のフランス語で書かれたオリジナル版をこの章のベースとすることを許諾してくれました。

<sup>2</sup>プリファレンス・ブラウザで、`halosEnabled` を設定することを忘れないでください。または、ワークスペース上で `Preferences enable: #halosEnabled` を実行します。

リック をすると、メニュー項目の上にハローが現れます。図 11.1 で示すように、黒いハンドル  を掴んでその項目を画面の適当な場所にドラッグします。

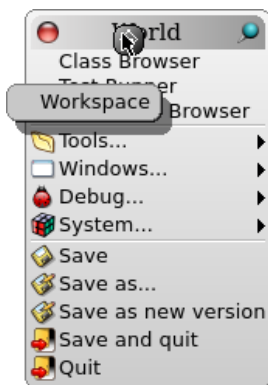



Figure 11.1: `Workspace` メニューのモーフを取り外して独立したボタンにしているところ。


Pharo の起動後、画面上にあるすべてのオブジェクトは、モーフと呼ばれます。つまり、これらのオブジェクトは `Morph` クラスのインスタンスです。 `Morph` クラスは多くのメソッドを持つ大きなクラスであるため、少ないコードで面白い動作をするサブクラスを作れます。あらゆるオブジェクトを表すモーフを作れますが、どのくらい良い表現が得られるかはオブジェクトによります。

 文字列オブジェクトを表すモーフを作るには、ワークスペースで以下のコードを実行します。

```
'Morph' asMorph openInWorld
```

これは、 `'Morph'` という文字列を表すモーフを作り、「ワールド」上で開きます。(つまり、表示します)「ワールド」とは、Pharo の画面のことです。これによりメタクリックで操作できるグラフィック要素 — モーフ — が得られます。

もちろん、今見ているものより面白い表現を持つモーフを定義することも可能です。 `class Object` クラスにある `asMorph` メソッドのデフォルトの実装では、単に `StringMorph` を作るだけです。例えば、 `Color tan asMorph` は、単に `Color tan printString` の実行結果をラベルを持つ `StringMorph` を返します。それでは、これを変更して色の付いた長方形を代わりに返すようにしましょう。

 ブラウザを開き `Color` クラスに以下のメソッドを追加します。

Method 11.1: `Color` のインスタンスを表すモーフを返す


```
Color>>asMorph
↑ Morph new color: self
```



ワークスペースで `Color orange asMorph openInWorld` を実行すれば、文字列のモーフの代わりにオレンジ色の長方形が現れます。

## 11.2 モーフを操作する


モーフはオブジェクトです。つまり、Smalltalk の他のオブジェクトと同様に操作できます。メッセージを送ることで属性を変えたり、新しいMorphのサブクラスを作ったりできます。どんなモーフも（たとえ画面に表示されていなくとも）位置と大きさを持っています。便宜上、すべてのモーフは画面上の四角形の領域を占めていると考えます。不規則な形状の場合であっても、その位置と大きさは、そのモーフを囲む最小の四角形の「箱」となります。この箱を、モーフのバウンディングボックス、あるいは単に「bounds」と言います。positionメソッドは、モーフの左上の座標（またはモーフのバウンディングボックスの左上の座標）を表すPointオブジェクトを返します。この座標系の原点は画面の左上の端にあります。y座標は画面の下方向に増えていき、x座標は右方向に増えていきます。extentメソッドもPointオブジェクトを返しますが、位置ではなくモーフの幅と高さからなる座標値を返します。

 ワークスペースに以下のコードを入力し `do it` します。

```
joe := Morph new color: Color blue.
joe openInWorld.
bill := Morph new color: Color red .
bill openInWorld.
```

さらに、`joe position`と入力して `print it` します。joeを動かすために、`joe position: (joe position + (10@3))`を繰り返して実行します。

大きさについても同様のことが可能です。joe extent はjoeの大きさを返します。joeを大きくするには、`joe extent: (joe extent * 1.1)`を実行します。モーフの色を変えるには、好きなColorオブジェクトを引数にして `color:` メッセージを送ります。例えば、`joe color: Color orange`のようにします。透明度を設定するには、`joe color: (Color orange alpha: 0.5)`を試してください。

 `bill` に `joe` の後を追いかけるには、以下のコードを繰り返して実行します。

```
bill position: (joe position + (100@0))
```

マウスでjoeを移動した後にこのコードを実行すれば、billはjoeの100ピクセル右側に移動するでしょう。

## 11.3 モーフの組み込み

新しいグラフィカルな表現を作る一つの方法は、モーフの内側に別のモーフを置くことです。これをモーフの組み込みと言います。モーフはどんな深さにも組み込めます。モーフを別のモーフに組み込むには、`addMorph:` というメッセージを入れ物となるモーフに送ります。

🕒 あるモーフを別のモーフに加えましょう。

```
star := StarMorph new color: Color yellow.  
joe addMorph: star.  
star position: joe position.
```

最後の行では、星を `joe` と同じ座標に置いています。組み込まれるモーフの座標は、入れ物となるモーフを基準にするのではなく画面を基準とすることに気をつけてください。モーフの位置を設定するための多くのメソッドがあります。Morphクラスの `geometry` をブラウズすれば自分自身で確認できます。例えば、`joe` の中心に星を置くには、`star center: joe center` を実行します。



Figure 11.2: 青色の半透明なモーフである `joe` が星を含んでいる。

マウスで星を掴もうとすると実際には `joe` を掴み、二つのモーフが一緒に動きます。星は `joe` の内側に埋め込まれています。 `joe` の内側にもっと多くのモーフを埋め込むことができます。プログラムによる埋め込みに加え、直接的な操作によっても埋め込むことができます。

## 11.4 自由にモーフを作り、描く

モーフの組み込みによりたくさんの便利で面白いグラフィカルな表現を作れますが、いずれ今までとまったく違う何かを作りたいと思うでしょう。そのためには、Morphのサブクラスを作り `drawOn:` メソッドをオーバーライドすれば、見た目を変更できます。

Morphic のフレームワークは、モーフの再描画が必要となった際に、モーフに対して drawOn:メッセージを送ります。drawOn:メッセージの引数は、Canvas クラス (またはそのサブクラス) のインスタンスで、期待される振る舞いはモーフが自分自身をキャンバスの bounds 内に描くことです。この知識を使って十字モーフを作ってみましょう。

🕒 ブラウザを開き、Morphを継承した CrossMorph クラスを新しく定義します。

### Class 11.2: CrossMorphを定義する

```
Morph subclass: #CrossMorph
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'PBE-Morphic'
```

以下のように drawOn:メソッドを定義できます。

### Method 11.3: CrossMorphを描く

```
drawOn: aCanvas
| crossHeight crossWidth horizontalBar verticalBar |
crossHeight := self height / 3.0 .
crossWidth := self width / 3.0 .
horizontalBar := self bounds insetBy: 0 @ crossHeight.
verticalBar := self bounds insetBy: crossWidth @ 0.
aCanvas fillRectangle: horizontalBar color: self color.
aCanvas fillRectangle: verticalBar color: self color
```



Figure 11.3: Halo を伴った CrossMorph。自由に大きさを変更できる。

モーフに bounds メッセージを送ると、Rectangle クラスのインスタンスであるバウンディングボックスが返されます。Rectangle クラスのインスタンスは数多くのメッセージを理解し、四角形に関連する他の四角形を作れます。この例では、引数に点を指定した insetBy:メッセージを使って高さを低くした四角形を作り、さらに幅を縮めた四角形を作っています。

🕒 新しいモーフをテスト するために、CrossMorph new openInWorld を実行します。

実行すると図 11.3 のようになります。しかし、マウスに反応する場所（つまりモーフを掴むのにクリックするところ）がバウンディングボックス全体になっていることに気づくと思います。これを解決しましょう。

Morphic フレームワークは、モーフがカーソルの下にあるかどうか知る必要があるとき、マウスポインタの下にバウンディングボックスがあるすべてのモーフに対して、containsPoint: メッセージを送ります。つまり、マウスに反応する場所をモーフの十字部分に制限するには、containsPoint: メソッドをオーバーライドします。

 CrossMorphクラスに以下のメソッドを定義します。

Method 11.4: CrossMorphのマウスに反応する場所を設定する。

```
containsPoint: aPoint
| crossHeight crossWidth horizontalBar verticalBar |
crossHeight := self height / 3.0.
crossWidth := self width / 3.0.
horizontalBar := self bounds insetBy: 0 @ crossHeight.
verticalBar := self bounds insetBy: crossWidth @ 0.
↑ (horizontalBar containsPoint: aPoint)
or: [verticalBar containsPoint: aPoint]
```

このメソッドは drawOn: と同じロジックを使っています。つまり、containsPoint: が true と答える領域は、drawOn: によって色が付けられた領域と一致すると確実に言えます。Rectangleクラスの containsPoint: を活用すれば、この面倒な仕事をやりとげることができます。

メソッド Method 11.3 と 11.4 のコードには二つの問題点があります。明らかなのは、コードが重複していることです。これはとても重大な過ちです。というのも、horizontalBar や verticalBar の計算方法を変える必要があるときに、二箇所にあるコードの一方を修正するのを簡単に忘れてしまうからです。解決方法は、これらの計算を新たな二つのメソッドに抽出することです。どちらも private プロトコルに格納します。

Method 11.5: horizontalBar.

```
horizontalBar
| crossHeight |
crossHeight := self height / 3.0.
↑ self bounds insetBy: 0 @ crossHeight
```

Method 11.6: verticalBar.

```
verticalBar
| crossWidth |
crossWidth := self width / 3.0.
↑ self bounds insetBy: crossWidth @ 0
```

これらのメソッドを使って drawOn: と containsPoint: の両方を書き直します。

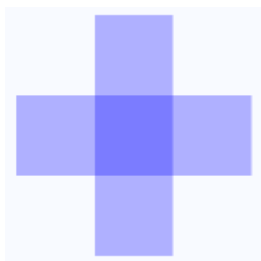


Figure 11.4: 十字の中央が2度描かれる。

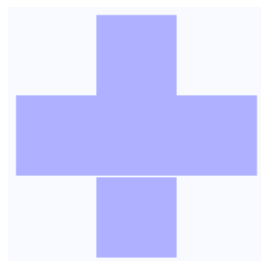


Figure 11.5: 塗られていない部分のある十字モーブ。

#### Method 11.7: リファクタリング後のCrossMorph»drawOn:.

```
drawOn: aCanvas
  aCanvas fillRectangle: self horizontalBar color: self color.
  aCanvas fillRectangle: self verticalBar color: self color
```

#### Method 11.8: リファクタリング後のCrossMorph»containsPoint:.

```
containsPoint: aPoint
  ↑ (self horizontalBar containsPoint: aPoint)
  or: [self verticalBar containsPoint: aPoint]
```

プライベートメソッドに意味のある名前を付けたおかげで、このコードはより簡単に理解できるようになりました。実際、すぐに2番目の問題が明らかになります。それは、十字の中央の部分が `horizontalBar` と `verticalBar` の両者の下になっており、2度描かれてしまうという問題です。不透明な色を塗る場合には問題になりませんが、図 11.4 に示すような半透明な色で描くとすぐにこのバグが露見します。

🔗 ワークスペースで、以下のコードを1行ずつ実行します。

```
m := CrossMorph new bounds: (0@0 corner: 300@300).
m openInWorld.
m color: (Color blue alpha: 0.3).
```

この解決法は、`verticalBar` を三つの部分に分け、上側と下側の部分だけ塗りつぶすということです。Rectangleクラスでこの面倒な仕事をしてくれるメソッドがまた見つかります。r1 areasOutside: r2は、r2の外側に出ているr1を構成する四角形の配列を返します。以下が改良されたコードになります。

Method 11.9: drawOn:メソッドの改良版。十字の中心を一度だけ塗りつぶす。

```
drawOn: aCanvas
  | topAndBottom |
  aCanvas fillRectangle: self horizontalBar color: self color.
```

```
topAndBottom := self verticalBar areasOutside: self horizontalBar.
topAndBottom do: [ :each | aCanvas fillRectangle: each color: self color]
```

このコードは正しく動作するように見えますが、何回か十字の大きさを変更しようとする、図 11.5 のように、ある大きさのときに 1 ピクセル分の幅の線が十字の下側と残りの部分を分けてしまうことに気づきます。これは丸め誤差のせいです。塗りつぶそうとする四角形の大きさが整数でないときに、`fillRectangle: color:` が不適切に数値を丸めてしまい、1 ピクセル分だけ塗りつぶされなくなるからです。このバグは、四角形の大きさを計算する際に明示的に四捨五入することで解決できます。

#### Method 11.10: 明示的に四捨五入した `CrossMorph`»`horizontalBar`

```
horizontalBar
| crossHeight |
crossHeight := (self height / 3.0) rounded.
↑ self bounds insetBy: 0 @ crossHeight
```

#### Method 11.11: 明示的に四捨五入した `CrossMorph`»`verticalBar`

```
verticalBar
| crossWidth |
crossWidth := (self width / 3.0) rounded.
↑ self bounds insetBy: crossWidth @ 0
```


## 11.5 対話とアニメーション

モーフを使って、いきいきとしたユーザインターフェースを実現するには、マウスやキーボードでモーフと対話できなければなりません。さらに、モーフは自分の見た目や位置を変えることで（つまり自分自身のアニメーションによって）、ユーザからの入力に反応する必要があります。

### マウスイベント

マウスボタンが押されたとき、Morphic はマウスポインタの下にある各モーフに対して `handlesMouseDown:` メッセージを送ります。もし、モーフが `true` を返すと、Morphic は直ちにそのモーフに対して `mouseDown:` メッセージを送ります。ユーザがマウスボタンを離れた際にも `mouseUp:` メッセージを送ります。すべてのモーフが `false` を返すと、Morphic はドラッグ&ドロップの操作を開始します。後述するように、`mouseDown:` と `mouseUp:` メッセージは、`MouseEvent` オブジェクトの引数を伴って送られます。この引数は、マウスの動作の詳細を表しています。

CrossMorphクラスを拡張して、マウスイベントを扱うようにしましょう。まず、すべての十字モーフが `handlesMouseDown`: メッセージに確実に `true` を返すことから始めます。

 CrossMorphに以下のメソッドを追加します。



Method 11.12: CrossMorphがマウスクリックに反応するように宣言する。

```
CrossMorph»handlesMouseDown: anEvent
↑true
```

十字モーフのクリックで赤色に変わり、アクションクリックで黄色に変わるようにしましょう。これは Method 11.13 によって実現できます。

Method 11.13: マウスクリックに反応してモーフの色を変えるようにする。

```
CrossMorph»mouseDown: anEvent
anEvent redButtonPressed "click"
  ifTrue: [self color: Color red].
anEvent yellowButtonPressed "action-click"
  ifTrue: [self color: Color yellow].
self changed
```

このメソッドでは、モーフの色を変えた後で `self changed` を送っていることに注意してください。これによって、Morphicは直ちに `drawOn`: を送るようになります。なお、いったんモーフがマウスイベントを処理すると、もはやモーフをマウスで掴んで動かすことができなくなることに気をつけてください。その代わりに Halo を使う必要があります。モーフ上のメタクリックで Halo を出し、モーフの上部にある茶色の  か、赤の  のいずれかでモーフを掴みます。

`mouseDown`: の引数の `anEvent` は、MouseEvent クラスのインスタンスです。これは MorphicEvent クラスのサブクラスです。MouseEvent では、`redButtonPressed` と `yellowButtonPressed` メソッドを定義しています。マウスイベントを扱うメソッドにどんなものがあるか調べるには、このクラスをブラウズします。

## キーボードイベント

キーボードイベントを得るには、次の3ステップを必要とします。

1. 「キーボードフォーカス」を特定のモーフに与えます。例えば、マウスがモーフの上にあるときに、フォーカスを与えることができます。
2. `handleKeystroke`: メソッドを使って、モーフがキーボードイベントを処理するようにします。このメッセージは、ユーザがキーを押した際に、キーボードフォーカスを持っているモーフへ送られます。
3. マウスがモーフの上がないときに、キーボードフォーカスを解放します。

キー入力に反応するように `CrossMorph` クラスを拡張します。まず、マウスがモーフ上にあることを通知するように変更します。これは、`handlesMouseOver:` メッセージに `true` を返すことで実現できます。

🕒 モーフがマウスポインタの下にあるときに、`CrossMorph` が対応できるように宣言します。

Method 11.14: 「マウスオーバー」イベントを扱えるようにする。

```
CrossMorph>handlesMouseOver: anEvent
↑true
```

マウスの位置の取り扱いは `handlesMouseDown:` の場合と同様です。マウスポインタがモーフに入る、または離れる際に、`mouseenter:` と `mouseleave:` メッセージがモーフに送られます。

🕒 `CrossMorph` クラスがキーボードフォーカスの獲得と解放を行えるように、二つのメソッドを定義します。三つ目のメソッドで実際にキー入力を処理します。

Method 11.15: マウスがモーフに入ったときにキーボードフォーカスを獲得する。

```
CrossMorph>mouseenter: anEvent
anEvent hand newKeyboardFocus: self
```

Method 11.16: マウスポインタがモーフから離れたときに、キーボードフォーカスを解放する。

```
CrossMorph>mouseleave: anEvent
anEvent hand newKeyboardFocus: nil
```

Method 11.17: キーボードイベントを受け取って処理する。

```
CrossMorph>handleKeystroke: anEvent
| keyValue |
keyValue := anEvent keyValue.
keyValue = 30 "up arrow"
  ifTrue: [self position: self position - (0 @ 1)].
keyValue = 31 "down arrow"
  ifTrue: [self position: self position + (0 @ 1)].
keyValue = 29 "right arrow"
  ifTrue: [self position: self position + (1 @ 0)].
keyValue = 28 "left arrow"
  ifTrue: [self position: self position - (1 @ 0)]
```

矢印キーを使ってモーフを動かすことができるようなメソッドを書きました。マウスがモーフの上にはないときには、`handleKeystroke:` メッセージが送ら



れないため、モーフはキーボードに反応しなくなることに注意してください。キーの値を知りたいときは、トランスクリプトウィンドウを出し、Transcript show: anEvent keyValueをMethod 11.17に加えてください。handleKeystroke:の引数のanEventは、KeyboardEventのインスタンスで、MorphicEventクラスのサブクラスです。このクラスをブラウズすれば、キーボードイベントについてさらに学習できます。

## Morphic アニメーション

Morphic は主に二つのメソッドからなる単純なアニメーション機能を提供します。step が一定の間隔でモーフに送られ、stepTime は step の間隔をミリ秒単位で指定します。<sup>3</sup>加えて、startStepping は step を開始させ、stopStepping はそれを停止させます。step が始まっているかどうかを知るには、isStepping を使います。

🕒 以下のようなメソッドを定義して、CrossMorphを点滅させます。

Method 11.18: アニメーションの間隔を設定する。

```
CrossMorph>stepTime
↑ 100
```

Method 11.19: アニメーションの step を作る。

```
CrossMorph>step
(self color diff: Color black) < 0.1
  ifTrue: [self color: Color red]
  ifFalse: [self color: self color darker]
```

アニメーションを開始させるには、CrossMorph上でインスペクタを開いて(Haloのデバッグハンドル🔧を使います)、下部の小さなワークスペースにself startSteppingと入力し、do itを実行します。代わりにhandleKeystroke:メソッドを変更して、+キーと-キーでstepを開始、停止させることもできます。

🕒 以下のコードをMethod 11.17に加えます。

```
keyValue = $+ asciiValue
  ifTrue: [self startStepping].
keyValue = $- asciiValue
  ifTrue: [self stopStepping].
```

<sup>3</sup>実際には、stepTimeはstep間の最小時間を指定します。仮にstepTimeを1にしても、Pharoの負荷が高くなりすぎて、それほど頻繁にstepをモーフに送れないので注意してください。

## 11.6 対話機能

ユーザに入力を促すために、UIManager クラスはすぐ使えるダイアログボックスを多く提供しています。例えば、request:initialAnswer: メソッドは、ユーザによって入力された文字列を返します。(図 11.6)

```
UIManager default request: 'What's your name?' initialAnswer: 'no name'
```

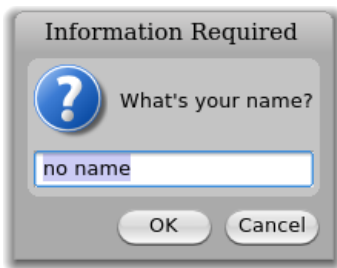


Figure 11.6: 入力ダイアログ

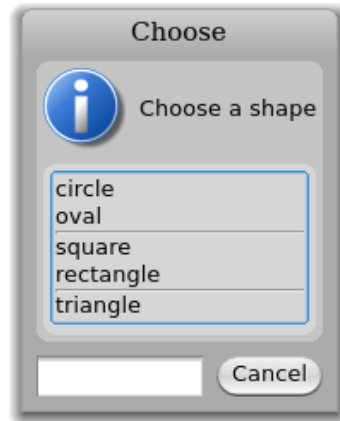



Figure 11.7: ポップアップメニュー

ポップアップメニューを出すには、様々な chooseFrom: メソッドを使います。(図 11.7)

```
UIManager default
```

```
chooseFrom: #('circle' 'oval' 'square' 'rectangle' 'triangle')
```

```
lines: #(2 4) message: 'Choose a shape'
```

 UIManager クラスをブラウズして、提供されている対話メソッドを試してみましょう。

## 11.7 ドラッグ&ドロップ

Morphic はドラッグ&ドロップもサポートします。受け入れ側のモーフとドロップされるモーフの二つのモーフからなる簡単な例について考えてみましょう。受け入れ側のモーフは、ドロップされるモーフが条件を満たしたときだけドロップを受け入れます。この例ではモーフが青色であることを条件とします。受け入れが拒否された場合、ドロップされるモーフが何をするか決めます。

🕒 まず受け入れ側のモーフを定義しましょう。

Class 11.20: 他のモーフを受け入れるモーフを定義する。

```
Morph subclass: #ReceiverMorph
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE-Morphic'
```

🕒 いつものやり方で初期化メソッドを定義します。

Method 11.21: ReceiverMorphを初期化する。

```
ReceiverMorph>initialize
  super initialize.
  color := Color red.
  bounds := 0 @ 0 extent: 200 @ 200
```

受け入れ側のモーフがドロップを受け入れるか否かは、どのように判断されるのでしょうか？ 一般的に、モーフの両者がドロップに同意する必要があります。受け入れ側は `wantsDroppedMorph:event:` メッセージに反応することで決定します。最初の引数はドロップされるモーフ、2番目の引数はマウスイベントです。例えば、受け入れ側はドロップ時に何らかの修飾キーが押されたかどうかを調べられます。ドロップされるモーフも、`wantsToBeDroppedInto:` メッセージが送られたときに、ドロップされる先のモーフが気に入るか調べるチャンスがあります。このメソッドのデフォルトの実装（`Morph`クラスで定義されている）では、`true`を返します。

Method 11.22: 色を基準にしてドロップを受け入れる。

```
ReceiverMorph>wantsDroppedMorph: aMorph event: anEvent
  ↑ aMorph color = Color blue
```

受け入れ側のモーフがドロップを望まない場合には、ドロップされるモーフに何が起こるのでしょうか？デフォルトの動作は、何もしないことです。つまり、ドロップされずに受け入れ側のモーフの上に置かれるだけです。より直感的な動作としては、ドロップされるモーフが元の位置に戻ることでしょう。これは、ドロップされるモーフを拒否する際に、受け入れ側のモーフが `repelsMorph:event:` メッセージに対して `true`を返すことで実現できます。

Method 11.23: ドロップ拒否時のモーフの振る舞いを変える。

```
ReceiverMorph>repelsMorph: aMorph event: ev
  ↑ (self wantsDroppedMorph: aMorph event: ev) not
```

以上が、受け入れ側のモーフに必要なことすべてです。

🕒 ワークスペースで `ReceiverMorph` と `EllipseMorph` のインスタンスを作ります。

```
ReceiverMorph new openInWorld.
EllipseMorph new openInWorld.
```

黄色い EllipseMorphを受け入れ側のモーフにドラッグ&ドロップしましょう。モーフはドロップを拒否され、元の位置に戻るはずです。

 この動作を変えるには、インスペクタを使って楕円のモーフの色を Color blueに変えます。青いモーフは ReceiverMorphに受け入れられるはずはです。

では、DroppedMorphと名付けた Morphクラスの特別なサブクラスを作って、もう少し経験を積みましょう。

Class 11.24: ReceiverMorphにドラッグ&ドロップ可能なモーフを定義する。

```
Morph subclass: #DroppedMorph
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'PBE-Morphic'
```

Method 11.25: DroppedMorphを初期化する。


```
DroppedMorph>>initialize
super initialize.
color := Color blue.
self position: 250@100
```

ドロップされるモーフが受け入れ側に拒否されたときにすべきことを設定できます。ここでは、モーフをマウスポインタに付いたままにします。

Method 11.26: ドロップを拒否されたモーフの反応を定義する。

```
DroppedMorph>>rejectDropMorphEvent: anEvent
|h|
h := anEvent hand.
WorldState
addDeferredUIMessage: [h grabMorph: self].
anEvent wasHandled: true
```

イベントに対して hand メッセージを送ると、HandMorphのインスタンスであるハンドが返されます。このオブジェクトはマウスポインタを表していて、マウスポインタに関係するものは何でも持っています。ここでは、ドロップを拒否されたモーフである selfをハンドが掴むように Worldに指示しています。

 DroppedMorphの二つのインスタンスを作成し、受け入れ側のモーフにドラッグ&ドロップします。

```
ReceiverMorph new openInWorld.
(DroppedMorph new color: Color blue) openInWorld.
(DroppedMorph new color: Color green) openInWorld.
```

緑色のモーフはドロップを拒否されるため、マウスポインタに付いたままになります。

## 11.8 例題

サイコロを転がすモーフを作りましょう。<sup>4</sup>モーフをクリックするとすべての面が高速に表示され、再度クリックするとアニメーションが止まります。



Figure 11.8: Morphic で作られたサイコロ

🕒 外枠を描くため、Morphクラスの代わりに BorderedMorph のサブクラスとしてサイコロを定義します。

Class 11.27: サイコロモーフを定義する。

```
BorderedMorph subclass: #DieMorph
  instanceVariableNames: 'faces dieValue isStopped'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE-Morphic'
```

インスタンス変数の faces は、サイコロの面の個数を記録します。ここで実装するサイコロモーフは、9までの面を許すことにします！ dieValue は現在表示されている面の値を表し、isStopped はサイコロのアニメーションが停止しているときに true となります。サイコロのインスタンスを作るために、DieMorph のクラス側に faces: n メソッドを定義し、n 個の面を持つ新しいサイコロを生成するようにします。

Method 11.28: 好きな個数の面を持つ新しいサイコロを生成する。

```
DieMorph class>>faces: aNumber
  ↑self new faces: aNumber
```

initialize メソッドは今までと同じやり方でインスタンス側に定義します。なお、new は新しく生成されたインスタンスに initialize メッセージを送ることを覚えておいてください。

## Method 11.29: DieMorphのインスタンスを初期化する。

```
DieMorph»initialize
super initialize.
self extent: 50 @ 50.
self useGradientFill; borderWidth: 2; useRoundedCorners.
self setBorderStyle: #complexRaised.
self fillStyle direction: self extent.
self color: Color green.
dieValue := 1.
faces := 6.
isStopped := false
```

サイコロの見た目をよくするために、BorderedMorphクラスのメソッドをいくつか使います。浮き出る効果を得るために枠線を太くしたり、角を丸めたり、表示面にグラデーションをかける等を行います。インスタンス側の faces: メソッドでは、正しい引数が与えられているかをチェックするように定義します。

## Method 11.30: サイコロの面の個数を設定する。

```
DieMorph»faces: aNumber
"Set the number of faces"
(aNumber isInteger
 and: [aNumber > 0]
 and: [aNumber <= 9])
ifTrue: [faces := aNumber]
```

サイコロが生成されたときにメッセージが送られる順序を知るのは良いことです。例えば、DieMorph faces: 9を評価したと仮定します。

1. クラスメソッドの DieMorph class»faces: では、DieMorph classクラスに new メッセージを送ります。
2. (Behaviorクラスから継承された DieMorph classクラスの) newメソッドで、新しいインスタンスが生成され、そのインスタンスに initializeメッセージが送られます。
3. DieMorphの initializeメソッドが、facesインスタンス変数に初期値6を設定します。
4. DieMorph class»newメソッドの結果できたインスタンスが、クラスメソッドの DieMorph class»faces: に返り、その新しいインスタンスに faces: 9メッセージが送られます。
5. インスタンスメソッドの DieMorph»faces: が実行され、facesインスタンス変数を9に設定します。

drawOn: を定義する前に、表示面に描く目の位置を求めるメソッドをいくつか定義します。

Methods 11.31: サイコロの目の位置を求める九つのメソッドを定義する。

```
DieMorph>face1
↑{0.5@0.5}
DieMorph>face2
↑{0.25@0.25 . 0.75@0.75}
DieMorph>face3
↑{0.25@0.25 . 0.75@0.75 . 0.5@0.5}
DieMorph>face4
↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75}
DieMorph>face5
↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.5@0.5}
DieMorph>face6
↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5}
DieMorph>face7
↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5}
DieMorph >face8
↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5 . 0.5@0.25}
DieMorph >face9
↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5 . 0.5@0.25 . 0.5@0.75}
```

上記のメソッドは、サイコロの各面について目の座標をコレクションの形で定義しています。それぞれの座標は  $1 \times 1$  の範囲に収めているので、拡大して実際の目の場所を求めます。

`drawOn:`メソッドは、サイコロの背景を描くための `super`送信と、目を描くという二つの仕事を行います。

Method 11.32: サイコロモーフを描く。

```
DieMorph>drawOn: aCanvas
super drawOn: aCanvas.
(self perform: ('face' , dieValue asString) asSymbol)
do: [:aPoint | self drawDotOn: aCanvas at: aPoint]
```


このメソッドの後半は、Smalltalk のリフレクション機能を使っています。サイコロの面に対応する `faceX`メソッドから得た座標のコレクションに対して、順に `drawDotOn:at:`メッセージを送ることでサイコロの目を描いています。適切な `faceX`メソッドを呼び出すために、`perform:`メソッドを使います。このメソッドが、`('face' , dieValue asString) asSymbol`により作られた文字列をメッセージとして送ります。このような `perform:`の使い方は至るところで見られます。

Method 11.33: サイコロの目を描く。

```
DieMorph>drawDotOn: aCanvas at: aPoint
aCanvas
fillOval: (Rectangle
```

```
center: self position + (self extent * aPoint)
extent: self extent / 6)
color: Color black
```

座標が [0:1] の範囲で正規化されているため、サイコロの大きさに合わせるために、`self extent * aPoint`で拡大します。

 ワークスペースでサイコロのインスタンスを作ることができます。

```
(DieMorph faces: 6) openInWorld.
```

表示面を変えるために、`myDie dieValue: 4`のように使えるアクセサを作ります。

Method 11.34: サイコロの表示面を設定する。

```
DieMorph>>dieValue: aNumber
(aNumber isInteger
 and: [aNumber > 0]
 and: [aNumber <= faces])
ifTrue:
 [dieValue := aNumber.
 self changed]
```

サイコロの面を高速に表示するためにアニメーション機能を使います。

Methods 11.35: サイコロのアニメーションを作る。

```
DieMorph>>stepTime
↑100

DieMorph>>step
isStopped iffFalse: [self dieValue: (1 to: faces) atRandom]
```

サイコロが回ってますね！

サイコロの回転をクリックで始めたり止めたりするには、前述したマウスイベントのやり方を使います。まず、マウスイベントを受け入れるようにします。

Methods 11.36: アニメーションの開始と停止のためにマウスクリックを処理する。

```
DieMorph>>handlesMouseDown: anEvent
↑true

DieMorph>>mouseDown: anEvent
anEvent redButtonPressed
ifTrue: [isStopped := isStopped not]
```

以上により、クリックでサイコロを振ったり止めたりできます。



## 11.9 キャンバスに関する追加情報

`drawOn:`メソッドは、`Canvas` クラスのインスタンスを引数として一つだけ取ります。このキャンバスとは、モーフ自身を描く領域のことです。キャンバスの描画メソッドを使うことで、モーフに自由な外観を与えられます。`Canvas`クラスの階層構造をブラウズすれば、いくつもの変形を見ることができます。通常用いられる `Canvas` クラスの変形は `FormCanvas` クラスなので、`Canvas` クラスと `FormCanvas` クラスの中から主要な描画メソッドを見つけられます。これらのメソッドには、点や線、多角形や四角形、楕円やテキスト、画像などの描画や、回転や拡大・縮小などの機能があります。

透明なモーフやその他の描画メソッド、アンチエイリアスなどを実現するために、他の種類のキャンバスも利用できます。これらの特徴を使うためには、`AlphaBlendingCanvas` クラスや `BalloonCanvas` クラスが必要となるでしょう。しかし、`drawOn:`が引数として `FormCanvas` のインスタンスを受け取るとしたら、`drawOn:`の中でどうやって他のキャンバスを得ればよいのでしょうか？ 幸いにも、あるキャンバスを別の種類に変換することが可能です。

🌀 DieMorphのアルファチャンネル透過率を 0.5 にするキャンバスを使うように、`drawOn:`を再定義します。

Method 11.37: 透明なサイコロを描く。

```
DieMorph>>drawOn: aCanvas
| theCanvas |
theCanvas := aCanvas asAlphaBlendingCanvas: 0.5.
super drawOn: theCanvas.
(self perform: ('face' , dieValue asString) asSymbol)
do: [:aPoint | self drawDotOn: theCanvas at: aPoint]
```

以上で終わりです。



Figure 11.9: アルファチャンネル透過率を設定したサイコロ。

## 11.10 まとめ

Morphic は、グラフィカルなフレームワークで、動的に GUI 要素を組み合わせることができます。

- `asMorph openInWorld`メッセージをオブジェクトに送ると、モーフに変換して画面に表示できます。
- モーフをメタクリックすると、ハンドルが表示されモーフを操ることができるようになります。(ハンドルの機能は個々のバルーンヘルプの説明を見ましょう)
- ドラッグ&ドロップによるか `addMorph:`メッセージを送ることで、モーフを別のモーフに組み込めます。
- 既にあるクラスのサブクラスを作り、`initialize`や `drawOn:`といった主要なメソッドを再定義できます。
- `handlesMouseDown:`メソッドや `handlesMouseOver:`メソッドなどを再定義すれば、マウスやキーボードに反応するようにモーフを制御できます。
- `step`メソッド (何をするか) と `stepTime`メソッド (ミリ秒単位の `step` 間隔) を定義すれば、モーフのアニメーションを作れます。
- `PopUpMenu`や `FillInTheBlank`といった多くの定義済みのモーフを使って、ユーザと対話できます。

## Chapter 12

# Seaside by Example

Seaside は Smalltalk で Web アプリケーションを作成するためのフレームワークです。2002 年に Avi Bryant によって開発が始まりました。一度マスターすれば、デスクトップアプリケーションと同じように Web アプリケーションを容易に作るできるようになります。

Seaside の有名な事例としては SqueakSource<sup>1</sup> と Dabble DB<sup>2</sup>があります。徹底したオブジェクト指向が Seaside の特徴です。XHTML 生成のためのテンプレート、Web ページ間の複雑な遷移の定義、状態を埋め込んだ URL といったものは Seaside では不要です。オブジェクトにメッセージを送るだけです。素敵です。

### 12.1 なぜ Seaside?

最近の Web アプリケーションは、ユーザとのやりとりをデスクトップアプリケーションと同じように行おうとします。アプリケーションはユーザに問いかけ、ユーザはフォームを記入したり、ボタンを押したりしてこれに応えます。ところが Web の仕組みは実際には逆なのです。Web ブラウザがサーバーにリクエストをすると、新しいページがサーバーから返ってきます。そのため Web アプリケーション開発 フレームワークは多くの問題に直面することになります。とりわけ問題なのは、この逆になっている制御フローの取り扱いです。そこで、多くの Web アプリケーションはブラウザの「戻る」ボタンを押すことを禁じています。押されてしまうとセッションの状態を保つのが困難になるからです。Web ページ間の複雑な制御の流れを表現するのは、やっかいなものです。さらに制御が分岐したりすると、表現するのが極めて難しくなるか、場合によっては不可能になってしまいます。

---

<sup>1</sup><http://SqueakSource.com>

<sup>2</sup><http://DabbleDB.com>

Seaside では Web アプリケーション開発を容易にするため、コンポーネントベースのフレームワークを採用しています。まず、制御の流れはメッセージ送信によって自然な形で表すことができます。Seaside は、Web アプリケーションの実行状態を、表示したページと共に覚えています。そのため「戻る」ボタンを押しても問題が生じません。

また、状態は Seaside が自動的に管理します。バックトラック機能が有効な場合、「戻る」ボタンを押すと変更した状態が復元します。トランザクション機能を使えば、戻るボタンを押したときに変更した状態が復元されないようにすることもできます。URL に状態を埋め込んだりする必要もありません。URL 生成も Seaside が行ってくれます。

Web ページはコンポーネント群によって構成されます。それぞれが独立した制御フローを持つことができます。XHTML のテンプレートはなく、簡単な Smalltalk のメッセージ送信によって正しい XHTML がプログラマ的に生成されます。Seaside は CSSCSS をサポートしています。そのためページ内容とレイアウトは明確に分離されます。

最後に、Seaside は、Web ブラウザ上での便利な開発ツールを提供しています。これによりインタラクティブな形で簡単に Web アプリケーションを開発、デバッグできます。サーバーを動作させながら、リコンパイルしてアプリケーションを拡張することだってできるのです。

## 12.2 はじめに

Seaside の Web サイト<sup>3</sup>から “Seaside One-Click Experience” をダウンロードすると、すぐに Seaside を始めることができます。ファイルには Windows, Mac OSX, Linux 用の Seaside 2.8 の環境一式が同梱されています。<sup>4</sup>Seaside のサイトには他にもドキュメントやチュートリアルなど、たくさんのリンクがあります。Seaside は長年にわたって進化を続けているものなので、すべてが最新の Seaside についてのものであるとは限らないので注意してください。

Seaside は Web サーバーを含んでいます。8080 ポートで動かすには WAKom startOn: 8080 を実行します。サーバーを終了するときは、WAKom stop です。デフォルトの管理者ユーザ名は admin であり、パスワードは seaside になっています。WADispatcherEditor initialize と実行すると、新たなユーザとパスワードを聞いてくるようになります。



 Seaside を開始して Web ブラウザで <http://localhost:8080/seaside/> を開きましょう。

図 12.1 のようなページが表示されます。

 examples▷counter ページをクリックしてください。(図 12.2)

<sup>3</sup><http://seaside.st>

<sup>4</sup>訳注：Seaside は頻繁にアップデートが行われており、翻訳時点では既に 3.0 が公開されています

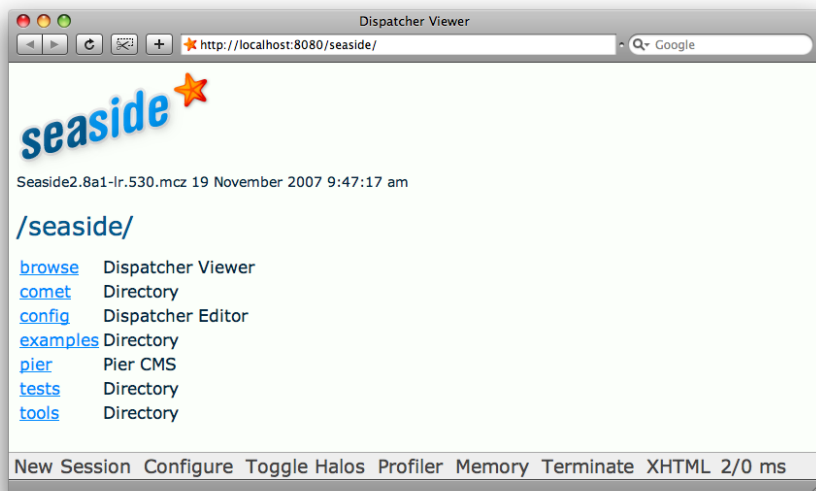


Figure 12.1: Seaside の初期画面

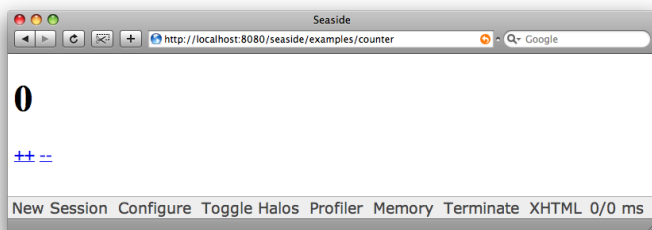


Figure 12.2: カウンターのアプリケーション

これは小さな Seaside アプリケーションのサンプルです。++ や -- のリンクをクリックするとカウンターの値を変えることができます。

🕒 リンクをクリックして値を変えてみましょう。Web ブラウザの「戻る」ボタンで前の状態に戻ってから ++ のリンクを再びクリックしてみてください。カウンターは、表示されていた値から増えています。決して「戻る」ボタンを押し始めたときの値からではありません。

図 12.1 の web ページの下の方にあるツールバーに注目してみてください。Seaside は「セッション」の概念をサポートしており、ユーザごとにアプリケーションの状態を管理します。New Session を押すと、カウンターアプリケーション

ンの新たなセッションが始まります。**Configure** ではアプリケーションの各種設定を Web ブラウザ経由で行うことができます。

(**Configure** の画面を閉じるには、右上の **x** リンクをクリックします)。**Toggle Halos** を押すと、Seaside で動作しているアプリケーションの状態が調べられます。**Profiler** と **Memory** ボタンはアプリケーション実行時のパフォーマンスに関する詳細な情報を見るためのものです。

**XHTML** は生成された Web ページの XHTML をバリデートするときに使います。W3C のバリデータを使っているため、対象とするページが外部からアクセス可能になっていないとこの機能は動作しません。

Seaside のアプリケーションは付け外しの可能な「コンポーネント」からできています。コンポーネントはありふれた Smalltalk オブジェクトであり、他と異なる点と言えばフレームワークが提供する WACComponent を継承したクラスのインスタンスであるということだけです。Pharo のイメージからコンポーネントやクラスを探索できますし、Web ブラウザからハローを通じて直接たどっていくこともできます。

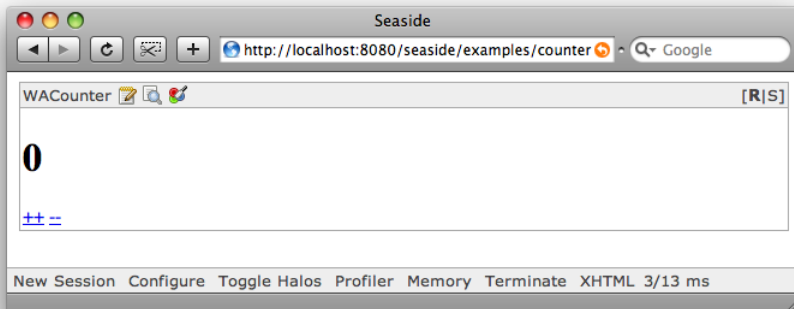


Figure 12.3: ハロー

🔍 **Toggle Halos** を押してみましょう。図 12.3 のような画面になるはずですが。左上に **WACounter** と表示されていますが、これは現在のページの構成要素となっているコンポーネントのクラス名を表しています。その隣には三つのボタンアイコンがあります。最初の鉛筆のアイコンは、このコンポーネントのクラスブラウザを開くためのものです。2 番目の虫眼鏡アイコンでは、現在アクティブになっている **WACounter** のインスタンスのインスペクタを開くことができます。3 番目の色の付いた丸のボタンを押すと、このコンポーネントに対する CSS スタイルシートエディタが表示されます。右上には **R** と **S** のリンクがあります。それぞれクリックするとコンポーネントが通常表示とソース表示とに切り替わります。こうしたリンクをすべて試してみましょう。ソースビューの表示のと

きにも、`++` や `_` は有効になっています。フォーマットもされているため、Web ブラウザのソース表示よりも見やすいかもしれません。

Seaside のクラスブラウザやオブジェクト・インスペクタは、サーバーが別のマシンで動作していて、画面がなかったり、リモートの環境だったりするときには非常に便利です。とはいえ、最初に Seaside のアプリケーションを開発するときには、ローカルでサーバーを動かすと思います。その場合は Pharo のイメージが提供する普段の開発ツールの方が使いやすいでしょう。

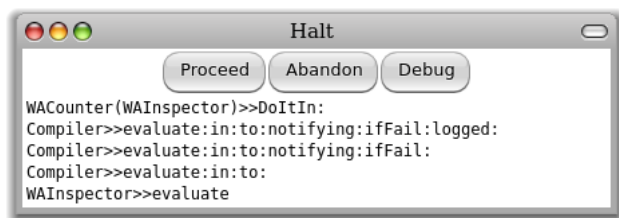


Figure 12.4: カウンターアプリケーションを halt させる

🔍 Web ブラウザのオブジェクト・インスペクタのリンクを使ってカウンターのインスペクタを開き、`self halt` を実行してみましょう。Web ページはロード中の状態になります。Seaside のイメージの方へ移動してみてください。ノートファイアが上がっていて (図 12.4)、WACounter のオブジェクトが `halt` の実行で止まっていることが確認できます。さらにデバッガを開いて `Proceed` を押してみましょう。Web ブラウザに戻るとカウンターは再び動作しています。

Seaside のコンポーネントは、異なるコンテキストで何度でも初期化できます。

🔍 Web ブラウザで `http://localhost:8080/seaside/examples/multicounter` を開いてみてください。いくつもの独立したカウンターからなるアプリケーションが立ち上がります。カウンターをいくつか選んで値を変えてみましょう。「戻る」ボタンを押しても問題なく動作します。ハローを表示してみると、入れ子になったコンポーネント群からアプリケーションができていことがわかります。クラスブラウザを使って、WAMultiCounter の実装がどうなっているか見てみましょう。クラスメソッドが三つ (`canBeRoot`、`description`、`initialize`)、インスタンスメソッドも三つ (`children`、`initialize`、`renderContentOn:`) 定義されていることがわかります。アプリケーションは、コンポーネント階層のルートになれるというだけで、単なるコンポーネントです。クラスメソッドの `canBeRoot` で `true` を返すことで、ルートになるための宣言を行います。

Seaside の Web インターフェースを使うと、アプリケーション (すなわちルートになるコンポーネント) を設定したり、コピーしたり、登録解除したりできます。設定を少しいじってみましょう。

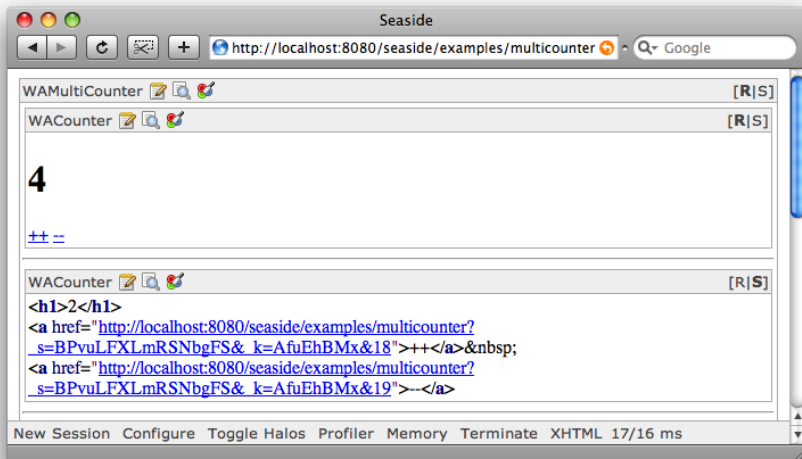


Figure 12.5: 独立したサブコンポーネント

🌀 Web ブラウザで `http://localhost:8080/seaside/config` を開きます。ユーザ ID とパスワードを入れて、ログインしてください。(デフォルトでは `admin/seaside` となっています)。“`examples`” の隣にある `Configure` をクリックします。“`Add entry point`” のところで、“`counter2`” と名前を入れ、タイプは `Application` のまま `Add` ボタンをクリックします(図 12.6 を参照)。次の画面で、`Root Component` を `WACounter` にして、`Save` を押し、`Close` で閉じます。これでカウンターの新しいアプリケーションが `http://localhost:8080/seaside/examples/counter2` で起動するようになりました。同様に、この設定ツール画面を通じて、登録したアプリケーションの解除を行うこともできます。

Seaside は二つのモードで動作します。一つは今まで見てきた `開発モード` です。デプロイモードにすると、画面下のツールバーが非表示となります。デプロイモードにするには、全体設定を行うツールのページからアプリケーションを選び、`Configure` をクリックするか、ツールバーに表示されている `Configure` をクリックします。どちらの場合も、デプロイモードのプルダウンボックスが表示されるので、`true` に設定します。この設定は新たなセッションを開始したタイミングで有効になります。Seaside 全体でモードを切り替えるには `WAGlobalConfiguration setDeploymentMode` や、`WAGlobalConfiguration setDevelopmentMode` を実行します。

設定ツール自体も Seaside のアプリケーションです。そのため同じように Web ブラウザから各種の設定ができます。もしも誤って “`config`” のアプリケーションを削除してしまった場合は、`WADispatcherEditor initialize` とすると元に戻ります。



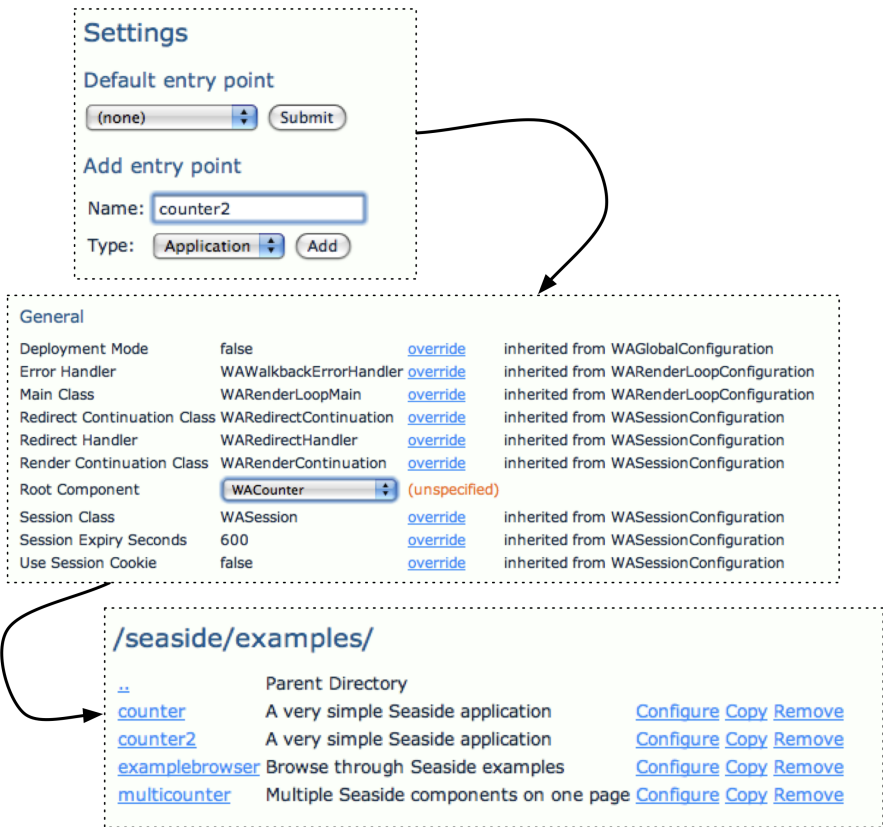


Figure 12.6: 新たなアプリケーションの設定

### 12.3 Seaside のコンポーネント

前の節で示したように、Seaside のアプリケーションはコンポーネント から成り立っています。Hello World のコンポーネントを作成し、Seaside の仕組みをさらに詳しく見てみましょう。

コンポーネントを定義するには WAComponent を直接あるいは間接的に継承するようにします (図 12.8)。

🌀 WAComponentを継承した WAHelloWorldクラスを定義してみましょう。

コンポーネントでは自身の表示の仕方を定める必要があります。このためには renderContentOn: をオーバーライドします。このメソッドには、引数として XHTML の生成方法を知っている WAHtmlCanvas のインスタンスが渡されます。

rendering プロトコルに以下のようにメソッドを定義してみましょう。

```
WAHelloWorld»renderContentOn: html
html text: 'hello world'
```

次に Seaside に対して、コンポーネントがスタンドアローンのアプリケーションであるという宣言を行います。

WAHelloWorld のクラス側に以下のメソッドを定義しましょう。

```
WAHelloWorld class»canBeRoot
↑ true
```

これでほとんどできあがりです。

Web ブラウザで <http://localhost:8080/seaside/config> を開き、“hello” のエントリーポイントを追加して、ルート のコンポーネントを WAHelloWorld に設定します。そして <http://localhost:8080/seaside/hello> を開くと、図 12.7 のように表示されます!

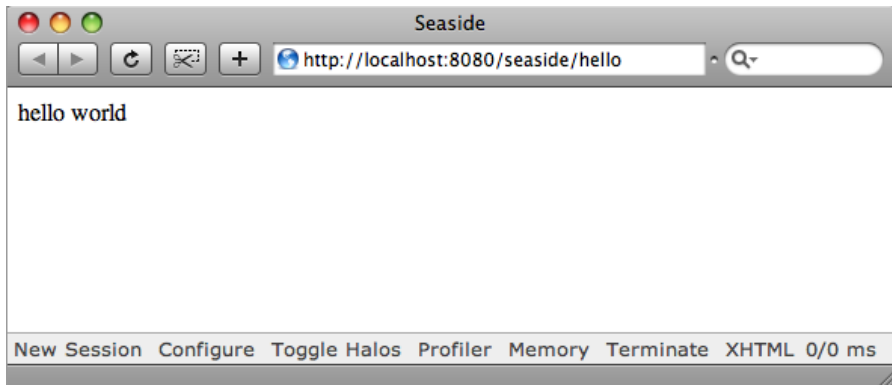


Figure 12.7: Seaside の “Hello World”

## 状態のバックトラックと “Counter” アプリケーション

“hello world” に比べると「カウンター」のアプリケーションは、ほんの少しだけ難しいものになっています。

WACounter はスタンドアローンのアプリケーションなので WACounter class では canBeRoot を実装して、true を返す必要があります。さらに、図 12.8 にあるように、アプリケーションとしての登録をクラスメソッド initialize の中で行っています。

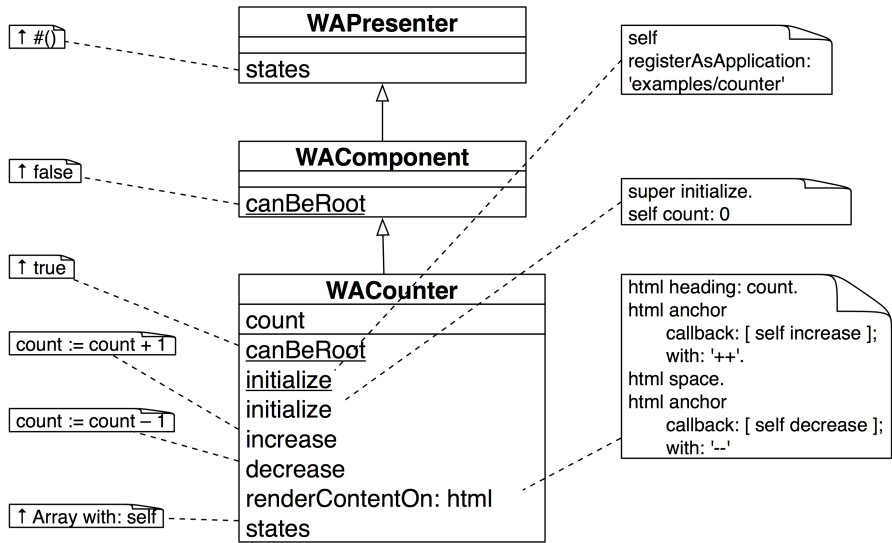


Figure 12.8: *counter* アプリケーションは `WACounter` クラスにより実装されています。アンダーラインの引いてあるメソッドはクラスメソッドで、普通のテキストで書かれたメソッドはインスタンスメソッドです。

`WACounter` には `increase` と `decrease` メソッドがあります。それぞれ Web ページで `++`、`--` のリンクをクリックしたときに呼び出されます。カウンターの値を保持するため、インスタンス変数として `count` が定義してあります。「戻る」ボタンを押してもカウンターの値がブラウザの値とずれないようにするには、`WACounter` の状態も元に戻せるようにしなければなりません。Seaside は状態を元に戻すためのメカニズムを備えていますが、アプリケーションのどの状態に対して行うかは開発者が指定する必要があります。

インスタンスメソッドの `states` を実装すると、どの状態を元に戻す対象にするかを定めることができます。`states` では対象とすべきオブジェクト群を配列の形で返すようにします。カウンターアプリケーションの場合、`Array with: self` としています。こうすると `WACounter` のオブジェクト自身が元に戻す対象として Seaside に登録されることとなります。

**注意** 状態を戻すオブジェクトを宣言する場合、ちょっとした注意点ががあります。Seaside は `states` 配列で宣言されたオブジェクトの *copy* を取ることで、過去の状態を保持します。この実際の処理は `WASnapshot` によって行われます。`WASnapshot` は `IdentityDictionary` のサブクラスで、キーとして元に戻す対象となるオブジェクト群、値として各オブジェクトのシャローコピーを持つ辞書です。特定のスナップショットにアプリケーションの状態を戻す必要があるときには、辞書に格納された値を用いてオブジェクトの変数の値を上書きします。

ここで注意すべき点があります。先ほどの `WACounter` の場合、バックトラックの対象は、インスタンス変数 `count` の値だけでよいのではと思うかもしれませんが、しかし、`states` メソッドで `Array with: count` のように書いてもうまく動作しません。これは `count` の中身が整数で、整数は不変のオブジェクトだからです。`increase` や `decrease` メソッドでは、整数オブジェクトの状態を 0 から 1 に変えたり、3 から 2 に変えたりしているわけではありません。その代わりに、`count` 変数の中身が代入されて別の整数オブジェクトに置き換えられているのです。整数が増えたり減ったりするたびに、`count` が指すオブジェクトは別のものに成り代わっています。このため、`WACounter»states` では `Array with: self` と書く必要があるのです。`WACounter` が以前の状態に戻るときには、各インスタンス変数の値がコピーされた `WACounter` から復元されます。これで `count` の値は正しく戻るようになります。

## 12.4 XHTML のレンダリング

Web アプリケーションでは、Web ページを動的に生成して表示 (レンダリング) を行います。12.3 節の節で述べたように、`Seaside` のコンポーネントは、それぞれが自身をレンダリングするようになっています。仕組みを知るために、カウンターのコンポーネントがどのようにレンダリングを行っているか、見ていきましょう。

### カウンターのレンダリング

カウンターのレンダリングは比較的単純です。図 12.8 のコードを見てください。カウンターの現在の値が XHTML の見出しとして表示されます。値の増減の操作はアンカー (リンク) として実装され、コールバックがブロックの形で仕掛けられています。アンカーをクリックすると `increase` や `decrease` がカウンターオブジェクトに対して送られるようになっています。

レンダリングで送るメッセージ群についてはすぐ後で詳しく解説しますが、その前にマルチカウンターの方もどうなっているか見てみましょう。

### Counter から MultiCounter へ

まず図 12.9 で示すように、`WAMultiCounter` もスタンドアローンのアプリケーションなので、`canBeRoot` を実装して `true` を返すようにしています。

マルチカウンターのコンポーネントは、サブコンポーネントを含んだコンポジットとなっています。そのため `children` メソッドを実装して、中に含まれるコンポーネント群を配列の形で返しています。

レンダリングでは、サブコンポーネント群を、水平線で区切りを入れて表示するように書いています。そのほか初期化を行うためのインスタンスメソッ

ド、クラスメソッドがありますが、基本的にはこれだけでよいのです!

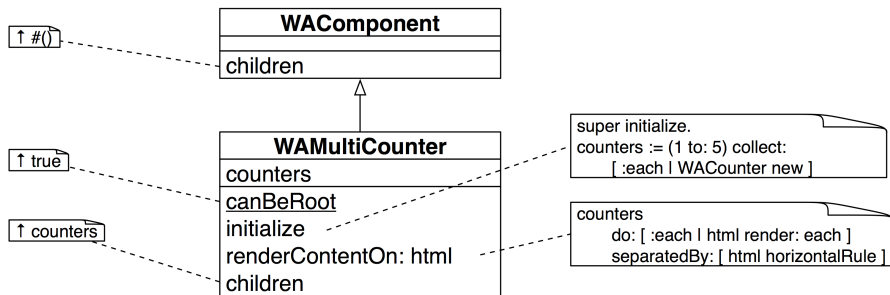


Figure 12.9: WAMultiCounter

## レンダリングの詳細

今までの例で見てきたように、Seaside では Web ページの生成にテンプレートを使いません。その代わりに、Smalltalk プログラムによって XHTML を生成します。Seaside のコンポーネントは `renderContentOn:` をオーバーライドする決まりになっています。このメソッドはコンポーネントのレンダリングのときにフレームワークの側から呼び出されます。`renderContentOn:` は引数を取らず、`html` キャンバス オブジェクトが渡されます。これを使ってコンポーネントはレンダリングを行います。慣習によりキャンバスの引数は `html` という名前になっています。`html` キャンバスは Morphic や別の GUI フレームワークでのキャンバスに似たものです。デバイス固有となる描画処理の詳細を、隠す役割を持っています。

基本的なレンダリングメッセージをいくつか挙げてみます。

```

html text: 'hello world'. "テキスト の表示"
html html: '&ndash;'. "XHTMLの直接指定"
html render: 1. "オブジェクトのレンダリング"
  
```

任意のオブジェクトを表示するには `render: anObject` を使うことができます。通常はサブコンポーネントの表示に使われます。引数となった `anObject` には `renderContentOn:` のメッセージが内部的に送られます。これはマルチカウンターの例で使いました (図 12.9 を参照)。

## ブラシの利用

キャンバス上で描画を行うために、たくさんのブラシを使うことができます。XHTML の各要素 (段落、テーブル、リストなど) に対応したブラシがあります。ブラシやそれに関連する便利メソッドを知るには、`WACanvas` とそのサブ

クラスをブラウズするとよいでしょう。renderContentOn: の引数は、WACanvas を継承した WARenderCanvas のインスタンスになっています。

既にカウンターとマルチカウンターの例で、以下のブラシを使いました。

```
html horizontalRule.
```

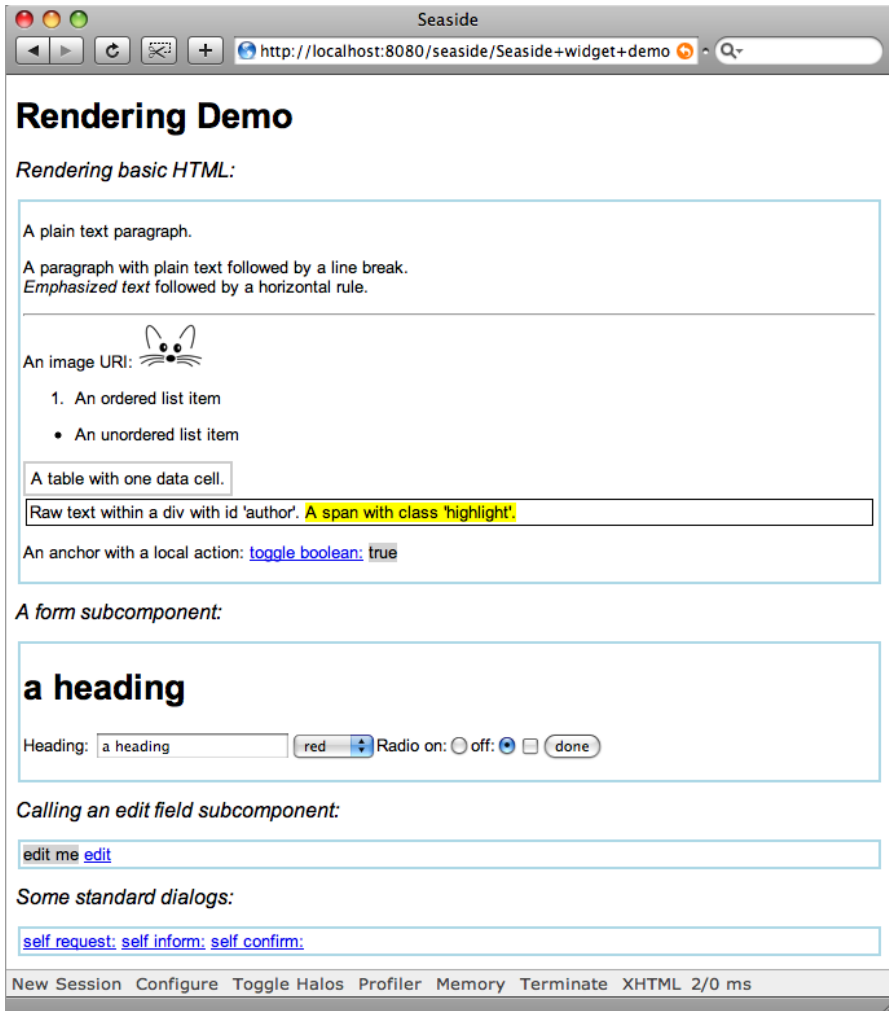


Figure 12.10: レンダリングのデモ

図 12.10 は、Seaside が提供する基本的なブラシをいろいろと使ってみた例です。<sup>5</sup>SeasideDemo のルートコンポーネントは単にサブコンポーネントの表示

<sup>5</sup>Method 12.1 のソースコードは <http://www.squeaksource.com/PharoByExample> プロジェクトの PBE

を行うだけです。サブコンポーネントは Method 12.1 を読むとわかりますが、SeasideHtmlDemo、SeasideFormDemo、SeasideEditCallDemo、SeasideDialogDemo となっています。

#### Method 12.1: SeasideDemo»renderContentOn:

```
SeasideDemo»renderContentOn: html
  html heading: 'Rendering Demo'.
  html heading
    level: 2;
    with: 'Rendering basic HTML: '.
  html div
    class: 'subcomponent';
    with: htmlDemo.
  "残りをレンダリング..."
```

コンポーネントはサブコンポーネントを返すというのを覚えているでしょうか。Seaside ではレンダリングのたびにサブコンポーネントを再利用します。

#### SeasideDemo»children

```
↑{ htmlDemo . formDemo . editDemo . dialogDemo }
```

上記のコードでは heading ブラシの使い方が少し違っていることに注意してください。heading: は、ブラシの生成と同時にテキストをメッセージの引数でセットする書き方になっています。次に出てくるのは、heading メッセージを送ってブラシを生成した後で、様々なプロパティをカスケードでセットしていくというものです。ブラシはたいていこの 2 種類の書き方ができるようになっています。

もしも カスケード を使って、ブラシに with: を送るなら、with: は必ず最後に書く必要があります。with: によってブラシのレンダリングが始まるからです。

Method 12.1 のメソッドでは、最初の見出しはデフォルトでレベル 1 になります。次の見出しはレベル 2 を明示的に指定しています。サブコンポーネントは XHTML で div の中に囲まれる形になります。div には “subcomponent” という CSS のクラスが付与されます。(CSS の使い方については 12.5 節で解説します) with: メッセージの引数は、文字列とは限りません。この例ではコンポーネントを渡しています。次の例でも出てきますが、レンダリングの操作を記述したブロックでもよいのです。

SeasideHtmlDemo コンポーネントは基本的なブラシをいろいろと使ったデモになっています。コードを見れば大体の利用の仕方がわかるでしょう。

#### SeasideHtmlDemo»renderContentOn: html

-SeasideDemo パッケージ内にあります。

```
self renderParagraphsOn: html.
self renderListsAndTablesOn: html.
self renderDivsAndSpansOn: html.
self renderLinkWithCallbackOn: html
```

上のように、レンダリングのメソッドが長くなった場合には細かいメソッド群に分割することがよく行われます。

一つのメソッド内に長々とレンダリング処理を書くやり方はお勧めできません。render\*On:という形でヘルパーメソッドに分割しましょう。そうしたメソッド群は *rendering* メソッドカテゴリに置いておくようにします。また、コンポーネントに renderContentOn:を直接送ってはいけません。代わりに render:を送るようにしてください。

さらにコードを見ていきます。最初のヘルパーメソッド SeasideHtmlDemo»renderParagraphsOn:は、XHTML の段落、通常のテキスト、強調表示されたテキスト、画像の表示の例になっています。Seaside では単純な要素なら、ただ文字列を指定するだけでレンダリングできますし、複雑なものであればブロックを使います。レンダリングのコードを書くときの指針として覚えておくといでしょう。

```
SeasideHtmlDemo»renderParagraphsOn: html
html paragraph: 'A plain text paragraph.'.
html paragraph: [
  html
  text: 'A paragraph with plain text followed by a line break. ';
  break;
  emphasis: 'Emphasized text ';
  text: 'followed by a horizontal rule.';
  horizontalRule;
  text: 'An image URI: '.
  html image
  url: self squeakImageUrl;
  width: '50']
```

次のヘルパーメソッド SeasideHtmlDemo»renderListsAndTablesOn:は、リストとテーブルの表示例になっています。テーブルではネストしたブロックを使い、列と列内のセルを表示します。

```
SeasideHtmlDemo»renderListsAndTablesOn: html
html orderedList: [
  html listItem: 'An ordered list item'].
html unorderedList: [
  html listItem: 'An unordered list item'].
html table: [
```



```
html tableRow: [
  html tableData: 'A table with one data cell.']]
```

次の例は、*div* や *span* の要素に、CSS の *class* や *id* 属性を指定するやり方を示しています。もちろん *class*: や *id*: といったメッセージは、*div span* 以外のブラシにも送ることができます。SeasideDemoWidget»style のメソッドで、XHTML の要素が実際にどのようなスタイルで表示されるかを定義しています (12.5 節を参照)。

```
SeasideHtmlDemo»renderDivsAndSpansOn: html
html div
  id: 'author';
  with: [
    html text: 'Raw text within a div with id "author". '.
    html span
      class: 'highlight';
      with: 'A span with class "highlight"'.]
```

最後に、リンクの簡単な例を見てみましょう。アンカー (リンク) にコールバックを設定しています。リンクをクリックするたびに、文字列が “true” と “false” に切り替わりますが、これはインスタンス変数の *toggleValue* の値がコールバックの実行により変化するからです。

```
SeasideHtmlDemo»renderLinkWithCallbackOn: html
html paragraph: [
  html text: 'An anchor with a local action: '.
  html span with: [
    html anchor
      callback: [toggleValue := toggleValue not];
      with: 'toggle boolean:'].
  html space.
  html span
    class: 'boolean';
    with: toggleValue ]
```

アプリケーションの状態を変えるような処理は、コールバック内を書くようにします。レンダリング中に、直接そのようなコードを書くべきではありません。

## フォーム

フォームも今までの例と同様のやり方で表示できます。図 12.10 の SeasideFormDemo コンポーネントのコードを示します。

```
SeasideFormDemo»renderContentOn: html
```

```
| radioGroup |
html heading: heading.
html form: [
  html span: 'Heading: '.
  html textInput on: #heading of: self.
  html select
    list: self colors;
    on: #color of: self.
  radioGroup := html radioGroup.
  html text: 'Radio on:'.
  radioGroup radioButton
    selected: radioOn;
    callback: [radioOn := true].
  html text: 'off:'.
  radioGroup radioButton
    selected: radioOn not;
    callback: [radioOn := false].
  html checkbox on: #checked of: self.
  html submitButton
    text: 'done' ]
```

フォーム要素は複雑なので、ブロックを使ってレンダリングします。状態の変更に関わるコードはすべてコールバック内で書かれており、レンダリング部分には書かれないことに注意してください。

on:of: メッセージは Seaside の特徴の一つとも言えるもので、特に説明が必要でしょう。上の例ではテキスト入力フィールドに、自身のインスタンス変数 `heading` を対応させるために使われています。アンカーやボタンもこのメッセージに答えられます。最初の引数はインスタンス変数の名前です。2 番目の引数はそのインスタンス変数を保持するオブジェクトを指定します。通常の名前付けルールに従って、インスタンス変数のアクセサ (`heading`) とミューテータ (`heading:.`) がクラスに定義されていることが前提となっています。上記のテキスト入力フィールドの例では、デフォルト値の設定や、値を更新するためのコールバックを書く手間を省くのに役立っています。on: #heading of: self を使うことで `heading` 変数の値はユーザの入力に応じて自動的に更新されます。

すぐ後にコンボボックスで `color` 変数を指定した別の例があります。その次のチェックボックスの例では、`checked` 変数の値が更新されるようになっています。Seaside の機能テストのコードを見ると、さらに様々なフォームの例が見つかります。Seaside-Tests-Functional のクラスカテゴリを見てみましょう。または、Web ブラウザで <http://localhost:8080/seaside/tests/alltests> を開き、WAIInputTest を選んで  ボタンを押すと、フォームのほとんどの機能を知ることができます。<sup>6</sup>

Web ブラウザの  ボタンを押せば、対応するソースコードにすぐにアクセスできるという Seaside の機能を、ここで活用してください。

<sup>6</sup>訳注: Seaside 3.0 では、URL が <http://localhost:8080/tests/functional/WAIInputPostFunctionalTest> に、メニューが [WAIInputGetFunctionalTest\(WAIInputPostFunctionalTest\)](#) に変わっています。

## 12.5 CSS: カスケーディング・スタイル・シート

CSS (カスケーディング・スタイル・シート)<sup>7</sup>は、Web アプリケーションで、内容と見栄えを分離する標準的な方法になっています。Seaside では、見栄えを意識してレンダリングのコードが煩雑になるのを、CSS の技術を使って防いでいます。

`style` というメソッドをコンポーネントに定義することで、使用するスタイルシートを決めることができます。メソッドの中身は、コンポーネントが使うスタイルについて定めた単なる CSS の文字列です。Web ページに配置されるコンポーネントのスタイルシートは結合されて一つになりますが、コンポーネントは自身のスタイルを自ら定義できます。抽象クラスを定義して、アプリケーションで使う共通のスタイルをメソッドとして定義しておくのもよいでしょう。

とはいえデプロイ時には、外部ファイルとしてスタイルシートを用意することも広く行われます。こうするとコンポーネントの機能的な部分と、見栄えの分離がさらに進み、完全に分離されることになります。( `WFileLibrary` クラスを使うと、別途サーバーを用意せずに、外部のファイルを利用できます)。

CSS をよく知っているのでしたら、これで説明は終わりです。念のため、以後は簡単な CSS の紹介を書いておきます。

CSS を使うと、段落やテキストの表示属性を直接 Web ページに埋め込む代わりに、要素の分類や表示にまつわるあれこれを独立したスタイルシートに定義できます。段落に該当する要素が `div` で、インラインのテキスト要素は `span` と呼ばれます。例えば自分で `"highlight"` といった形で、要素に名前を付け、ハイライトされるテキストを定義することができます (例を参照のこと)。スタイルシートの中では、ハイライトがどのように表示されるべきかを記述します。

基本的に CSS は、XHTML の要素をどのようなフォーマットで表示するかについてのルールを記述したものです。ルールは二つの部分から成り立っています。セレクトと呼ばれる部分では、どの XHTML 要素にルールを適用するかを指定し、宣言の部分では、その要素に適用される属性の値を定めます。

図 12.11 は、図 12.10 で使われているスタイルシートのメソッドを示したものです。最初のルールは `body` 要素で使われるフォントを指定しています。さらに、レベル 2 の見出し (`h2`)、テーブル (`table`)、セル (`td`) についての定義が続きます。

残りのルールは、特定の `"class"` や `"id"` を持つ XHTML 要素についてです。クラス属性の CSS のセレクトは `"."` で始まり、`id` 属性のセレクトは `"#"` で始まります。`class` と `id` の違いは、ページ内で `id` 属性の要素は一つしか存在できないのに対して (`ID` のため)、クラス属性の要素は複数あってもよいということです。`highlight` のようなクラス属性は、ページ内に何度出てきてもかまいません。`id` 属性はページ内で唯一の要素、例えばメニュー、更新日、作者の名前などで使われます。また、XHTML の各要素には、複数のクラス属性が指定できます。その場合は指定された順番に沿って、表示の属性が適用されていきます。

<sup>7</sup><http://www.w3.org/Style/CSS/>

```
SeasideDemoWidget»style
↑
body {
  font: 10pt Arial, Helvetica, sans-serif, Times New Roman;
}
h2 {
  font-size: 12pt;
  font-weight: normal;
  font-style: italic;
}
table { border-collapse: collapse; }
td {
  border: 2px solid #CCCCCC;
  padding: 4px;
}
#author {
  border: 1px solid black;
  padding: 2px;
  margin: 2px;
}
.subcomponent {
  border: 2px solid lightblue;
  padding: 2px;
  margin: 2px;
}
.highlight { background-color: yellow; }
.boolean { background-color: lightgrey; }
.field { background-color: lightgrey; }
,
```

Figure 12.11: SeasideDemoWidget 共通スタイルシート

セレクトタは組み合わせることができます。div.subcomponent というセレクトタは、div で、クラス属性が “subcomponent” の要素のみを選択することになります。

あまり必要になることはありませんが、ネストした要素を指定することも可能です。例えば “p span” は、段落内 (p) の span を選択します。div 内の span にマッチすることはありません。

CSS については多くの本や Web ページがあります。CSS でどれだけのことができるかを知るには、CSS Zen Garden<sup>8</sup>のサイトを見てみるとよいでしょう。CSS のスタイルシートを変えるだけで、同じ内容の文書をまったく異なる見栄えにできるということがわかります。

---

<sup>8</sup><http://www.csszengarden.com/>

## 12.6 制御フローの管理

Seaside では、複雑なフローを持つ Web アプリケーションを、非常に簡単に開発できるようになっており、二つの基本的な仕組みが提供されています。

1. コンポーネントは、`caller call: callee`のように書くことで、他のコンポーネントを呼び出すことができます。呼び出し側は一時的に表示されなくなり、呼ばれた側に制御が移ります。`answer:`によって、制御を元に戻すことができます。呼び出し側は通常 `self` ですが、現在表示されているコンポーネントであればどれでもかまいません。
2. 全体のワークフローは タスク の仕組みで定義できます。タスクは、(`WAComponent` でなく) `WATask` を継承する特別なコンポーネントです。タスクは内部にコンポーネントを持たず、`renderContentOn:` を実装する代わりに、`go` メソッドを定義し、コンポーネントを順番に `call:` する形で全体の流れを書いていきます。

### call: と answer:

`call:` と `answer:` は単純なダイアログ形式の UI を実現するために使われます。

`call:` と `answer:` を使った簡単な例は、図 12.10 のレンダリングのデモに含まれています。SeasideEditCallDemo のコンポーネントは、テキストフィールドと `edit` リンクを表示しています。`edit` のリンクにはコールバックが仕掛けられていて、テキストフィールドの値で初期化した上で SeasideEditAnswerDemo のインスタンスを呼び出すようになっています。コールバックではさらに、呼び出したコンポーネントから受け取った値を、テキストフィールドに反映させるようになっています。

(以下のコード例では目立たせるために `call:` と `answer:` のメッセージにアンダーラインを引いています。)

```
SeasideEditCallDemo»renderContentOn: html
html span
  class: 'field';
  with: self text.
html space.
html anchor
  callback: [self text: (self call: (SeasideEditAnswerDemo new text: self text))];
  with: 'edit'
```

次に表示すべき Web ページ全体を意識する部分がまったくない、ということがエレガントです。実行すると SeasideEditCallDemo コンポーネントによって表示されている部分が SeasideEditAnswerDemo コンポーネントに置き換わり、ページ全体の表示が行われます。外側のコンポーネントや隣接するコンポーネントはそのままです。

call: や answer: はレンダリング中に直接記述してはいけません。コールバック や、タスクの go メソッドの中で書くようにします。

SeasideEditAnswerDemo コンポーネントも非常にシンプルです。テキストフィールドの初期値を表示し、送信ボタンのコールバックで、更新されたテキストフィールドの値を返すように書いてあるだけです。

```
SeasideEditAnswerDemo»renderContentOn: html
```

```
html form: [
  html textInput
    on: #text of: self.
  html submitButton
    callback: [ self answer: self text ];
    text: 'ok'.
]
```

わずかこれだけでページの遷移が実現できます。

Seaside は適切なコンポーネントに制御を移し、レンダリングを自動的に行ってくれます。Web ブラウザの「戻る」ボタンも問題なく動作します。(ただし副作用をロールバックできるようにするにはもう少しコードが必要になります)。

## 便利なメソッド群

call/answer を使ったやりとりには非常によく出てくるパターンがあります。そのため Seaside は、SeasideEditAnswerDemo のような些細なコンポーネントをわざわざ書かずにすむように、便利なメソッドを提供しています。便利メソッドによって生成されるダイアログを図 12.12 に示しました。これらの便利メソッドの動作は SeasideDialogDemo»renderContentOn: で確認できます。

request: メッセージは、編集可能なテキストフィールド付きのコンポーネントを開きます。ユーザが入力したテキストが呼び出し側に返るようになっていきます。ラベルやデフォルト値も指定できます。

```
SeasideDialogDemo»renderContentOn: html
```

```
html anchor
  callback: [ self request: 'edit this' label: 'done' default: 'some text' ];
  with: 'self request:'.
```

```
...
```

inform: メッセージは、指定した文字列を表示するコンポーネントを開きます。“ok” ボタンを押すとコンポーネントが閉じ、制御が呼び出し側に戻ります。このコンポーネントは self を返すだけです。

```
...
html space.
html anchor
  callback: [ self inform: 'yes!' ];
  with: 'self inform:'.
...
```

confirm: メッセージを使うと “Yes”/“No” の質問ダイアログとなり、ユーザはどちらかを選ぶことになります。真偽値を呼び出し側に返すので、それによって次の動作を決めることができます。

```
...
html space.
html anchor
  callback: [
    (self confirm: 'Are you happy?')
    ifTrue: [ self inform: ':-)' ]
    ifFalse: [ self inform: ':-(' ]
  ];
  with: 'self confirm:'.
```

WComponent の *convenience* プロトコルには、chooseFrom:caption: のような役に立つメソッドがさらに定義してあります。

### タスク

WATask を継承すると、タスク を定義できます。タスク自身はレンダリングを行わず、単に制御フローに沿って他のコンポーネントを call: してだけです。

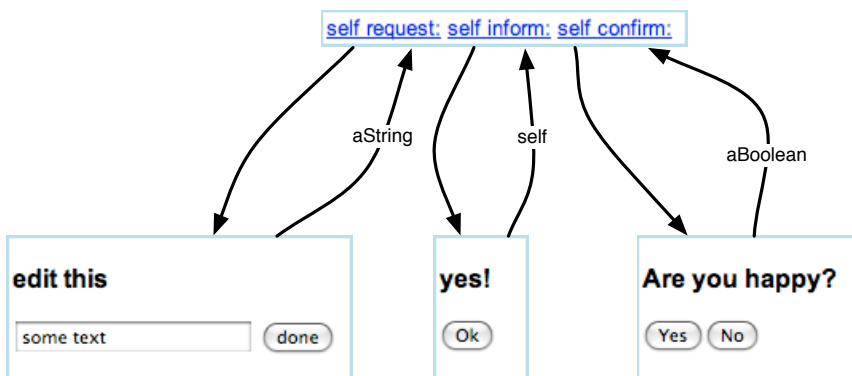


Figure 12.12: 標準ダイアログ

go に、制御の流れを書いています。

*Seaside-Tests-Functional* クラスカテゴリに定義されている `WAConvenienceTest` は、タスクの簡単な使用例になっています。動作を確認するには、Web ブラウザで `http://localhost:8080/seaside/tests/alltests` の URL を開き、`WAConvenienceTest` を選択して `Restart` ボタンを押します。<sup>9</sup>

```
WAConvenienceTest>go
[ self chooseCheese.
  self confirmCheese ] whileFalse.
self informCheese
```

このタスクでは三つのコンポーネントの呼び出しを順次行っています。まず `chooseFrom: caption:` の便利メソッドで、`WAChoiceDialog` のダイアログが開きます。ここでユーザは好きなチーズを選択します。

```
WAConvenienceTest>chooseCheese
cheese := self
  chooseFrom: #'Greyerzer' 'Tilsiter' 'Sbrinz')
  caption: 'What's your favorite Cheese?'.
cheese isNil ifTrue: [ self chooseCheese ]
```

次に `WAYesOrNoDialog` でユーザの選択を確認します (ダイアログは `confirm:` で生成されたものです)。

```
WAConvenienceTest>confirmCheese
↑self confirm: 'Is ', cheese, ' your favorite cheese?'
```

最後に便利メソッド `inform:` によって、`WAFormDialog` が呼び出されます。

```
WAConvenienceTest>informCheese
self inform: 'Your favorite cheese is ', cheese, '.'
```

生成されるダイアログを図 12.13 に示します。

## トランザクション

10

12.3 節の節で、*Seaside* ではバックトラック対象となるオブジェクトを登録することで、コンポーネントの状態を Web ページごとに記録できるという話をしました。コンポーネント側で必要なのは、`states` メソッドを定義し、バックトラック対象のオブジェクトを配列の形で返すことだけでした。

<sup>9</sup>訳注: *Seaside* 3.0 では、`WAConvenienceTest` ではなく、`WAFlowConvenienceFunctionalTest` に変わっています。動作確認では、`http://localhost:8080/tests/functional` を開き、`WAFlowConvenienceFunctionalTest` を選びます。

<sup>10</sup>訳注: トランザクションは *Seaside* 3.0 で未サポートとなったため、この節の内容は意味を持たなくなりました。



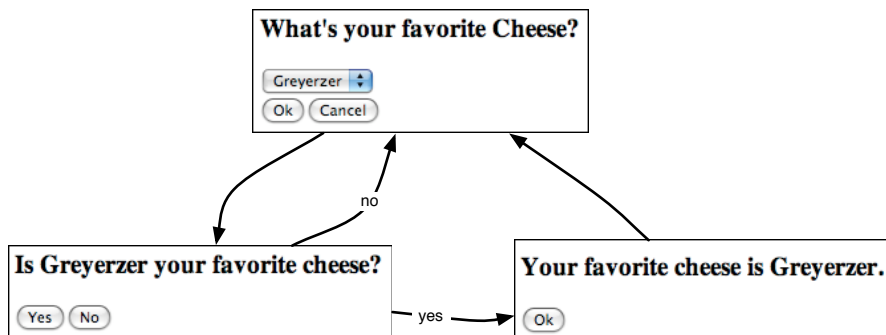


Figure 12.13: 単純なタスクの例

しかし時にはバックトラックをしたくないときもあります。ユーザが行った操作を、意図せず取り消してしまうことを防ぎたいときがあるのです。これは「ショッピングカート問題」などと呼ばれます。ユーザがショッピングカートの精算をすませた後は、Web ブラウザのボタンで戻って、カートに商品を追加できてしまっはまずいのです。

Seaside では、一連の操作をひとまとめにしてタスク内でトランザクションとして定義することにより、この問題を解決します。トランザクション内ではバックトラックができますが、トランザクションが終了すると戻ることはできません。トランザクションを越えて戻ろうとしても、前のページは無効になります。無効なページの表示時に Seaside が警告を出し、直近の有効なページへと自動的にリダイレクトします。

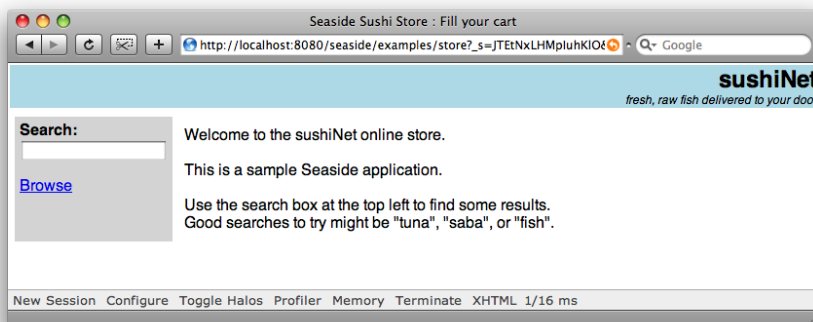


Figure 12.14: 寿司屋デモ

寿司屋デモ は、Seaside の多くの特徴を示すサンプルアプリケーションで

す。もちろんトランザクションも使われています。このサンプルは Seaside に含まれており、Web ブラウザで <http://localhost:8080/seaside/examples/store> の URL を指定すると起動します。<sup>11</sup>


寿司屋デモのワークフローは以下のようになっています。

1. サイトを訪れる
2. 寿司をブラウズ、または検索する
3. ショッピングカートに寿司を追加する
4. 注文を開始する
5. カートの内容を確認する
6. 届け先の住所を入力する
7. 住所を確認する
8. 支払い情報を入力する
9. 寿司を待つ

ハローを表示してみると、トップレベルのコンポーネントは WASTore となっていることがわかります。このコンポーネントはタイトルバーの表示を行っているだけで、WASToreTask のインスタンスである task にレンダリングを委譲しています。

```
WASTore>renderContentOn: html
"... タイトル部分のレンダリング..."
html div id: 'body'; with: task
```

ワークフローの順番を定義しているのが WASToreTask です。一度サブミットした情報を、ユーザが戻って変更できないようにしていることが重要です。

 いくつか寿司を購入した後で、カートの中にさらに寿司を追加するため、Web ブラウザの「戻る」ボタンを押してみましょう。“*That page has expired.*”と出て追加できません。

Seaside では、プログラマがワークフローの特定部分をトランザクションとして定義できるようになっています。トランザクションが終了すると、ユーザはそこに戻って操作をやり直すことはできません。トランザクションを定義するには、`isolate:` メッセージをタスクに送ります。引数でトランザクション部分ブロックで指定します。寿司屋デモでは以下のようにしています。

```
WASToreTask>go
| shipping billing creditCard |
cart := WASToreCart new.
self isolate:
```

<sup>11</sup>もしも同梱されていない場合は SqueakSource のページ (<http://www.squeaksource.com/SeasideExamples/>) からロードできます。

```

[[self fillCart.
 self confirmContentsOfCart]
 whileFalse].

self isolate:
 [shipping := self getShippingAddress.
 billing := (self useAsBillingAddress: shipping)
  ifFalse: [self getBillingAddress]
  ifTrue: [shipping].
 creditCard := self getPaymentInfo.
 self shipTo: shipping billTo: billing payWith: creditCard].

self displayConfirmation.

```


ここでは二つのトランザクションが定義されていることがわかります。最初のトランザクションは、カートに寿司を入れて内容を確認する部分です。(fillCart等のヘルパーメソッドは、内部でサブコンポーネントを生成したり呼び出したりしています。)一度カートの内容を確認してしまうと戻ることはできません。2番目のトランザクションは、住所と支払い方法を入れる部分です。支払い方法を確定するまでは、このトランザクション内で戻ることもできます。しかし、トランザクションを終えてしまうと、戻ろうとしても常に警告が出て失敗します。

トランザクションをネストすることもできます。WANestedTransactionで簡単なデモを見ることができます。最初のisolate:のブロック内に、さらに別のisolate:ブロックが含まれています。

```

WANestedTransaction»go
 self inform: 'Before parent txn'.
 self isolate:
  [self inform: 'Inside parent txn'.
   self isolate: [self inform: 'Inside child txn'].
   self inform: 'Outside child txn'].
 self inform: 'Outside parent txn'

```

 <http://localhost:8080/seaside/tests/alltests> を開いて **WATransactionTest** のメニューを選び、**Restart** ボタンを押してみましょう。親側と子側のトランザクション内で **OK** ボタンやブラウザの **戻る** ボタンを使い、行ったり来たりしてみましょう。トランザクション終了後は、戻って **OK** ボタンを押しても、エラーが出て終了後のページに必ずダイレクトされます。

## 12.7 サンプルアプリ作成によるチュートリアル

ここでは Seaside のアプリケーションを一から作成してみることにします。<sup>12</sup>RPN (逆ポーランド記法) 計算機を Seaside のアプリケーションとして作ることにしましょう。単純なスタックマシンを実装して、モデル部分で使うことにします。この計算機アプリでは二種類の表示方法を提供します。一つ目はスタックの先頭の値のみを表示するタイプで、もう一つはスタックの内容をすべて表示するタイプです。図 12.15 は、二つの表示方法を切り替える様子を表しています。

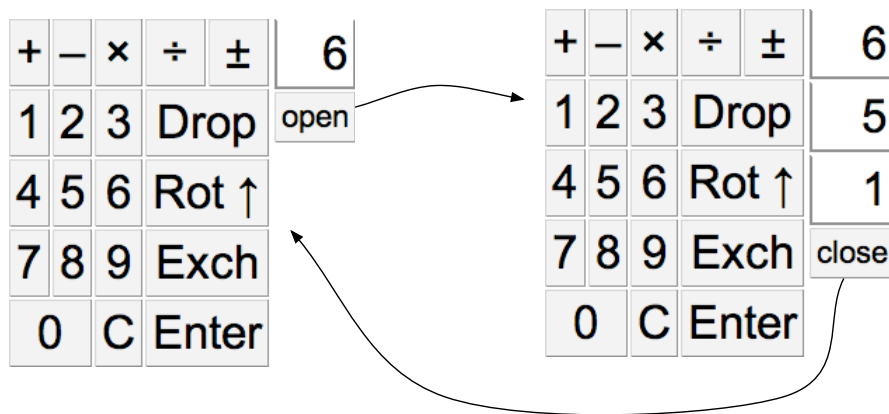



Figure 12.15: スタックマシンを使った RPN 計算機

スタックマシンの実装を、テストを書きながら始めていきましょう。

 contents というインスタンス変数を持つ MyStackMachine というクラスを定義してください。contents は OrderedCollection で初期化します。

```
MyStackMachine»initialize
super initialize.
contents := OrderedCollection new.
```

スタックマシンは値のプッシュやポップをサポートしなければなりません。またスタックの先頭を見る機能や、積んだ値に対して加減乗除を計算する機能も必要です。

 スタック操作のテストを書いてから実装してみましょう。以下はテストのサンプルです。


<sup>12</sup>このチュートリアルには二、三時間かかります。すべてのソースコードを見たい場合は SqueakSource のプロジェクト <http://www.squeaksource.com/PharoByExample> から入手できます。PBE-SeasideRPN のパッケージを選んでロードします。クラス名を少し変えてあるので、すべて読み込んでも、本書に従ってアプリケーションを作っていくことができます。

```
MyStackMachineTest>>testDiv
```

```
stack
  push: 3;
  push: 4;
  div.
self assert: stack size = 1.
self assert: stack top = (4/3).
```

加減乗除では、事前にスタックに値が二つ積まれているかをチェックするヘルパーメソッドが必要でしょう。条件が整っていない場合はエラーとなるようにします。<sup>13</sup>これができてしまえば、各メソッドの実装はほぼ一行か二行ですみます。

デバッグしやすくするため、`MyStackMachine>>printOn:`も実装しておくといでしょう。インスペクタを使って、スタックマシンの状態を観察できるようになります。(ヒント: `contents`に表示を委譲してしまうのが楽です)。

 `dup` (先頭の値をコピーしてプッシュする)、`exch` (スタックの上位二つの値を入れ替える)、`rotUp` (スタックの内容を上下逆にする — 先頭の要素が最後になる)、を `MyStackMachine` に実装し、スタックマシンを完成させましょう。

スタックマシンの実装がすんだら、次はいよいよ `Seaside` 上に `RPN` 計算機を作っていきます。

クラスを五つ定義することにします:

- `MyRPNWidget` — `RPN` 計算機のコンポーネントで共通に使う、振る舞いや `CSS` を定義するための抽象クラスです。`WACComponent`のサブクラスで、以下の四つのクラスのスーパークラスになります。
- `MyCalculator` — ルートコンポーネントです。クラスメソッドでアプリケーションの登録を行います。サブコンポーネントを生成し、レンダリングを行わせます。またバックトラック用に状態の登録も行います。
- `MyKeypad` — 計算機のキーボードを表示します。
- `MyDisplay` — スタックの先頭の値と、詳細なビューに切り替えるためのボタンを表示します。
- `MyDisplayStack` — スタックの詳細なビューと、元のビューに戻るためのボタンを表示します。`MyDisplay` のサブクラスです。

 `MyCalculator`クラスカテゴリに `MyRPNWidget`を定義してみましょう。アプリケーションで共通に使う `style` メソッドを書くようにします。

アプリケーション用に `CSS` を定義します。もっと凝ってみてもかまいません。

<sup>13</sup>事前条件を指定するには `Object>>assert:` を使うといでしょう。不正な状態でスタックマシンを使おうとしているときには `AssertionFailure`が起こるというわけです。

```
MyRPNWidget»style
  ↑ `table.keypad { float: left; }
td.key {
  border: 1px solid grey;
  background: lightgrey;
  padding: 4px;
  text-align: center;
}
table.stack { float: left; }
td.stackcell {
  border: 2px solid white;
  border-left-color: grey;
  border-right-color: grey;
  border-bottom-color: grey;
  padding: 4px;
  text-align: right;
}
td.small { font-size: 8pt; }
```

④ ルートコンポーネント `MyCalculator` を定義して、アプリケーションとして登録しましょう。(クラスメソッド `canBeRoot` と `initialize` を実装します) `MyCalculator` »`renderContentOn:` では、クラス名の表示などの簡単なレンダリングを書いて、*Web* ブラウザで表示されるかを確認してみましょう。

`MyCalculator` では、`MyStackMachine`、`MyKeypad`、`MyDisplay` のインスタンスを保持することにします。

④ `MyRPNWidget` のサブクラスとして `MyKeypad` と `MyDisplay` を定義しましょう。コンポーネントはどれもスタックマシンのインスタンスを共有するので、インスタンス変数 `stackMachine` と初期化用のメソッド `setMyStackMachine:` を `MyRPNWidget` に用意します。`MyCalculator` には `keypad` と `display` のインスタンス変数を定義し、`MyCalculator` »`initialize` で初期化することになります。(super `initialize` を送るのを忘れないように!)

④ `initialize` メソッドでは、共有する `stackMachine` を `keypad` と `display` に渡すようにします。レンダリングメソッド `MyCalculator` »`renderContentOn:` の実装は、単に `keypad` と `display` に委譲するだけです。サブコンポーネントを正しく表示するには、`MyCalculator` »`children` を実装して、`keypad` と `display` からなる配列を返すようにしなければなりません。`keypad` と `display` にもレンダリングメソッドを作りますが、中身は空にしておきます。まずは計算機がちゃんとサブコンポーネントを表示しているかを確認してみましょう。

次は `display` のレンダリングメソッドの中身を書き、スタックの先頭を表示できるようにします。

④ レンダリングでは、“`keypad`” という CSS クラス名のテーブルを書き、行には “`stackcell`” というクラス名のセルを一つだけ入れることにします。また、

スタックが空の場合、先頭に 0 を *push* するようにしておきます。(MyKeypad »ensureMyStackMachineNotEmpty というメソッドを定義して使うことにします)。さらに “keypad” のクラス名で空のテーブルを表示させます。これで、計算機は 0 を含んだ一つのセルを表示するようになります。ハロを表示させてみると 図 12.16 のようになるはずですよ。

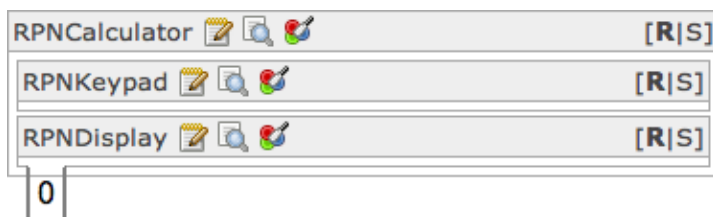


Figure 12.16: スタックの先頭を表示

今度はスタックとやりとりするための UI を書いていくことにしましょう。

🕒 まず、UI を簡単に書けるようにするために、以下のヘルパーメソッドを定義することにします。

```
MyKeypad»renderStackButton: text callback: aBlock colSpan: anInteger on: html
html tableData
  class: 'key';
  colSpan: anInteger;
  with:
    [html anchor
     callback: aBlock;
     with: [html html: text]]
```

```
MyKeypad»renderStackButton: text callback: aBlock on: html
self
  renderStackButton: text
  callback: aBlock
  colSpan: 1
  on: html
```

これらのメソッドは、キーパッドのボタンを表示しコールバックを定義するために使います。ボタンの中には複数のカラムを占めるものもありますが、基本はカラムを一つ使うことにします。


🕒 以下のように二つのヘルパーメソッドを使ってキーパッドのレンダリングを書いてみましょう。(ヒント: 数字キーと “Enter” キーから初めて、計算用のキーは後から付けると楽ですよ)。

```

MyKeypad»renderContentOn: html
self ensureStackMachineNotEmpty.
html table
  class: 'keypad';
  with: [
    html tableRow: [
      self renderStackButton: '+' callback: [self stackOp: #add] on: html.
      self renderStackButton: '&ndash;' callback: [self stackOp: #min] on: html.
      self renderStackButton: '&times;' callback: [self stackOp: #mul] on: html.
      self renderStackButton: '&divide;' callback: [self stackOp: #div] on: html.
      self renderStackButton: '&plusmn;' callback: [self stackOp: #neg] on: html ].
    html tableRow: [
      self renderStackButton: '1' callback: [self type: '1'] on: html.
      self renderStackButton: '2' callback: [self type: '2'] on: html.
      self renderStackButton: '3' callback: [self type: '3'] on: html.
      self renderStackButton: 'Drop' callback: [self stackOp: #pop]
        colSpan: 2 on: html ].
    "などなど"
    html tableRow: [
      self renderStackButton: '0' callback: [self type: '0'] colSpan: 2 on: html.
      self renderStackButton: 'C' callback: [self stackClearTop] on: html.
      self renderStackButton: 'Enter'
        callback: [self stackOp: #dup. self setClearMode]
        colSpan: 2 on: html ]]

```

キーパッドがブラウザできちんと表示されるか確認しましょう。当然ですが、クリックしてみてもまだ計算機としては動作しません。


 **MyKeypad»type:** を実装して、タイプした数字がスタックの先頭に反映されるようにしましょう。入力された値は文字列で得られるので、スタックの値を文字列にして連結し、整数にしてからスタックに戻す必要があります。具体的には以下のようなコードになります。

```

MyKeypad»type: aString
stackMachine push: (stackMachine pop asString, aString) asNumber.

```

数字キーを打つと、表示が更新されるようになりました。(MyStackMachine»pop でポップした値を返すようにきちんと実装していないと、動作しません!)

 次に **MyKeypad»stackOp:** を書くことにしましょう。以下はちょっとしたコツを使っています。

```

MyKeypad»stackOp: op
[ stackMachine perform: op ] on: AssertionFailure do: [].

```

ポイントはすべての操作が成功するわけではないということです。例えば二つの数値がスタックに積まれていないと、加算できません。当面は単にエ



ラーを無視するように書いておきます。もっと機能を充実させたい場合は、エラーハンドリングのブロックにユーザに注意を促すように書いていくことができます。

🔗 逆ポーランド 記法の計算機はこれで一応動くものになりました。数字キーを押した後に **Enter** キーを押して、いくつかの数値をプッシュしてみましょう。**+** を押して二つの数値が足されることを確認してください。

数値の入力が予想した動きと若干違うのではないのでしょうか。新たな数値を入力しているのか、既存の数値を更新しているのかをモードによって区別できないとまずいようです。

🔗 `MyKeypad>type:` を修正し、入力モードに応じて動作を変えるようにしてみましょう。`mode` というインスタンス変数を定義して、三つの値のいずれかを取るようになります。数値を入力しているときは `#typing` です。計算ボタンを押した後は、`#push` モードとなり、入力していた値を強制的にプッシュするようになります。**Enter** を押した後は、`#clear` となり、次の入力の前にスタックの先頭をクリアします。`type:` メソッドを書き直したのが以下のコードです。

```
MyKeypad>type: aString
  self inPushMode ifTrue: [
    stackMachine push: stackMachine top.
    self stackClearTop ].
  self inClearMode ifTrue: [ self stackClearTop ].
  stackMachine push: (stackMachine pop asString, aString) asNumber.
```

”

前よりは入力が便利になりましたが、スタックの先頭の値を見るができない点がまだ不満です。

🔗 `MyDisplay` のサブクラスとして `MyDisplayStack` を定義しましょう。`MyDisplay` のレンダリングメソッドにボタンの描画を追加して、コールバックで `MyDisplayStack` のインスタンスを `call:` するようにします。`HTML` のアンカーを使って、以下のように書けるでしょう。

```
html anchor
  callback: [ self call: (MyDisplayStack new setMyStackMachine: stackMachine)];
  with: 'open'
```

コールバックが呼ばれると、`MyDisplay` コンポーネントの表示は、一時的に `MyDisplayStack` コンポーネントへと切り替わり、スタックのすべての情報を表示するようになります。`self answer` によって、このコンポーネントが終了して閉じると、制御は元の `MyDisplay` へと戻ります。


🔗 `MyDisplayStack` のレンダリングメソッドを定義して、スタックのすべての内容を表示するようにしましょう。(スタックマシンの `contents` の値を返すアク

セサを用意するか、`MyStackMachine»do:`を定義して、スタックの値をイテレートできるようにするとよいでしょう)。“close” ボタンも付けることにします。コールバックでは単に `self answer` を実行するだけです。

```
html anchor
  callback: [ self answer];
  with: 'close'
```

これで `open` と `close` で表示を切り替えられるようになりました。

とはいえ、まだ考慮していなかったこともあります。いくつか操作をしてみた後で、Web ブラウザの「戻る」ボタンで戻って、操作してみるとどうなるでしょうか。(例えば `open` でスタックの全体を表示させ、`1`、`Enter` と 2 回入力し、`+` を押します。この時点でスタックの表示は “2” と “1” となっているはずですが。ここで「戻る」ボタンを押してみましょう。スタックは “1” を三つ表示した状態になり、`+` を押すと、“3” が表示されます。つまりバックトラックが動作していないのです。

 `MyCalculator»states` メソッドを実装し、スタックマシンの `contents` を返すようにしましょう。今度はちゃんとバックトラックが効いています。

さて、これで動くアプリケーションができあがりました。乾杯!

## 12.8 AJAX の利用

AJAX (Asynchronous JavaScript and XML) はクライアント側で JavaScript を活用することで、Web アプリケーションをさらにインタラクティブにする技術です。

よく知られた JavaScript のライブラリとしては Prototype (<http://www.prototypejs.org>) と script.aculo.us (<http://script.aculo.us>) があります。Prototype は、JavaScript を簡潔に書けるようにするためのフレームワークです。script.aculo.us は、Prototype 上に、アニメーションや、ドラッグ&ドロップなどをサポートする機能を追加したものです。Seaside では “Scriptaculous” というパッケージを通じて、これらのフレームワークを利用することができます。

Seaside のインストールイメージには、Scriptaculous パッケージが最初から入っています。最新のバージョンは <http://www.squeaksource.com/Seaside> から入手可能です。<http://scriptaculous.seasidehosting.st> でデモを見ることができます。イメージに Scriptaculous が入っていれば、<http://localhost:8080/javascript/scriptaculous> でも、デモを試せます。

Scriptaculous パッケージでは Seaside でおなじみのアプローチが取られており、Smalltalk のオブジェクトを使ってアプリケーションをモデリングすれば、JavaScript のコードが自動生成されます。

簡単な例として、クライアント側で JavaScript を使うことで、RPN 計算機

をより自然な UI に変えていきましょう。今までは、キーを打つたびにページ全体がリフレッシュされていました。ページはそのまま、クライアントが操作した部分のみが更新されるようにしたいと思います。

🔗 JavaScript から表示部分を特定するには、*id* を振る必要があります。計算機のレンダリングメソッドを以下のように変えましょう。<sup>14</sup>

```
MyCalculator»renderContentOn: html
  html div id: 'keypad'; with: keypad.
  html div id: 'display'; with: display.
```

🔗 キーパッドを押したときに再描画されるようにするには、キーパッド側で表示コンポーネントを知る必要があります。インスタンス変数 *display* を *MyKeypad* に追加します。初期化用のメソッドとして *MyKeypad* » *setDisplay:* を用意し、*MyCalculator* » *initialize* で初期化しましょう。これでボタンに JavaScript のコードを割り当てるできるようになります。*MyKeypad* » *renderStackButton:callback:colSpan:on:* を以下のように変えましょう。

```
MyKeypad»renderStackButton: text callback: aBlock colSpan: anInteger on: html
  html tableData
    class: 'key';
    colSpan: anInteger;
    with: [
      html anchor
        callback: aBlock;
        onClick:      "JavaScriptのイベント 処理"
          (html updater
            id: 'display';
            callback: [ :r |
              aBlock value.
                r render: display ];
            return: false);
          with: [ html html: text ] ]
```

*onClick:* で JavaScript のイベントハンドラを指定しています。html updater によって *SUUpdater* のインスタンスが返ってきます。これは JavaScript の *Ajax.Updater* オブジェクト (<http://www.prototypejs.org/api/ajax/updater>) に該当する *Smalltalk* のオブジェクトです。このオブジェクトは AJAX リクエストの送信を行い、返ってきたテキストを使い特定部分の更新を行います。*id:* によって、*updater* に XHTML のどの DOM 要素を更新すべきかを指定しています。この例の場合は *'display'* の *id* が振られた *div* 要素になります。*callback:* で、ユーザがボタンを押したときのコールバックをブロックの形で指定します。ブロックの引数 *r* は、新たに生成されるレンダラで、これを使って *display* のコンポー

<sup>14</sup>チュートリアル例を自分で実装していない場合は、<http://www.squeaksource.com/PharoByExample> から、*PBE-SeasideRPN* パッケージをロードして、コードを修正していけばよいでしょう。クラスの接頭辞は本書と違い、*My\**ではなく *RPN\**になっています。

ネットの表示を行います。(コード上はメソッドの引数となっている `html` レンダラにもアクセスできますが、コールバックのブロックが評価される時には無効になっているので注意してください。) コールバック内では `display` のコンポーネントをレンダリングする前に、`aBlock` を評価しています。適切な処理を外から渡して行わせるためです。

`return: false` では、リンクに設定されていた元々のコールバックを呼ばないように JavaScript 側に指定しています。元のコールバックが呼ばれると、ページ全体がリフレッシュされてしまうからです。リンクの `callback` 部分を削除して、コールバックを設定しないようにもできますが、JavaScript が無効になっても動作するように、あえてこのようにしています。

🕒 計算機の数字キーを押してみても、まだページ全体のリフレッシュが行われてしまうことを確認してみてください。(Web ページの URL がボタンを押すたびに変わります)。

クライアント側の JavaScript の振る舞いを実装しましたが、まだ無効なままです。では、JavaScript のイベント処理を有効にしましょう。

🕒 計算機の下ツールバーに表示されている `Configure` リンクをクリックしてください。“`Add Library:`”の部分で `SULibrary` を選び、`Add` ボタンを押して `Close` で閉じれば完了です。

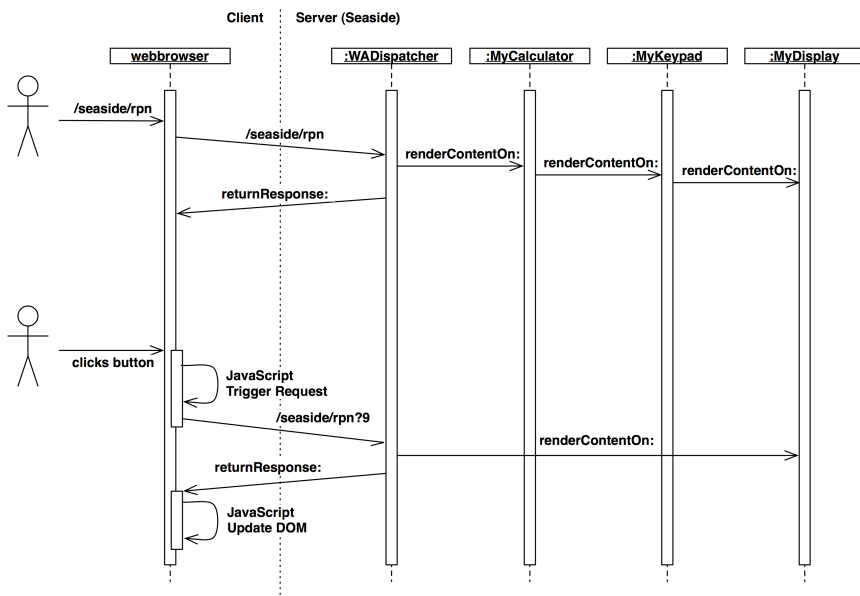
ライブラリを手で追加する代わりに、プログラムの行うこともできます。Seaside にアプリケーションを登録する際に以下のようにして指定します。

```
MyCalculator class»initialize
  (self registerAsApplication: 'rpn')
  addLibrary: SULibrary
```

🕒 もう一度 RPN 計算機を試してみましよう。前よりも自然な形で操作できます。キーボタンを押しても URL が変わらなくなりました。

動作は確認できましたが、仕組みがどうなっているのか気になるかもしれません。図 12.17 で、AJAX なしの場合と、有効にした場合とで、RPN 計算機のメッセージの流れを示しました。AJAX の場合は、基本的にレンダリング処理を飛ばして、`display` コンポーネントのみの更新を行います。JavaScript はリクエストの送信と、対応する DOM 要素の更新の両方を受け持っています。生成される JavaScript のソースを見てみましょう。

```
new Ajax.Updater(
  'display',
  'http://localhost/seaside/RPN+Calculator',
  {'evalScripts': true,
   'parameters': ['_s=zcdqfonqwbeYzkza', '_k=jMORHtqr', '9'].join('&')});
return false
```



### Seaside: AJAX Processing (simplified)

Lukas Renggli, 2007

Figure 12.17: Seaside での AJAX の処理 (簡略化したもの)

より高度なサンプルについては <http://localhost:8080/javascript/scriptaculous> を見るとよいでしょう。

**Tips** サーバー側のデバッグは通常の Smalltalk のデバッグを使うことができます。クライアント側のデバッグは FireFox (<http://www.mozilla.com>) を入れて、FireBug (<http://www.getfirebug.com/>) プラグインを有効にしておく便利です。

## 12.9 この章のまとめ

- Seaside を始めるには “Seaside One-Click Experience” イメージを <http://seaside.st> からダウンロードするのが最も簡単です。
- WAKom startOn: 8080 でサーバーが開始し、WAKom stop で終了します。

- 管理用の ID とパスワードは `WADispatcherEditor initialize` で初期化できます。
- `Toggle Halos` をクリックすると、コンポーネントのソースコードやオブジェクトの状態、CSS や XHTML を見ることができます。
- `WAGlobalConfiguration setDeploymentMode` で開発用のツールバーを隠せます。
- Seaside のアプリケーションはコンポーネントからできています。コンポーネントとは `WComponent` を継承したクラスのインスタンスのことです。
- Seaside の Web 管理ツールからアプリケーションとして登録できるコンポーネントを、ルートコンポーネントと呼びます。クラスメソッドの `canBeRoot` を実装して `true` を返すようにします。`initialize` のクラスメソッドで、`self registerAsApplication:` を自身に送ることで、登録することもできます。引数はアプリケーションのパスになります。`description` をオーバーライドすると Web 上のコンフィギュレーションエディタにアプリケーションの説明を表示させることも可能です。
- 状態をバックトラックするには、コンポーネントで `states` メソッドを定義して、復元したいオブジェクトの配列を返すように実装する必要があります。ユーザが「戻る」ボタンを押しても状態が正常に保たれます。
- コンポーネントは `renderContentOn:` を実装することで、自身を Web ページ内に表示させることができます。引数として XHTML レンダリング用のキャンバスがわたってきます。(通常は `html` という名前になっています)
- コンポーネントは、自身が含むサブコンポーネントを `self render: subcomponent` で表示させることができます。
- XHTML は ブラシにメッセージを送ることで生成されます。キャンバスに対して、`paragraph` や `div` などのメッセージを送ると、XHTML 要素に対応したブラシが返ってきます。
- ブラシに `with:` を含んだメッセージを連続して送るときは、必ず `with:` が最後になるようにします。`with:` によって XHTML 生成が終わり、その結果がレンダリングされるからです。
- 状態を操作するアクションはコールバックの中で書くようにします。レンダリングメソッド内でコンポーネントの状態を変えてはいけません。
- ブラシに `on:` インスタンス変数 `of:` オブジェクトを送ることで、フォームの要素やリンクに対し、インスタンス変数の値を簡単に結び付けることができます。
- スタイルシートの文字列を返す `style` メソッドを実装すると、コンポーネントが使う CSS をクラス階層中に定義できます。(通常デプロイ時には、外部の CSS を固定の URL で参照させるようにします。)

- 制御フローは、`x call: y`のように書くことで表せます。この場合、`x`のコンポーネントは`y`のコンポーネントへと置き換えられます。`y`は`answer:`で制御を`x`に戻すことができます。`call:`のレシーバはたいていは`self`ですが、それ以外のコンポーネントであってもかまいません。
- 制御フローはタスクとして表すこともできます。タスクは`WATask`のサブクラスとして定義します。`go`メソッドの中に、ワークフローを連続した`call:`のメッセージ送信として書いていきます。
- ユーザと簡単なやりとりを行うため、`WAComponents`の便利メソッドとして、`request:`、`inform:`、`confirm:`、`chooseFrom:caption:`が用意されています。
- ユーザが「戻る」ボタンを押して以前の状態にアプリケーションを戻してしまうのを防ぐには、`isolate:`を使います。ブロックでワークフローを分割することで、ブロックを越えて過去の状態に戻ることができなくなります。





Part III

# **Advanced Pharo**



# Chapter 13

## クラスとメタクラス

第5章で見てきたように、Smalltalk ではすべてがオブジェクトであり、いずれも何らかのクラスのインスタンスです。クラスも例外ではありません。つまり、クラスはオブジェクトであり、クラスオブジェクトは他のクラスのインスタンスとなっています。このオブジェクトモデルは、簡潔で、シンプルかつエレガントに統一されており、オブジェクト指向プログラミングの本質を捉えています。しかし、不慣れなうちはこの一貫性のために混乱してしまうかもしれません。この章では、それが複雑ではなく、“魔法”でも特別なものでもない、ただシンプルな規則が一貫して適用されているということを示します。これらの規則を追えば、システムがなぜこのように成り立っているのかがわかるでしょう。

### 13.1 クラスとメタクラスのルール

Smalltalk のオブジェクトモデルは、一貫した数少ないコンセプトの上に成り立っています。Smalltalk の設計者は、オッカムの剃刀と呼ばれる規則を使って、モデルが必要以上に複雑になる要因を削ぎ落としました。

第5章で見た、オブジェクトモデルのルールを思い出しましょう。

- Rule 1.** すべてのものはオブジェクトである。
- Rule 2.** すべてのオブジェクトは、あるクラスのインスタンスである。
- Rule 3.** すべてのクラスは、スーパークラスを持つ。
- Rule 4.** すべての出来事は、メッセージを送ることで生じる。
- Rule 5.** メソッド探索は継承チェーンに沿って行われる。

この章の冒頭で述べたように、Rule 1 の規則よりクラスもオブジェクトであるが成り立ちます。そして、Rule 2 より、クラスも何らかのクラスのインスタ

ンスであることがわかります。このクラスのクラスをメタクラスといいます。あるクラスのメタクラスは、そのクラスを定義した際に自動的に作られます。普段はメタクラスのことを気にする必要はないでしょう。しかし、ブラウザで“クラス側”を選んだときには、実は異なるクラスを見ているということを思い返してください。クラスとそのメタクラスは、前者が後者のインスタンスであるということを除けば、それぞれ別々のクラスであるということです。

クラスとメタクラスを正確に説明するには、第5章のルールに付け足す必要があります。

**Rule 6.** すべてのクラスはメタクラスのインスタンスである。

**Rule 7.** メタクラスの階層はクラスの階層と並列に存在する。

**Rule 8.** すべてのメタクラスは Class と Behavior を継承している。

**Rule 9.** すべてのメタクラスは Metaclass のインスタンスである。

**Rule 10.** Metaclassのメタクラスは、Metaclassのインスタンスである。

これら 10 個のルールが、Smalltalk のオブジェクトモデルのすべてです。

以下では、まず第5章で紹介した五つのルールを小さな例を使いながら説明します。そして、同じ例を用いて、新しいルールを深く見ていきます。

## 13.2 Smalltalk オブジェクトモデルの復習

すべてがオブジェクトなので、「青色オブジェクト」も Smalltalk ではオブジェクトです。

```
Color blue → Color blue
```

すべてのオブジェクトはあるクラスのインスタンスです。青色オブジェクトのクラスは Color です。

```
Color blue class → Color
```

興味深いことに、*alpha* (透明度) を指定すると、異なるクラス (TranslucentColor) のインスタンスが得られます。

```
(Color blue alpha: 0.4) class → TranslucentColor
```

Morph を作り、その色としてこの半透明オブジェクトを指定することができます。

```
EllipseMorph new color: (Color blue alpha: 0.4); openInWorld
```

これを実行すると、図 13.1 のようになります。

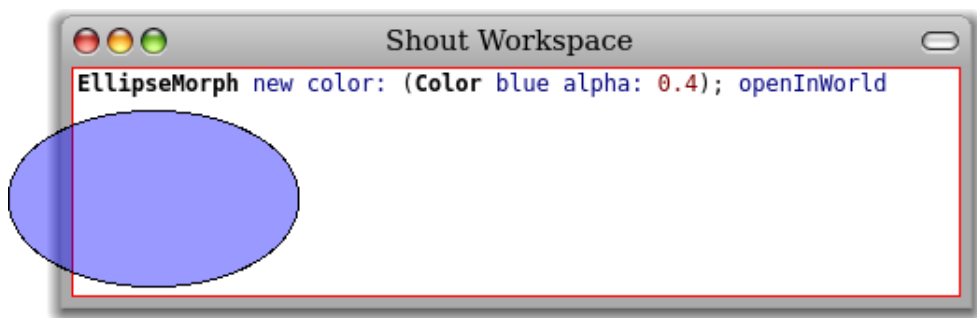


Figure 13.1: 半透明の楕円モーフ。

Rule 3 より、どのクラスもスーパークラスを持ちます。TranslucentColor のスーパークラスは Color です。そして、Color のスーパークラスは Object です。

```
TranslucentColor superclass  → Color
Color superclass             → Object
```

Rule 4 「すべての出来事は、メッセージを送ることで生じる」より、以下のことが導きだせます。blue は Color へのメッセージ、class と alpha: は青色オブジェクトへのメッセージ、openInWorld は楕円 Morph へのメッセージ、そして、superclass は TranslucentColor と Color へのメッセージです。すべてがオブジェクトなので、どのケースでもメッセージのレシーバはオブジェクトです。ただし、いくつかのケースではレシーバがクラスとなっています。

Rule 5 「メソッド探索は継承チェーンに沿って行われる」より、Color blue alpha: 0.4 で得られたオブジェクトに class メッセージを送ると、図 13.2 で示すように、メッセージは対応するメソッドが見つかる Object で取り扱われます。

この図は *is-a* の継承関係を示したものです。図のように (Color blue alpha: 0.4 で得られた) translucentBlue オブジェクトは、TranslucentColor のインスタンスであるわけですが、同時に Color オブジェクトであるとも、Object オブジェクトであるとも言えるのは、これらのクラスで定義されたすべてのメッセージに回答することができるからです。さらに補足すると、isKindOf: メッセージをオブジェクトに送ることで、レシーバが引数で与えられたクラスとである (*is a*) の関係にあるかどうかを調べることができます。

```
translucentBlue := Color blue alpha: 0.4.
translucentBlue isKindOf: TranslucentColor → true
translucentBlue isKindOf: Color           → true
translucentBlue isKindOf: Object          → true
```

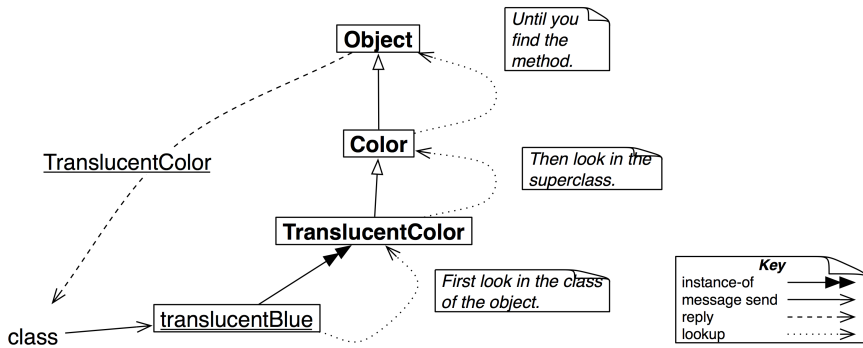


Figure 13.2: 半透明色オブジェクトへのメッセージ送信。

### 13.3 すべてのクラスはメタクラスのインスタスである

13.1 節で述べたように、そのインスタスがクラスであるようなクラスはメタクラスと呼ばれます。

**メタクラスは暗黙的な存在である** メタクラスはクラスを定義すると自動的に作成されます。これはプログラマが気にする必要はないので、暗黙的であると言えます。作成されるクラスごとに、そのクラスを唯一のインスタスとして持つ暗黙的なメタクラスが作成されます。

通常のクラスはグローバル変数によって参照できますが、メタクラスは匿名の存在です。しかし、インスタスであるクラスを通して、いつでもメタクラスを参照できます。例えば、Color のクラスは Color class のインスタスであり、Object のクラスは、Object class のインスタスです。

```
Color class   → Color class
Object class  → Object class
```

図 13.3 は、クラスがその (匿名の) メタクラスのインスタスであることを示しています。

クラスもまたオブジェクトであることから、メッセージを送って情報を得ることも簡単にできます。以下を見てみましょう。

```
Color subclasses      → {TranslucentColor}
TranslucentColor subclasses → #()
TranslucentColor allSuperclasses → an OrderedCollection(Color Object
ProtoObject)
```

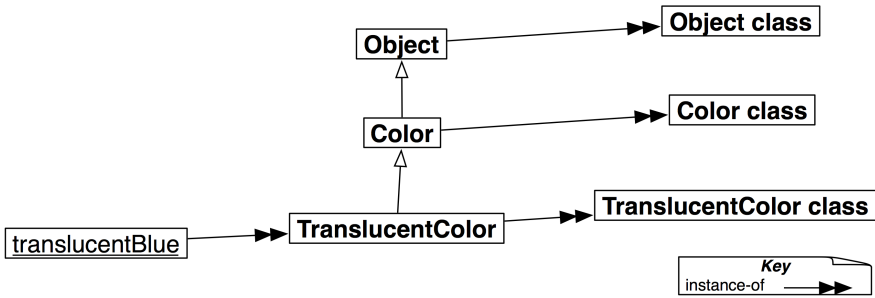


Figure 13.3: TranslucentColor のメタクラスと、そのスーパークラス。

```

TranslucentColor instVarNames    → #('alpha')
TranslucentColor allInstVarNames → #('rgb' 'cachedDepth' 'cachedBitPattern' 'alpha')
TranslucentColor selectors      → an IdentitySet(#pixelWord32
#asNontranslucentColor #privateAlpha #pixelValueForDepth: #isOpaque
#isTranslucentColor #storeOn: #pixelWordForDepth: #scaledPixelValue32 #alpha
#bitPatternForDepth: #hash #isTransparent #isTranslucent
#balancedPatternForDepth: #setRgb:alpha: #alpha: #storeArrayValuesOn:)
  
```

## 13.4 メタクラスの階層はクラスの階層と並列に存在する

Rule 7 は、メタクラスのスーパークラスに任意のクラスがなれるわけではないことを示しています。メタクラスのスーパークラスとして許されるのは、メタクラスの唯一のインスタンスであるクラスのスーパークラスのメタクラスです。

```

TranslucentColor class superclass → Color class
TranslucentColor superclass class → Color class
  
```

これが、「メタクラス階層はクラス階層と並列に存在する」ことの意味です。図 13.4 は、TranslucentColor の階層において、どのようになっているかを示しています。

```

TranslucentColor class          → TranslucentColor class
TranslucentColor class superclass → Color class
TranslucentColor class superclass superclass → Object class
  
```

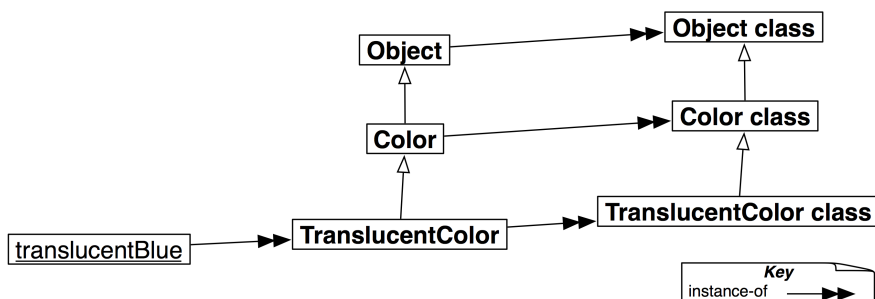


Figure 13.4: メタクラス階層と並列に存在するクラス階層。

クラスとオブジェクトの一貫性 ここで一息入れて考え直してみると、オブジェクトであろうとクラスであろうとメッセージを送るのに違いがないことは興味深いことです。どちらのケースも、メソッド探索はメッセージのレシーバクラスから始まり、継承チェーンをたどっていきます。

従って、クラスに送ったメッセージの探索は、メタクラスの階層チェーンをたどらなければなりません。例えば、blue メソッドは Color のクラス側に定義されています。TranslucentColor に blue メッセージを送信した場合は、他のメッセージとまったく同様にメソッドを探索します。すなわち、探索は TranslucentColor class でまず行われ、対応するメソッドが見つかるまでメタクラス階層を Color class までたどっていきます。(図 13.5 を参照)

```
TranslucentColor blue → Color blue
```

TranslucentColor のインスタンスではなく、普通の Color blue オブジェクトが得られることに注意してください— 不思議なところはどこにもありません！

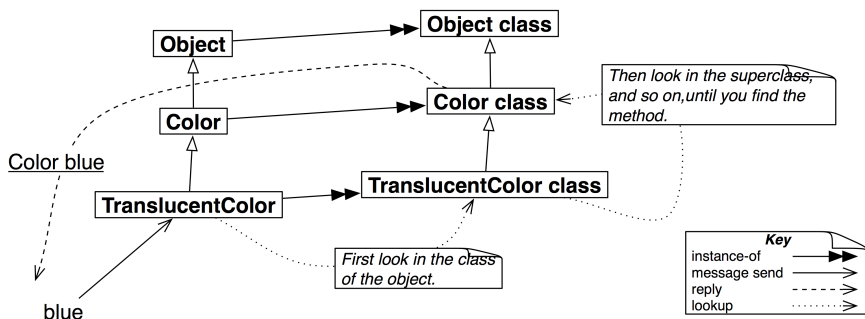



Figure 13.5: クラスに対するメッセージ探索は、通常のオブジェクトの場合と同じ。



すなわち、Smalltalk では統一された方法でメソッド探索が行われていると言えます。クラスもまた単なるオブジェクトであり、他のオブジェクトと同じように振る舞います。クラスは、新しいインスタンスを作る能力を持ちますが、これは、クラスがたまたま `new` というメッセージに反応することができて、`new` に対するメソッドが新しいインスタンスの作り方を知っているからにすぎません。通常はクラス以外のオブジェクトはこのメッセージに応答できませんが、そうする理由があるなら、`new` メソッドをメタクラス以外に追加することを止めるものは何もありません。

クラスもオブジェクトなので、インスペクトすることもできます。

 `Color blue` と `Color` をインスペクトしてみましょう。

前者は `Color` のインスタンスを、後者は `Color` 自身をインスペクトしていることに注意してください。どちらの場合もインスペクタのタイトルバーにはオブジェクトのクラスが表示されるので、少し混乱するかもしれません。

図 13.6 にあるように、`Color` のインスペクタによって、スーパークラス (`superclass`)、インスタンス変数 (`instanceVariables`)、メソッド辞書 (`methodDict`) などを見ることができます。

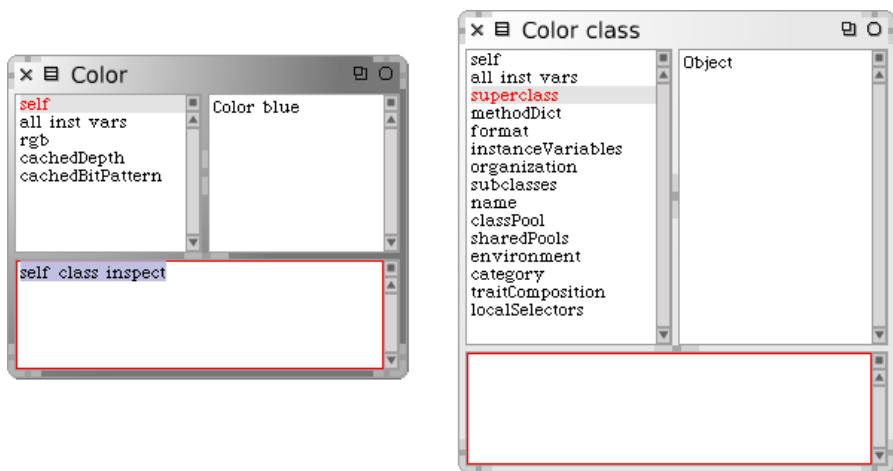


Figure 13.6: クラスもオブジェクトである。

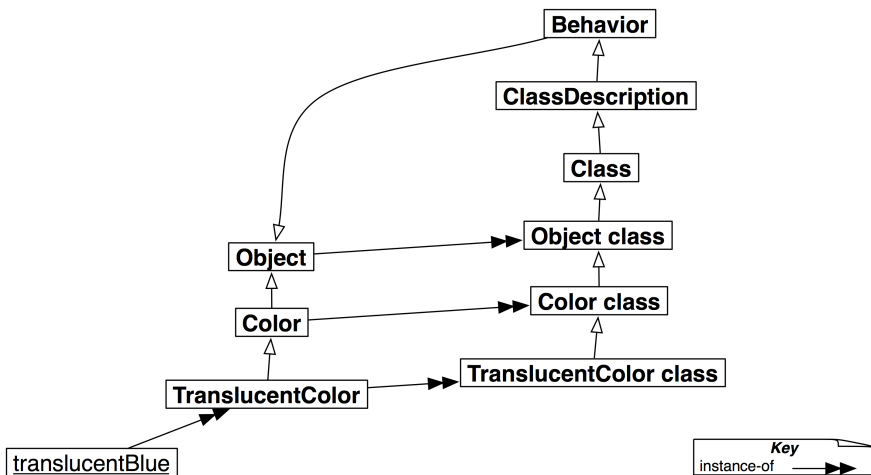


Figure 13.7: メタクラスは、Class と Behavior を継承している。

## 13.5 すべてのメタクラスは Class と Behavior を継承している

どのメタクラスもクラスである (*is-a*) ため、Class を継承しています。Class のスーパークラスは ClassDescription であり、さらにそのスーパークラスは Behavior です。Smalltalk ではすべてがオブジェクトである (*is-a*) ため、結局これらのクラスも Object を継承していることになります。図 13.7 に、完全な図を示します。

**new** はどこで定義されているのか？ **new** がどこで定義されており、どのように探索されるのかについて考えてみるのが、メタクラスが Class と Behavior を継承しているということの重要性を理解する手助けになります。**new** メッセージがクラスに送られるとき、図 13.8 のように、メタクラスの継承チェーンをたどって、スーパークラスである Class、ClassDescription、そして Behavior の順にメソッド探索が行われます。

「どこに **new** が定義されているのか」という問いは決定的に重要です。**new** は、まず Behavior で定義され、必要に応じてそのサブクラスで再定義されます。これには、プログラマが定義したクラスのメタクラスも含まれます。**new** メッセージをクラスに送った場合、まずは通常通りそのクラスのメタクラスから探索が開始され、そこで見つからなければ、**new** を再定義しているクラスまたは最終的には Behavior まで、順にスーパークラスをたどって探索されます。

TranslucentColor **new** を評価した結果は TranslucentColor のインスタンスであり、たとえメソッドが Behavior で見つかったとしても Behavior のインスタンス

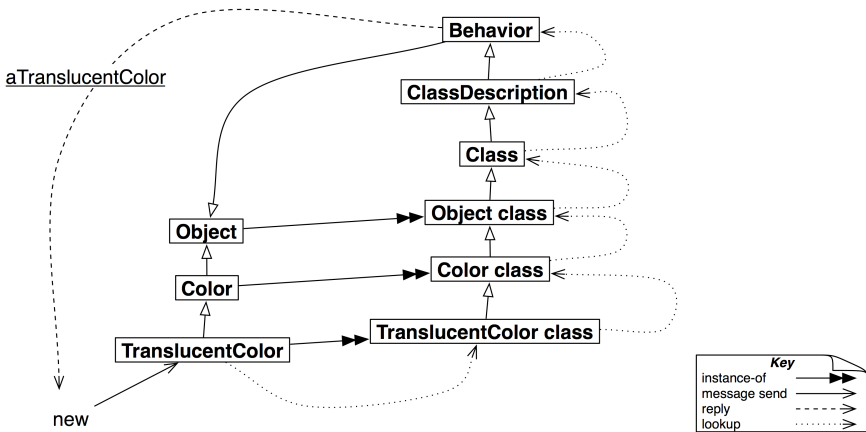


Figure 13.8: new は普通のメッセージで、メタクラスチェーンをたどって探索される。

になるわけではないことに注意してください。newは常にそのメッセージを受け取ったクラスである self のインスタンスを返します。これは newが他のクラスで実装されている場合でも同じです。

TranslucentColor new class → TranslucentColor "Behaviorのインスタンスではない!"

よくある間違いは、レシーバとなっているクラスのスーパークラスで new を探すことです。同じことは、指定された大きさのオブジェクトを生成するための標準的なメッセージである、new:についても言えます。例えば、Array new: 4 は、四つの要素を持つ配列を生成します。Arrayやそのスーパークラスで、このメソッドの定義を見つけることはできないでしょう。代わりに、実際のメソッド探索が行われる Array classとそのスーパークラスの階層に目を向ける必要があります。

**Behavior、ClassDescription および Class の責務について** Behavior は、インスタンスを持つオブジェクトのための必要最小限の状態を保持しています。すなわち、スーパークラスへのリンク、メソッド辞書、およびクラスのフォーマットです。クラスのフォーマットは、ポインタか否かの区別、コンパクトクラスか否かの区別、およびインスタンスの基本的な大きさをコード化したものです。Behaviorは Objectを継承しているので、それ自身とすべてのサブクラスは、普通のオブジェクトとして振る舞うことができます。

Behaviorは、コンパイラへの基本的なインターフェースも提供します。メソッド辞書の生成、メソッドのコンパイル、インスタンスの生成（すなわち new,

basicNew, new:, basicNew:)、クラス階層の操作(すなわち superclass:, addSubclass:)、メソッドへのアクセス(すなわち selectors, allSelectors, compiledMethodAt:)、インスタンス、およびインスタンス変数へのアクセス(すなわち allInstances, instVarNames ...)、クラス階層へのアクセス(すなわち superclass, subclasses)、および問い合わせ(すなわち hasMethods, includesSelector, canUnderstand:, inheritsFrom:, isVariable)のメソッドを提供します。

ClassDescription は抽象クラスであり、Class と Metaclass という直接のサブクラスが必要とする機能を提供しています。ClassDescription は Behavior が提供する基礎に、以下のような多くの機能を加えます。それらは、インスタンス変数の名前の管理、プロトコルによるメソッドの分類、チェンジセットの管理と変更の記録、そして変更のファイルアウトに必要なほとんどの機能などです。

Class は、すべてのクラスに共通する振る舞いを提供し、クラス名、メソッドのコンパイル、メソッドの保存、インスタンス変数を提供します。また、クラス変数名、共有プール変数(addClassVarName:, addSharedPool:, initialize)の実際のインターフェースも提供します。メタクラスは、唯一のインスタンス(すなわち、メタクラスでないクラス)に対するクラスであり、クラスとしてインスタンスにサービスを提供するため、すべてのメタクラスは最終的に Class を継承しています。

## 13.6 すべてのメタクラスは Metaclass のインスタンスである

メタクラスもまたオブジェクトです。図 13.9 が示すように、メタクラスは Metaclass のインスタンスです。Metaclass のインスタンスは匿名のメタクラスで、それぞれ唯一のインスタンスを持ちます。つまり、それがクラスというわけです。

Metaclass は、メタクラスに共通する振る舞いを提供し、メタクラスの唯一のインスタンスに対して初期化されたインスタンスを生成するインスタンス生成(subclassOf:)、クラス変数の初期化、メタクラスのインスタンス、メソッドのコンパイル、およびクラスの情報(継承関係、インスタンス変数、など)のメソッドを提供します。

## 13.7 Metaclass のメタクラスは Metaclass のインスタンスである

残る質問は一つだけです。Metaclass class のクラスは何でしょうか？

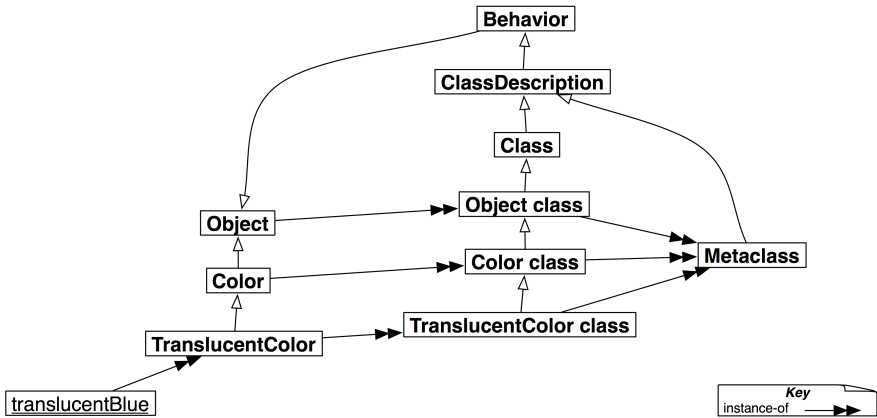


Figure 13.9: すべてのメタクラスは Metaclass のインスタンスである。

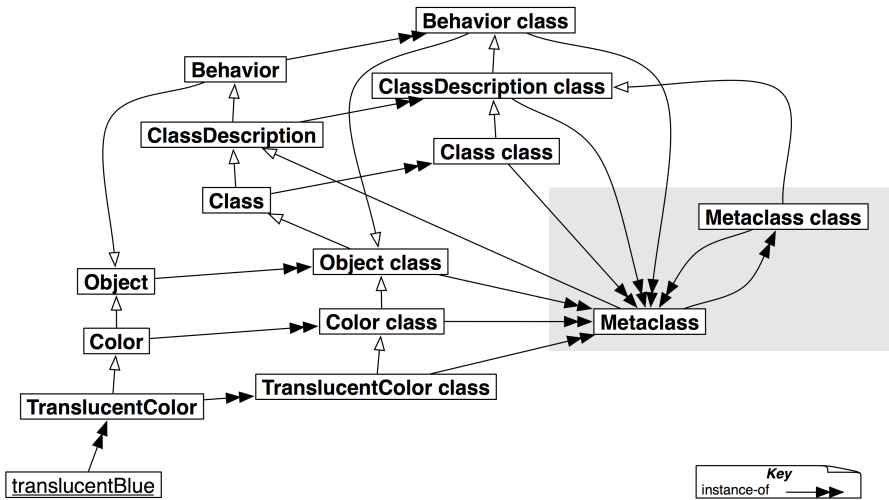


Figure 13.10: すべてのメタクラスは Metaclass のインスタンスである。Metaclass のメタクラスも同様。

答えは簡単でメタクラスです。もちろんシステム内の他のすべてのメタクラスと同様に、Metaclassのインスタンスであるべきです。(図 13.10 を参照)

この図は、すべてのメタクラスが Metaclass のインスタンスであり、Metaclass のメタクラスもまた同様であることを示しています。13.9 と 13.10 を比べると、Object classに至るまでメタクラスの階層がクラス階層の完全な鏡像になっていることがわかるでしょう。

以下の例は、クラス階層の問い合わせによって、図 13.10 が正確であることを示しています。(現実にはちょっとしたウソが混じっており、Object class superclassの結果は ProtoObject class であって Classではありません。Pharo では、Classに至るため、もう一つ上のスーパークラスに行く必要があります。)

Example 13.1: クラス階層

TranslucentColor superclass	→	Color
Color superclass	→	Object

Example 13.2: クラス階層に並列するメタクラス階層

TranslucentColor class superclass	→	Color class
Color class superclass	→	Object class
Object class superclass superclass	→	Class "NB: skip ProtoObject class"
Class superclass	→	ClassDescription
ClassDescription superclass	→	Behavior
Behavior superclass	→	Object

Example 13.3: Metaclass のインスタンス

TranslucentColor class class	→	Metaclass
Color class class	→	Metaclass
Object class class	→	Metaclass
Behavior class class	→	Metaclass

Example 13.4: Metaclass class は Metaclass のインスタンスである

Metaclass class class	→	Metaclass
Metaclass superclass	→	ClassDescription

## 13.8 この章のまとめ

クラスがどのように組織されており、一貫性のあるオブジェクトモデルがどのような効果を及ぼしているのかについて、よく理解できたのではないのでしょうか。忘れてたり混乱したりしたなら、いつでもメッセージ・パッシングが鍵であることを思い出してください。メッセージのレシーバのクラスからメソッドを探せばよいのです。このことはどんなレシーバについても同様です。レシーバのクラスでメソッドが見つからなければ、そのスーパークラスを探せばよいのです。

- すべてのクラスはメタクラスのインスタンスである。メタクラスは暗黙的である。メタクラスはクラスを定義するとき、そのクラスを唯一のインスタンスとするように自動的に作成されます。

- メタクラスの階層はクラスの階層と並列に存在する。クラスのメソッド探索は、普通のオブジェクトのメソッド探索と並行しており、メタクラスの継承チェーンに沿って行われます。
- すべてのメタクラスは **Class** と **Behavior** を継承している。すべてのクラスは **Class** のインスタンスです (*is-a*)。メタクラスはクラスでもあるため **Class** を継承しています。 **Behavior** はインスタンスを持つすべての存在に共通した振る舞いを提供します。
- すべてのメタクラスは **Metaclass** のインスタンスである。 **ClassDescription** は **Class** および **Metaclass** に共通することがらすべてを提供します。
- **Metaclass** のメタクラスは **Metaclass** のインスタンスである。クラス-インスタンスの関係 (*instance-of*) は閉じたループを形成します。すなわち、**Metaclass class class** → **Metaclass** ということです。





# Chapter 14

## リフレクション

Smalltalk はリフレクティブなプログラミング言語です。つまり、プログラムは自分自身の実行や構造を「(ランタイムシステムに) 反映する (リフレクトする)」ことができます。ランタイムシステムのメタオブジェクトを通常のオブジェクトとして具現化 (*reify*) することができるので、メタオブジェクトに問い合わせを行ったり、インスペクトして内容を見たりすることができます。Smalltalk のメタオブジェクトには、クラス、メタクラス、メソッド辞書、コンパイル済みメソッド、実行時スタックなどがあります。このようなリフレクションはイントロスペクションとも呼ばれ、多くのモダンなプログラミング言語でサポートされています。

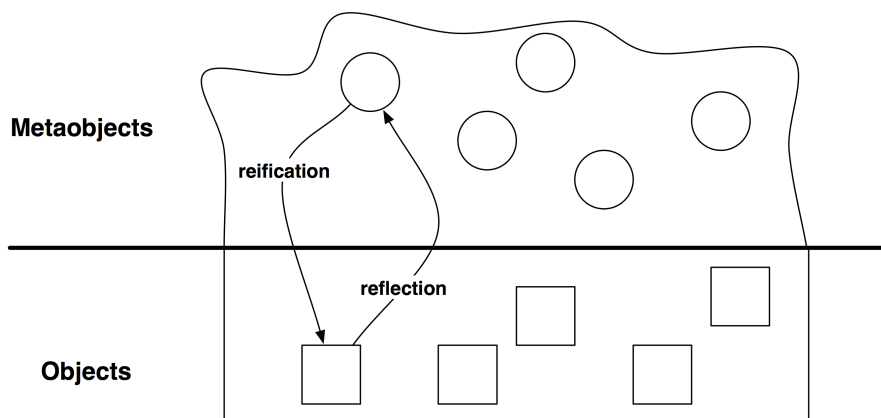


Figure 14.1: 具現化とリフレクション。

Smalltalk では逆の操作も可能で、具現化されたメタオブジェクトを変更して、ランタイムシステムに反映させることもできます (図 14.1 参照)。このようなリフレクションはインターセッションとも呼ばれ、主に動的なプログラミン


グ言語でサポートされていますが、静的な言語では極めて限られています。

他のプログラム（または自分自身）を操作するようなプログラムをメタプログラムと呼びます。リフレクティブなプログラミング言語では、イントロスペクションとインターセッションがどちらもサポートされているべきです。イントロスペクションとは、言語自身を定義するデータ構造（オブジェクト、クラス、メソッド、実行時スタックなど）を検査する機能です。インターセッションとは、そのようなデータ構造を変更できること、言い換えれば、言語の意味論やプログラムの振る舞いを、プログラム自身から変更できる機能です。ランタイムシステムの構造を検査・変更することを構造的リフレクションと呼び、それらの構造の解釈を変更することを振る舞いのリフレクションと呼びます。

この章では、主に `ind` 構造的リフレクションを扱います。実用的な例を多数挙げながら、`Smalltalk` がどのようにイントロスペクションとメタプログラミングをサポートしているのかを探っていきます。


## 14.1 イントロスペクション

インスペクタを使うと、オブジェクトの内容を見たり、インスタンス変数の値を変更したり、オブジェクトにメッセージを送ったりすることができます。

 次のコードをワークスペースで評価してみましょう：

```
w := Workspace new.
w openLabel: 'My Workspace'.
w inspect
```

このコードを評価すると、インスペクタとワークスペースがもう一つ開きます。このインスペクタは新しく開いたワークスペースの内部状態を示しており、左ペインにはインスタンス変数（`dependents`, `contents`, `bindings` など）、右ペインには選択したインスタンス変数の値が表示されます。`contents` インスタンス変数は、ワークスペースのテキストエリアに表示されている内容を表します。この時点で `contents` を選択すると、インスペクタの右ペインには空文字列が表示されます。

 この空文字列のところに `'hello'` と入力して、`accept` してみましょう。

これで `contents` 変数の値が変わりますが、まだワークスペースウィンドウは変化に気づかないので再描画されません。ウィンドウを更新させるには、インスペクタの下ペインで `self contentsChanged` を評価します。

### インスタンス変数へのアクセス

インスペクタはどのように動いているのでしょうか？ `Smalltalk` では、インスタンス変数はすべて保護されています。理論上は、アクセサメソッドがそのク

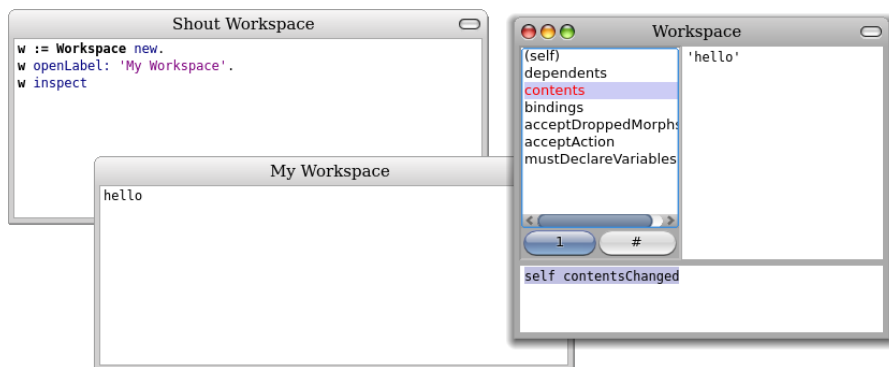


Figure 14.2: !ワークスペース!をインスペクトしている様子。

ラスに定義されていない限り、他のオブジェクトからインスタンス変数にアクセスすることはできません。しかし、実際にはインスペクタはアクセサがなくてもインスタンス変数にアクセスできます。これは Smalltalk がリフレクションの機能を持っているからです。Smalltalk では、クラスはインスタンス変数を名前または数値の添字で定義します。インスペクタは、Objectクラスで定義されているメソッド `instVarAt: index` または `instVarNamed: aString` を使ってインスタンス変数にアクセスします。数値の添字 `index` で位置を指定するか、インスタンス変数名 `aString` を指定します。インスタンス変数に新しい値を代入するには、`instVarAt:put:` か `instVarNamed:put:` を使います。

例えば次のコードを評価すると、最初のワークスペースで変数 `w` に束縛されていたオブジェクトの内部状態を変更できます。

```
w instVarNamed: 'contents' put: 'howdy'; contentsChanged
```

注意: これらのメソッドは開発ツールを作る分には便利ですが、通常のアプリケーション記述には使わない方がよいでしょう。リフレクティブなメソッドはオブジェクトのカプセル化境界をまたいでしまうので、理解も保守も難しいコードの原因となってしまいます。

`instVarAt:` と `instVarAt:put:` はどちらもプリミティブメソッドです。プリミティブメソッドは、バーチャルマシンの原始的操作として実装されています。これらのメソッドのコードを調べれば、特殊なプラグマ構文 `<primitive: N>` (`N`は整数) が使われていることがわかるでしょう。

Object>instVarAt: index

"プリミティブ。オブジェクト内の固定長配列で管理されているインスタンス変数を返す。"

<primitive: 73>

"固定長配列の範囲を超えてアクセスする。"

↑self basicAt: index - self class instSize

通常はプリミティブ呼び出しの行以降のコードは実行されません。その部分はプリミティブが失敗した場合にのみ実行されます。この例の場合では、存在しないインスタンス変数にアクセスしようとする、プリミティブに続くコードがバックアップとして試みられます。この仕組みにより、プリミティブメソッドが失敗したときにデバッガを起動することもできます。プリミティブメソッドのコードは変更できますが、Pharo システムの安定性を脅かしかねないため注意してください。

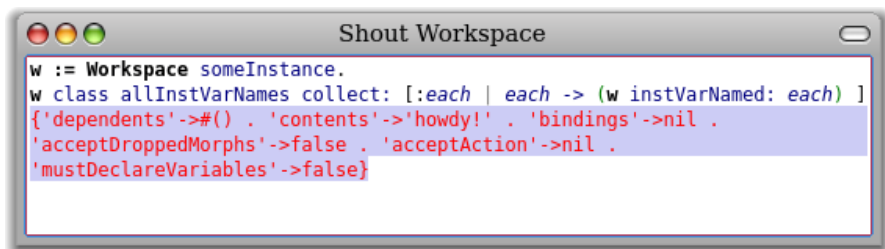


Figure 14.3: ある Workspace オブジェクトのすべてのインスタンス変数を表示している様子。

図 14.3 は、Workspace クラスのあるインスタンス (w) のすべてのインスタンス変数を表示する方法を示しています。allInstVarNames メソッドは、与えられたクラスにあるすべてのインスタンス変数の名前を返します。

同様の方法で、同じ特徴を持つインスタンスを集めることもできます。例えば SketchMorph クラスのインスタンスのうち、owner インスタンス変数の内容がワールドモーフであるもの (すなわち現在表示されているすべてのイメージ) を集めてみましょう。次の式を試してみてください。

```
SketchMorph allInstances select: [:c | (c instVarNamed: 'owner') isWorldMorph]
```

## インスタンス変数に対する繰り返し処理

instanceVariableValues メッセージについて調べてみましょう。このメソッドは、継承したものを除くすべてのインスタンス変数の値をコレクションとして返します。次に例を示します:

```
(1@2) instanceVariableValues → an OrderedCollection(1 2)
```

このメソッドは Object クラスで次のように実装されています:

```
Object>instanceVariableValues
"レシーバのクラスで追加されたインスタンス変数の値を集めたコレクションを返す。"
| c |
c := OrderedCollection new.
self class superclass instSize + 1
to: self class instSize
do: [: i | c add: (self instVarAt: i)].
↑ c
```

このメソッドは、スーパークラスで使われている最後のインスタンス変数の添字の直後から開始して、レシーバのクラスが定義しているインスタンス変数の添字に対して繰り返し処理をしています。(instSize メソッドは、クラスが定義している名前付きインスタンス変数の数を返します)。

## クラスやインターフェースへの問い合わせ

Pharo の開発ツール (コードブラウザ、デバッガ、インスペクタなど) はすべて、これまでに見て来たリフレクティブな機能を使っています。

もういくつか、開発ツールを作る際に役立つメッセージを挙げましょう:

isKindOf: aClass は、レシーバが aClass そのいずれかのスーパークラスのインスタンスであれば真を返します。例えば:

```
1.5 class           → Float
1.5 isKindOf: Number → true
1.5 isKindOf: Integer → false
```

respondsTo: aSymbol は、セレクトアが aSymbol であるメソッドをレシーバが持っていれば真を返します。例えば:

```
1.5 respondsTo: #floor → true "Numberクラスはfloorを実装しています"
1.5 floor           → 1
Exception respondsTo: #, → true "例外クラスはグループ化できません"
```

警告:ここで紹介した機能は、開発ツールの記述には大変便利ですが、通常のアプリケーション作成には適していません。オブジェクトが属するクラスや理解できるメッセージを調べようとするのは、カプセル化の原則に反しており、設計に問題がある兆候です。しかしながら、開発ツールは普通のアプリケーションではありません。開発ツールの対象はソフトウェアそのものなので、コードの内部を詳細に調べる権利があると言えます。

## コード・メトリクス

Smalltalk のイントロスペクションを使って、いろいろなコード・メトリクスが簡単に調べられるという例を見てみましょう。コード・メトリクスとは、継承の階層の深さ、直接的・間接的なサブクラスの数、各クラスのメソッドやインスタンス変数の数、クラスで新しく定義されたメソッドやインスタンス変数の数などを計測することです。以下に Morphクラスのメトリクスをいくつか示します。Morphクラスは Pharo のすべてのグラフィカル・オブジェクトのスーパークラスなので、それ自身が巨大で、かつ巨大な継承階層の根となっていることがわかります。もしかするとリファクタリングが必要かも！

Morph allSuperclasses size.	→	2 "継承階層の深さ"
Morph allSelectors size.	→	1378 "メソッドの数"
Morph allInstVarNames size.	→	6 "インスタンス変数の数"
Morph selectors size.	→	998 "Morphで新しく定義されたメソッドの数"
Morph instVarNames size.	→	6 "Morphで新しく定義されたインスタンス変数の数"
Morph subclasses size.	→	45 "直接のサブクラスの数"
Morph allSubclasses size.	→	326 "サブクラスの総数"
Morph linesOfCode.	→	5968 "総コード行数"

オブジェクト指向言語において最も興味深いメトリクスの一つは、スーパークラスから継承したメソッドに拡張を加えているメソッドの数です。このメトリクスから、クラスとそのスーパークラスの関係に関する情報が得られます。次の節以降では、このような疑問に対する答えを見つけるために、ランタイムの情報に関する知識をどのように使えばよいのかを見ていきます。

## 14.2 コードをブラウズする

Smalltalk では、すべてがオブジェクトです。特にクラスもまたオブジェクトであり、そのインスタンスを探索するための便利な機能を提供しています。これから挙げるメッセージはほとんど Behaviorで実装されているので、どのクラスも理解できます。

前に挙げたように、クラスに#someInstanceメッセージを送ると、そのクラスのあるインスタンスが得られます。

```
Point someInstance → 0@0
```

すべてのインスタンスを集めるには#allInstancesを、メモリ中にあるインスタンスの数を得るには#instanceCountを使うことができます。

```
ByteString allInstances → #('collection' 'position' ...)
ByteString instanceCount → 104565
String allSubInstances size → 101675
```

以下の機能は、クラスから特定の条件を満たすメソッドを抽出できるので、アプリケーションをデバッグするときに役立ちます。

- `whichSelectorsAccess:` は、指定したインスタンス変数を読み書きするすべてのメソッドのセレクタを返します。
- `whichSelectorsStoreInto:` は、指定したインスタンス変数の値を変更するすべてのメソッドのセレクタを返します。
- `whichSelectorsReferTo:` は、指定したメッセージを送るすべてのメソッドのセレクタを返します。
- `crossReference` 各メッセージを、そのメッセージを送るメソッドの集合に関連づける

```
Point whichSelectorsAccess: 'x'    →  #(#\'\' #= #scaleBy: ...)
Point whichSelectorsStoreInto: 'x' →  #(#setX:setY: ...)
Point whichSelectorsReferTo: #+    →  an IdentitySet(#rotateBy:about: ...)
Point crossReference              →  an Array(
  an Array('*' an IdentitySet(#rotateBy:about: ...))
  an Array('+ ' an IdentitySet(#rotateBy:about: ...))
  ...)
```

次のメッセージは継承階層も考慮した結果を返します:

- `whichClassIncludesSelector:` 与えられたメッセージを実装しているクラスを返します。
- `unreferencedInstanceVariables` レシーバのクラスでもサブクラスでも使われていないインスタンス変数をリストにして返します。

```
Rectangle whichClassIncludesSelector: #inspect → Object
Rectangle unreferencedInstanceVariables      → #()
```

`SystemNavigation` は、システム内のソースコードのブラウズや問い合わせを行うのに便利なメソッドを持つファサードです。`SystemNavigation default` を評価して得られるインスタンスを使ってシステム情報を調べられます。例えば:

```
SystemNavigation default allClassesImplementing: #yourself → {Object}
```

次のメッセージの説明は不要でしょう:

```
SystemNavigation default allSentMessages size    → 24930
SystemNavigation default allUnsentMessages size → 6431
SystemNavigation default allUnimplementedCalls size → 270
```

中には実装されているのに使われていないメッセージもありますが、`perform:`を使うなどして、ソースコードに現れない形で送られることもあるので、必ずしも不要とは限りません。ただし、送られているのに実装されていないメッセージは、送信したときに失敗することになるので、問題がある可能性が高いと言えます。このようなメッセージは、不完全な実装、廃止された API、ライブラリ不足を示している可能性があります。

`SystemNavigation default allCallsOn: #Point` は、ソースコード中で `Point` を明示的にレシーバとしているすべてのメッセージを返します。

以上の機能はすべて、`Pharo` のプログラミング環境 (特にコードブラウザ) に統合されています。既にご存知だとは思いますが、メッセージをブラウズする際に使える便利なキーボードショートカットがあります。それらは、メッセージの実装を取得するための `CMD-m`、メッセージの送信者を取得するための `CMD-n` などです。一方で、このような問い合わせ機能が多数 `SystemNavigation` クラスの *browsing* プロトコルに属するメソッドとして実装されていることは、それほど知られていないかもしれません。例えば、次の式を評価すると `ifTrue:` メッセージのすべての実装をプログラムのブラウズできます:

```
SystemNavigation default browseAllImplementorsOf: #ifTrue:
```

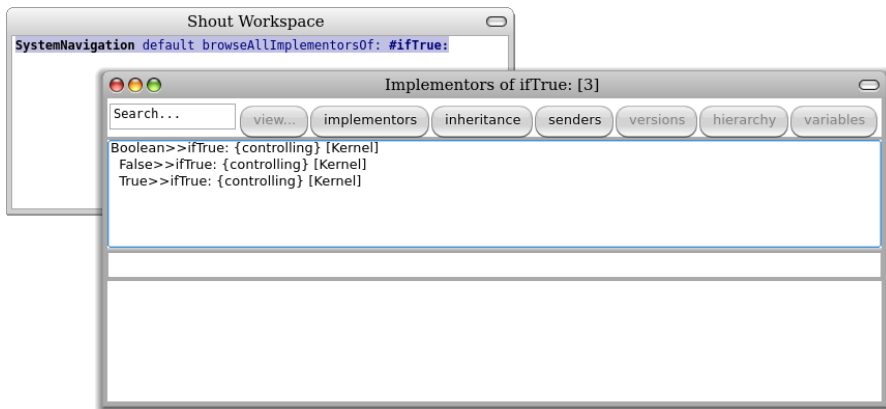


Figure 14.4: すべての `#ifTrue:` の実装をブラウズしている様子。

特に便利なメソッドは `browseAllSelect:` と `browseMethodsWithSourceString:` の二つです。以下にシステム中のすべてのメソッドから `super` 送信を行うものをブラウズする方法を示します。最初の方法は総当たりのですが、二つ目の方法ではより正確で、検索結果から不要なものを多少減らすことができます。

```
SystemNavigation default browseMethodsWithSourceString: 'super'.
SystemNavigation default browseAllSelect: [:method | method sendsToSuper ].
```



## 14.3 クラス、メソッド辞書、メソッド

クラスもオブジェクトなので、他のオブジェクトと同様にインスペクト・エクスプロアできます。


 Point exploreを評価してみましょう。

図 14.5 では、エクスプローラが Point クラスの構造を表示しています。このクラスがメソッドを辞書として保持しており、この辞書のキーがメソッドのセレクタとなっていることがわかります。セレクタ `#*` をキーとしている要素として、`Point*` が逆コンパイルされたバイトコードとして表示されています。

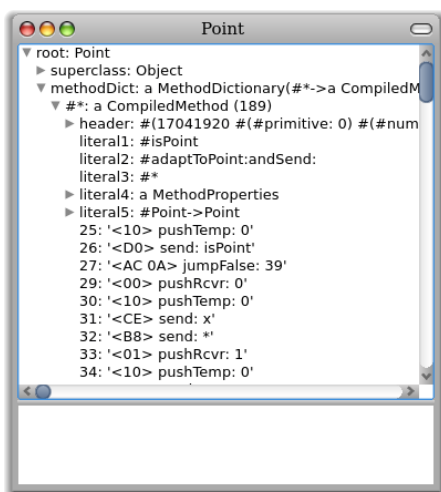


Figure 14.5: Pointクラスと`#*`メソッドのバイトコードをエクスプロアする。

クラスとメソッドの関係について考えてみましょう。図 14.6 を見ると、クラスとメタクラスは共通するスーパークラスとして `Behavior` を持っていることがわかります。`Behavior` が、`new` などのようにクラスに関するメソッドとして重要なものが定義されている場所となっています。どのクラスもメソッドセレクタとコンパイル済みメソッドを関連づけるメソッド辞書を持っています。コンパイル済みメソッドは自分が所属するクラスを知っています。図 14.5 から、`literal5literal5` にキーをシンボル `Point`、値を `Point` クラスとするアソシエーションとして所属クラスの情報が含まれていることがわかります。

システムに関する情報を問い合わせる際にクラスとメソッドの関係を利用することができます。例えば、あるクラスのメソッドのうち新しく定義されたもの、つまりスーパークラスのメソッドをオーバーライドしていないものを見つけるには、次のようにクラスからメソッド辞書をたどります：

```
[aClass| aClass methodDict keys select: [:aMethod |
(aClass superclass canUnderstand: aMethod) not ]] value: SmallInteger
```

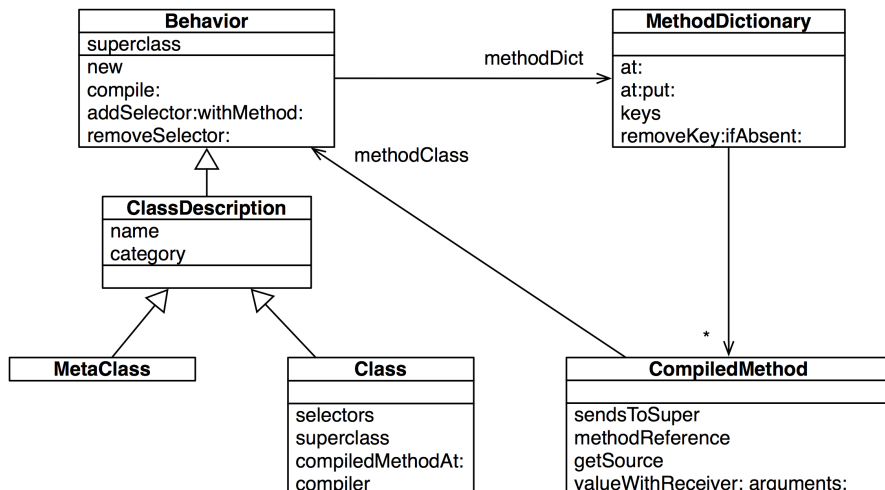


Figure 14.6: クラス、メソッド辞書、コンパイル済みメソッド

```
→ an IdentitySet(#threeDigitName #printStringBase:nDigits: ...)
```

コンパイル済みメソッドは、メソッドのバイトコードを保持しているだけでなく、システムに問い合わせを行う便利なメソッドも数多く提供しています。isAbstractメソッド (subclassResponsibilityを送るかどうか) はそのようなメソッドの一つです。isAbstractメソッドを使って、抽象クラスのすべての抽象メソッドを同定することができます。

```
[aClass| aClass methodDict keys select: [:aMethod |
(aClass>>aMethod) isAbstract ]] value: Number
→ #(#storeOn:base: #printOn:base: #+ #- #* #/ ...)
```

このコードでは、セレクタに対応するコンパイル済みメソッドを取得するための>>メッセージが使われています。

ある継承階層内で使われている super 送信、例えば Collection クラスのサブクラスで super 送信を行っているメソッドの一覧をブラウズするには、以下のようにいくらか高度な問い合わせを行います。

```
class := Collection.
SystemNavigation default
browseMessageList: (class withAllSubclasses gather: [:each |
each methodDict associations
select: [:assoc | assoc value sendsToSuper]
thenCollect: [:assoc | MethodReference class: each selector: assoc key]])
name: 'Supersends of ', class name , ' and its subclasses'
```

目的のメソッドを同定するため、クラスからメソッド辞書を經由してコンパイル済みメソッドへと探索している過程に注目してください。MethodReferenceはコンパイル済みメソッドのための軽量プロキシで、多くのツールで使われています。CompiledMethod>methodReference はコンパイル済みメソッドへのメソッド参照 (MethodReferenceのインスタンス) を返す便利なメソッドです。

```
(Object>>#)= methodReference methodSymbol  →  #=
```

## 14.4 ブラウザ環境

SystemNavigation はシステムコードに対してプログラムの問い合わせを行ったりブラウザしたりする便利な機能を提供しますが、もっと良い方法もあります。Pharo に統合されているリファクタリングブラウザを使うと、複雑な問い合わせを対話的にもプログラムのにも行うことができます。

例えば、Collection クラス階層のメソッドで、メソッド自身と異なるメッセージを super に送っているメソッドに興味があるとしましょう。このようなコードは、普通であれば super 送信を self 送信に置き換えるべきなので、悪いコードの臭いがすると考えられます。(考えてみてください。super を使う必要があるのは、メソッドをオーバーライドして拡張するときだけです。継承されたメソッドにアクセスするのであれば、self にメッセージを送れば済みます。)

リファクタリングブラウザは、問い合わせを関心のあるクラスやメソッドに限定するエレガントな方法を提供しています。


 ブラウザを開いて Collection クラスを見てみましょう。クラス名をアクションクリックして refactoring scope>subclasses with を選択すると、Collection クラス階層に限定されたブラウザ環境が開きます。この限定されたスコープの中で refactoring scope>super-sends を選択すると、Collection クラス階層にある super 送信を行うすべてのメソッドを列挙した新しいウィンドウが開きます。この中からあるメソッドをクリックして、refactor>code critics を選んでください。Lint checks>Possible bugs>Sends different super message と選んでゆき、browse をアクションクリックしてみましょう。

図 14.7 にあるように、Collection クラス階層にはそのようなメソッドが 19 個あって、その中には super printOn: を送っている Collection>printNameOn: も含まれていることがわかります。

このブラウザ環境はプログラムからでも用意できます。例えば、次のコードでは Collection とそのサブクラスを扱う BrowserEnvironment を生成し、そこから super 送信を行うメソッドのみを抽出して、その結果を表示するウィンドウを開きます。

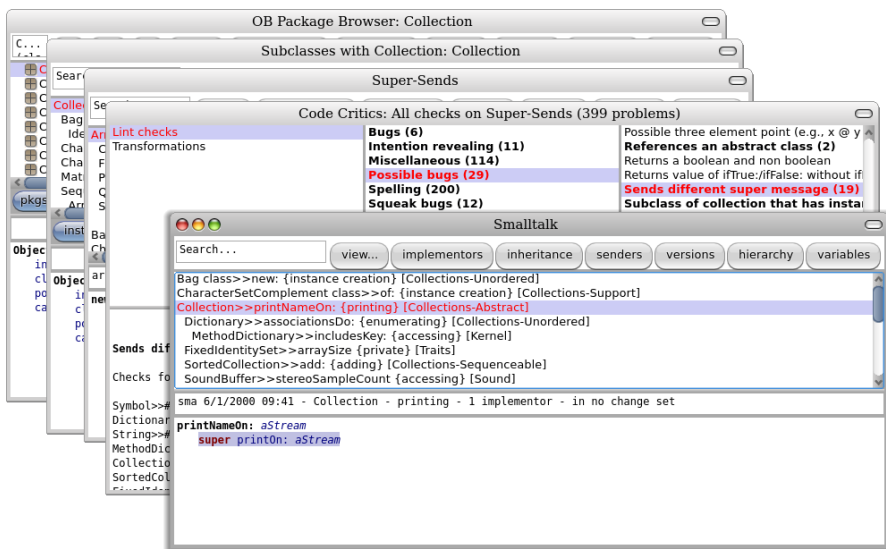


Figure 14.7: 自身のセレクタとは異なる super 送信を行うメソッドを探す様子。

```
((BrowserEnvironment new forClasses: (Collection withAllSubclasses))
 selectMethods: [:method | method sendsToSuper]
 label: 'Collection methods sending super';
 open.
```

このコードは先に挙げた `SystemNavigation` を使った例と同等ですが、よりコンパクトになっていることに注目してください。

最終的には次のようにして、自身のメソッドセレクタと異なる super 送信を行うメソッドをプログラマ的に探せます:

```
((BrowserEnvironment new forClasses: (Collection withAllSubclasses))
 selectMethods: [:method |
 method sendsToSuper
 and: [(method parseTree superMessages includes: method selector) not]])
 label: 'Collection methods sending different super';
 open
```

ここではメソッドセレクタと異なる super メッセージを探すために、コンパイル済みメソッドのそれぞれから、(リファクタリングブラウザが使用する) 構文木を取得しています。RBProgramNodeクラスの `querying` プロトコルを見て、構文木に何を尋ねることができるのかを確認してください。

## 14.5 実行時コンテキストにアクセスする


これまで Smalltalk のリフレクティブな機能を使ったオブジェクト、クラス、メソッドへの問い合わせの仕方やエクスペロアの方法を見てきました。では、実行時環境についてはどうでしょう？

### メソッドコンテキスト


実際のところ、実行中のメソッドに対応する実行時コンテキストはバーチャルマシンが管理しています — 仮想イメージの中にはないのです！しかし、デバッガは明らかに実行時コンテキストにアクセスして動作しています。さらには、実行時コンテキストを探ることも、他のオブジェクトと同様に何の問題もありません。どうなっているのでしょうか？

実は、デバッガには何の不思議なこともありません。この秘密は、これまで軽く触れただけだった `thisContext` という擬似変数にあります。メソッド中で `thisContext` が参照されたときにはいつでも、そのメソッドのコンテキストが具現化され、リンクした一連の `MethodContext` オブジェクトとしてイメージから操作できるようになります。

この仕組みは簡単に実験できます。

 Integer»factorialの実装に下線部の式を挿入してしてください:

```
Integer»factorial
"レシーバの階乗を返す。"
self = 0 ifTrue: [thisContext explore. self halt. ↑ 1].
self > 0 ifTrue: [↑ self * (self - 1) factorial].
self error: 'Not valid for negative integers'
```

 3 factorialをワークスペースで評価します。図 14.8 のようにデバッガウィンドウとエクスペローラの両方が表示されるはずですが。

貧相なデバッガへようこそ！ここでエクスペロアされたオブジェクトのクラスをブラウズすると（すなわちエクスペローラの下のパネルで `self browse` を評価すると）、`thisContext` が、`MethodContext` のインスタンスであることがわかるでしょう。`sender` の連鎖からたどれるすべてのオブジェクトも同様です。

`thisContext` は日常的なプログラミングで使うものではありませんが、デバッガのようなツールを実装したり、コールスタックの情報にアクセスしたりする際には必要不可欠です。次の式を評価すると、`thisContext` を使っているメソッドを見つけられます。

```
SystemNavigation default browseMethodsWithSourceString: 'thisContext'
```

実は、メッセージのセンダを取得することが `thisContext` の最も一般的な用途となっています。次に示すコードは典型的な例です：

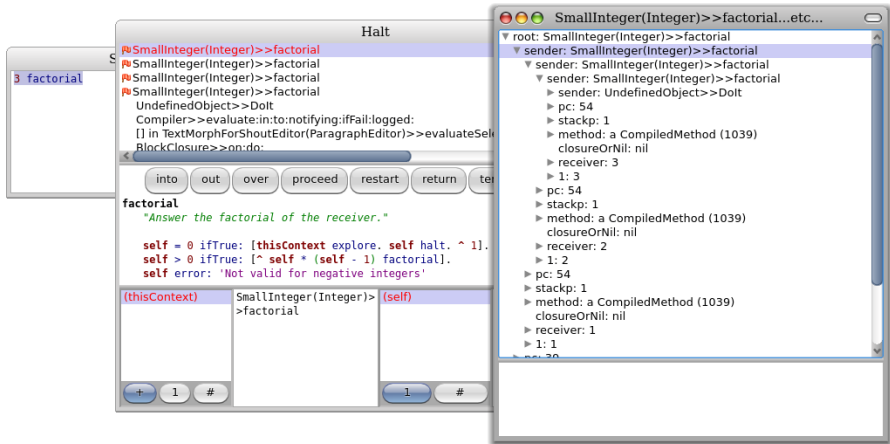


Figure 14.8: thisContext をエクスポアしている様子。

#### Object»subclassResponsibility

”このメッセージにより、サブクラスの挙動に関するフレームワークを設定します。  
サブクラスがこのメッセージを実装すべきであると宣言します。”

```
self error: 'My subclass should have overridden ', thisContext sender selector
printString
```

Smalltalk の慣習では、self subclassResponsibility を送るメソッドは抽象メソッドとみなされます。しかし、Object»subclassResponsibility はどの抽象メソッドが呼ばれたのかという情報を含むエラーメッセージをどうやって提供するのでしょうか？ 答えは簡単。thisContextにセンダを尋ねるだけです。

### 賢いブレークポイント

Smalltalk でブレークポイントを設定するには、メソッド内の関心のある箇所 self halt を評価するコードを書きます。この式が実行されると thisContext が具現化され、そのブレークポイントにおけるデバッガのウィンドウが開きます。しかしながら、システムで集中的に使われるメソッドにブレークポイントを設定すると問題が起きてしまいます。

例えば、OrderedCollection»add:の実行を解析したいとしましょう。このメソッド内にブレークポイントを設定すると困ったこととなります。

🕒 新しい仮想イメージを用意して、次のブレークポイントを設定します:

```
OrderedCollection»add: newObject
self halt.
↑self addLast: newObject
```

仮想イメージがデバッグウィンドウすら開かずに直ちにフリーズしてしまいます！ 次の点を理解すれば、問題がはっきりとわかります。(i) `OrderedCollection` »`add:`はシステムのあちこちで使われているので、変更をアクセプトしてごく間もない時点でブレークポイントが起動します。(ii) しかし、起動したデバッグ自身が `add:`メッセージを `OrderedCollection`のインスタンスに送るので、そのデバッグが邪魔されてしまいます！ このような問題を避けるには、デバッグしたい状況でのみ条件的に停止する方法が必要です。その用途にぴったりなのが `Object`»`haltIf:` です。

では `add:`を、例えば `OrderedCollectionTest`»`testAdd`のコンテキストから呼ばれたときにだけ停止させたいとしましょう。

🌀 また新しい仮想イメージを用意して、次のブレークポイントを設定してみましょう：

```
OrderedCollection»add: newObject
self haltIf: #testAdd.
↑self addLast: newObject
```

`OrderedCollectionTest`を実行してみましよう。今度はフリーズしません。( `OrderedCollectionTest`は `CollectionsTests-Sequenceable` カテゴリにあります)

どう動いているのでしょうか？ `Object`»`haltIf:`を見てみましょう：

```
Object»haltIf: condition
| cntxt |
condition isSymbol ifTrue: [
    "指定されたセレクタ・シンボルのメソッドがセンダの呼び出しチェーン
    に含まれている場合のみ停止する"
    cntxt := thisContext.
    [cntxt sender isNil] whileFalse: [
        cntxt := cntxt sender.
        (cntxt selector = condition) ifTrue: [Halt signal].
    ]
    ↑self.
].
...
```

`haltIf:`は実行時スタックを `thisContext`から遡りながら、呼び出し側のメソッドの名前が引数と一致するか確認します。一致したら例外を発生させます。この例外はデフォルトの挙動として、デバッグを起動します。

`haltIf:`の引数に真偽値や真偽値を返すブロックを渡してもよいですが、それらの場合は単純な処理になるので `thisContext`は使わずに済みます。

## 14.6 理解されないメッセージをインターセプトする

ここまで、Smalltalk のリフレクティブな機能をオブジェクト、クラス、メソッド、実行時スタックに問い合わせを行ったりエクスポアしたりするのに主に使ってきました。ここからは Smalltalk システムの構造に関する知識を、メッセージをインターセプトしたり振る舞いを実行時に変えたりするためにどう使っていけるのかを見ていきます。

オブジェクトはメッセージを受け取ると、まず自身のクラスのメソッド辞書からメッセージに対応するメソッドを探します。対応するメソッドが見つからなければ、Object にたどり着くまでクラス階層を遡りつつ探します。それでも見つからなければ、オブジェクトは `doesNotUnderstand:` メッセージを、元々のメッセージセクタとメッセージへの引数を引数として自分自身に送ります。メソッド探索の処理が `doesNotUnderstand:` の送信に対しても Object  $\gg$  `doesNotUnderstand:` が見つかるまで繰り返され、その結果としてデバグが起動します。

しかし、もし `doesNotUnderstand:` がメソッド検索パスにある Object のサブクラスでオーバーライドされていたらどうなるでしょう？ これはある種の非常に動的な振る舞いを実現するのに便利な方法です。`doesNotUnderstand:` をオーバーライドすることで、オブジェクトは理解できないメッセージに応答するための代替的な戦略を持つことができます。

この手法の最も一般的な応用例は、(1) 軽量プロキシの実装と (2) 足りないコードの動的なコンパイルまたはロード、です。

### 軽量プロキシ

最初の例として、既存のオブジェクトのプロキシとして振る舞う「最小限のオブジェクト」を紹介します。このプロキシは固有のメソッドを実質的に一つも持たないので、どんなメッセージも `doesNotUnderstand:` で捕捉されることになります。このメッセージを実装すれば、プロキシは実体のオブジェクトにメッセージを移譲する前に特別な動作を行うことができます。

これがどのように実装できるのかを見てみましょう。<sup>1</sup>

まず、`LoggingProxy` を次のように定義します：

```
ProtoObject subclass: #LoggingProxy
  instanceVariableNames: 'subject invocationCount'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE-Reflection'
```

Object ではなく ProtoObject のサブクラスとして定義していることに注意してください。Object が持つ 400 個以上のメソッド (!) を継承したくないからです。

<sup>1</sup> `PBE-Reflection` を <http://www.squeaksource.com/PharoByExample/> からロードできます



```
Object methodDict size  → 408
```

このプロキシには二つのインスタンス変数があります。subject はプロキシの実体であり、invocationCount はプロキシがインターセプトしたメッセージの数です。二つのインスタンス変数を初期化し、インターセプトしたメッセージの数を返すアクセサを提供します。初期化時点では、subject 変数はプロキシオブジェクト自身を指しています。

```
LoggingProxy»initialize
  invocationCount := 0.
  subject := self.
```

```
LoggingProxy»invocationCount
  ↑ invocationCount
```

この例では、理解できないすべてのメッセージをインターセプトし、トランスクリプトに表示し、メッセージ受信回数を更新し、メッセージを実体へ転送します。

```
LoggingProxy»doesNotUnderstand: aMessage
  Transcript show: 'performing ', aMessage printString; cr.
  invocationCount := invocationCount + 1.
  ↑ aMessage sendTo: subject
```

ここでちょっとした魔法の登場です。Point オブジェクトと LoggingProxy オブジェクトを生成し、プロキシに「その Point オブジェクトになれ (become:)」と伝えます。

```
point := 1@2.
LoggingProxy new become: point.
```

これにより、イメージ内にある、この Point オブジェクトへのすべての参照をプロキシへの参照へと入れ替えます（逆も同様です）。重要なのは、これによってプロキシの subject インスタンス変数もこの Point オブジェクトを参照するようになることです！

```
point invocationCount  → 0
point + (3@4)          → 4@6
point invocationCount  → 1
```

これはたいていの場合うまく動作するのですが、足りない点もあります：

```
point class  → LoggingProxy
```

興味深いことに、class メソッドは ProtoObject で実装されておらず（Object で実装されています）、LoggingProxy は Object を継承していません！ class はメッセー

ジとして送られるのではなく、仮想マシンによって直接処理されます。<sup>2</sup>

このような特別なメッセージ送信を無視するとしても、この手法では克服不可能な、self 送信をインターセプトできないという根本的な問題があります。

```
point := 1@2.
LoggingProxy new become: point.
point invocationCount  →  0
point rect: (3@4)      →  1@2 corner: 3@4
point invocationCount  →  1
```

このプロキシは rect:メソッド中の二度の self 送信を数え損ねています:

```
Point>rect: aPoint
↑ Rectangle origin: (self min: aPoint) corner: (self max: aPoint)
```

この方法のプロキシを使えばメッセージをインターセプトできますが、プロキシに起因する制限に気をつけてください。14.7 節では、メッセージをインターセプトするためのより一般的な方法を見ていきます。

## 欠けているメソッドを生成する

理解されないメッセージをインターセプトする手法の一般的な応用例として、未実装のメソッドの動的なロードまたは生成が挙げられます。多数のメソッドを持つ巨大なクラスライブラリを考えてみましょう。ライブラリ全体をロードする代わりに各クラスのスタブをロードしてもよいでしょう。このスタブは、そのクラスで定義されたすべてのメソッドのソースコードがどこにあるかを知っています。スタブは理解できないメッセージをすべて捕捉し、欠けているメソッドを必要に応じて動的にロードします。この処理をある時点で無効化すれば、それまでにロードされたコードが特定のクライアントアプリケーションに最低限必要なものとして保存することができます。

この方法のシンプルな応用例を見てみましょう。次に示すクラスは必要に応じてインスタンス変数へのアクセサを自動的に追加します:

```
DynamicAccessors>>doesNotUnderstand: aMessage
| messageName |
messageName := aMessage selector asString.
(self class instVarNames includes: messageName)
ifTrue: [
```

<sup>2</sup>yourselfも同様に、決して送られません。レシーバによって仮想マシンが直接解釈する可能性のあるメッセージを次に示します: +-<> <=> =~=\*! == @ bitShift: // bitAnd: bitOr: at: at:put: size next nextPut: atEnd blockCopy: value value: do: new new: x y 次に示すセレクタは決して送られません。コンパイラが比較とジャンプのバイトコードにインライン展開します: ifTrue: ifFalse: ifTrue:ifFalse: ifFalse:ifTrue: and: or: whileFalse: whileTrue: whileFalse whileTrue to:do: to:by:do: caseOf: caseOf:otherwise: ifNil: ifNotNil: ifNil:ifNotNil: ifNotNil:ifNil: これらのメッセージを論理値ではないオブジェクトに送ろうとした場合も、レシーバが mustBeBooleanをオーバーライドするか、NonBooleanReceiver例外を捕捉して適切な論理値を返せば実行を継続できます。

```
self class compile: messageName, String cr, '↑', messageName.
↑ aMessage sendTo: self ].
↑ super doesNotUnderstand: aMessage
```

理解できないメッセージはすべてここで捕捉されます。送られたメッセージと同名のインスタンス変数があれば、自身のクラスにアクセサをコンパイルするように依頼します。その後、メッセージをもう一度送信します。

DynamicAccessorsクラスに（初期化されていない）インスタンス変数 *x* があり、*x* のアクセサが定義されていないとしましょう。次の式を評価すると、アクセサが動的に生成されて *x* の値を取得できます：

```
myDA := DynamicAccessors new.
myDA x → nil
```

DynamicAccessorsオブジェクトに最初に *x* メッセージが送られたときに何が起きるか、順に見ていきましょう（図 14.9 を参照）。

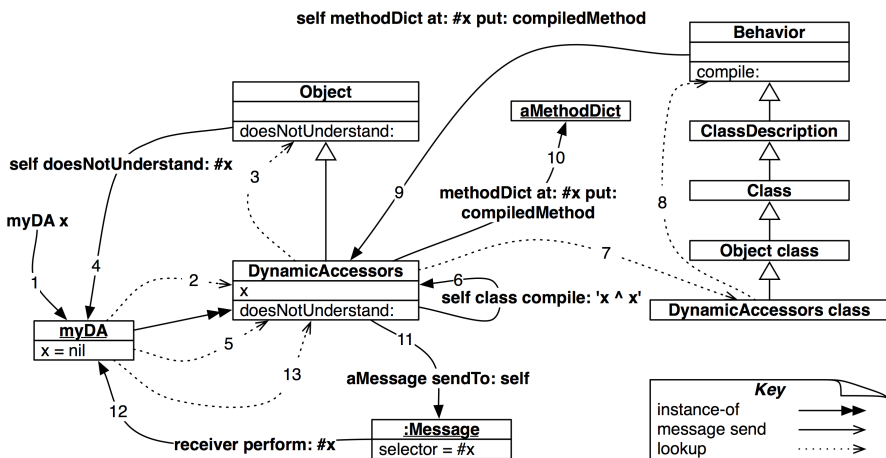


Figure 14.9: アクセサを動的に生成する。

(1) *x* を myDA に送ると (2) メソッドがクラスのメソッド辞書から探索されますが、(3) クラス階層をたどっても見つかりません。(4) 従って self doesNotUnderstand: #*x* がオブジェクトに送り返され、(5) 改めてメソッド探索が行われます。DynamicAccessorsではすぐに doesNotUnderstand: が見つかり、(6) 文字列 'x ^ x' をコンパイルするよう依頼します。(7) compile メソッドが探索され、(8) 最終的に Behavior クラスで見つかります。(9-10) コンパイル済みメソッドが DynamicAccessors のメソッド辞書に追加されます。(11-13) 最後に再び *x* メッセージが再送されますが、今度は対応するメソッドが見つかります。


この方法はインスタンス変数のセッターや、その他のお約束コード、例えば Visitor パターンで使うメソッドの生成などに使えます。

ステップ (13) で使われている `Object»perform:` に注目してください。このメソッドを使うと、実行時に動的にメッセージを生成して送れます。


```
5 perform: #factorial → 120
6 perform: ('fac', 'torial') asSymbol → 720
4 perform: #max: withArguments: (Array with: 6) → 6
```

## 14.7 メソッドラッパーとしてのオブジェクト

これまで見てきたように、Smalltalk ではコンパイル済みメソッドも通常のオブジェクトであり、プログラマーが使うことのできる多くの問い合わせ用のメソッドを用意しています。ただし、どんなオブジェクトでもコンパイル済みメソッドの役を演じることができるということはちょっと意外かもしれません。そのオブジェクトは、`run:with:in:` およびその他いくつかの重要なメッセージに応答しさえすればいいのです。

 空のクラス `Demo` を定義します。 `Demo new answer42` を評価して、いつもの *"Message Not Understood"* エラーの発生を確認してください。

ここで、普通のオブジェクトを `Demo` クラスのメソッド辞書にインストールします。

 `Demo methodDict at: #answer42 put: ObjectsAsMethodsExample new.` を評価し、再び `Demo new answer42` の結果を表示してください。今度は `42` という回答を得られます。

`ObjectsAsMethodsExample` クラス定義を調べると次の二つのメソッドが見つかります:

```
answer42
↑42

run: oldSelector with: arguments in: aReceiver
↑self perform: oldSelector withArguments: arguments
```

`Demo` のインスタンスが `answer42` メッセージを受け取るといつも通りにメソッド探索が行われますが、仮想マシンは普通のオブジェクトがコンパイル済みメソッド役を演じようとしていることを検出します。そこで、仮想マシンは `ObjectsAsMethodsExample` オブジェクトに `run:with:in:` メッセージを送ります。このメッセージの引数は最初に送られたメソッドセレクタ、引数、レシーバです。 `ObjectsAsMethodsExample` は `run:with:in:` メソッドを実装しているので、メッセージをインターセプトして自身に移譲します。

次のようにすれば、この偽のメソッドを削除できます:

```
Demo methodDict removeKey: #answer42 ifAbsent: []
```

ObjectsAsMethodsExampleをさらに詳しく調べると、スーパークラスが flushCache、methodClass:、selector:のメソッドを定義していることがわかりますが、定義だけで何も実装されていません。これらのメッセージはコンパイル済みメソッドに送られる可能性があるので、コンパイル済みメソッドのふりをするために実装しておく必要があります (flushCacheはこの中で最も重要なメソッドです。他のメソッドは Behavior»addSelector:withMethod: または MethodDictionary »at:put: でメソッドをインストールする場合に必要になります)。

## メソッドラッパーをテストカバレッジの評価に使う

メソッドラッパーはメッセージをインターセプトする有名な方法です<sup>3</sup>。オリジナルの実装<sup>4</sup>では、メソッドラッパーは CompiledMethod のサブクラスのインスタンスです。インストールされると、メソッドラッパーは元のメソッドの起動前後に特別な動作を実行できます。アンインストールすると、元のメソッドがメソッド辞書の然るべき位置に戻されます。

Pharo ではメソッドラッパー を簡単に実装できます。CompiledMethod のサブクラスを作る代わりに run:with:in: を実装します。実際、メソッドラッパーとしてのオブジェクトの、軽量の実装<sup>5</sup>が存在します。しかし、これを書いている時点では、この実装は標準の Pharo には組み込まれていません。

いずれにせよ、Pharo のテストランナーはまさにこの手法を使ってテストカバレッジを評価しています。テストランナーがどのように動作するのか見てみましょう。

テストカバレッジの開始点は TestRunner»runCoverage です:

```
TestRunner»runCoverage
| packages methods |
... "カバレッジを測定するメソッド" を決定する"
self collectCoverageFor: methods
```

TestRunner»collectCoverageFor: メソッドに、カバレッジを調べるアルゴリズムがわかりやすく書かれています:

```
TestRunner»collectCoverageFor: methods
| wrappers suite |
wrappers := methods collect: [ :each | TestCoverage on: each ].
suite := self
  reset;
  suiteAll.
[ wrappers do: [ :each | each install ].
 [ self runSuite: suite ] ensure: [ wrappers do: [ :each | each uninstall ] ]
valueUnpreemptively.
```

<sup>3</sup>John Brant et al., Wrappers to the Rescue. In Proceedings European Conference on Object Oriented Programming (ECOOP'98). Volume 1445, Springer-Verlag 1998.

<sup>4</sup><http://www.squeaksource.com/MethodWrappers.html>

<sup>5</sup><http://www.squeaksource.com/ObjectsAsMethodsWrap.html>

```

wrappers := wrappers reject: [ :each | each hasRun ].
wrappers isEmpty
  ifTrue:
    [ UIManager default inform: 'Congratulations. Your tests cover all code under
      analysis.' ]
  ifFalse: ...

```

カバレッジを調べたいメソッドごとにラッパーが生成され、それらがインストールされます。ここでテストが実行され、テスト実行後にラッパーがアンインストールされます。最終的にカバーされていないメソッドに関するフィードバックを得られます。

ラッパー自身はどのように動くのでしょうか？ `TestCoverage`ラッパーには3個のインスタンス変数 (`hasRun`、`reference`、`method`)があり、次のように初期化されます。

```

TestCoverage class>>on: aMethodReference
  ↑ self new initializeOn: aMethodReference

TestCoverage>>initializeOn: aMethodReference
  hasRun := false.
  reference := aMethodReference.
  method := reference compiledMethod

```

`install` メソッドと `uninstall` メソッドは単純な方法でメソッド辞書を更新します:

```

TestCoverage>>install
  reference actualClass methodDictionary
  at: reference methodSymbol
  put: self

TestCoverage>>uninstall
  reference actualClass methodDictionary
  at: reference methodSymbol
  put: method

```

`run:with:in:`メソッドは `hasRun`インスタンス変数を更新し、ラッパーをアンインストールし (カバレッジが検証されたので不要になります)、元のメソッドにメッセージを再送します。

```

run: aSelector with: anArray in: aReceiver
  self mark; uninstall.
  ↑ aReceiver withArgs: anArray executeMethod: method

mark
  hasRun := true

```

(ProtoObject»withArgs:executeMethod: を調べて、メソッド辞書から取り除かれたメソッドがどう起動されるのか確認してください。)

たったこれだけです！

メソッドラッパーを使えば、あらゆる種類の動作をメソッドの通常処理の前後で実行することができます。典型的な応用例は計測（メソッドの呼び出しパターンに関する統計情報の収集）、選択的な事前・事後条件の確認、メモ化（メソッドの処理結果を選択的にキャッシュすること）です。

## 14.8 プラグマ

プラグマとはプログラムに付与する注釈の一種で、プログラムに関連するデータを追加することができます。プラグマはメソッド実行に対して直接的な影響は及ぼしません。プラグマには以下のようなものを含むいくつかの使い道があります：

- コンパイラへの情報: プラグマはコンパイラにメソッド呼び出しをプリミティブ関数として扱うよう指示します。プリミティブ関数は仮想マシンか外部プラグインで定義する必要があります。
- 実行時処理: プラグマによって付加された情報を実行時に検証することができます。

プラグマはメソッド宣言にのみ適用できます。メソッドには複数のプラグマを宣言でき、Smalltalk 文より前に宣言する必要があります。プラグマはリテラルを引数に取る静的なメッセージ送信と言えます。

この章の最初でプリミティブを紹介したときにプラグマが出てきました。プリミティブはプラグマ宣言以上のもではありません。instVarAt:メソッド内の<primitive: 73>を見てください。このプラグマのセレクトは primitive:で、引数はリテラル 73です。

おそらくコンパイラはプラグマの大口ユーザです。SUnit も注釈を活用するツールです。SUnit はテスト単体からアプリケーションのカバレッジを推定することができます。しかし、カバレッジから一部のメソッドを除外したくなることもあるでしょう。SplitJointTest classの documentationメソッドはそのような例となっています：

```
SplitJointTest class»documentation
<ignoreForCoverage>
"self showDocumentation"

↑'This package provides function.... "'
```

メソッドに<ignoreForCoverage>プラグマを加えるだけで、カバレッジの適用範囲を制御できます。

プラグマは `Pragma` のインスタンスであり、1 級のオブジェクトです。コンパイル済みメソッドに `pragmas` メッセージを送ると、プラグマの配列を返します。

```
(SplitJoinTest class >> #showDocumentation) pragmas.  
  → an Array(<ignoreForCoverage>)  
(Float>>#+) pragmas → an Array(<primitive: 41>)
```

(以下の `allNamed:in:` メソッドにより) 特定のプラグマを定義しているメソッドをクラスから取得することもできます。 `SplitJoinTest` には `<ignoreForCoverage>` と注釈されたクラスメソッドがあります:

```
Pragma allNamed: #ignoreForCoverage in: SplitJoinTest class → an Array(<  
  ignoreForCoverage> <ignoreForCoverage> <ignoreForCoverage>)
```

`allNamed:in:` の変種が、`Pragma` のクラスメソッドとして定義されています。

プラグマは自身が定義されたメソッド (`method` を使います)、そのメソッド名 (`selector`)、そのメソッドを持つクラス (`methodClass`)、引数の数 (`numArgs`)、リテラルを引数として持つかどうか (`hasLiteral:` と `hasLiteralSuchThat:`) を知っています。

## 14.9 章のまとめ

リフレクションとは、ランタイムシステムのメタオブジェクトに対して普通のオブジェクトと同様に問い合わせたり、検証したり、変更すらできる機能です。

- インспекタは `instVarAt:` や関連するメソッドを使って、オブジェクトの「プライベートな」インスタンス変数に問い合わせたり、変数の内容を変更したりします。
- クラスの全インスタンスを取得したいときは、`Behavior»allInstances` を送ります。
- `class`、`isKindOf:`、`respondsTo:` などのメソッドはメトリクスの収集や開発ツールの実装には便利ですが、通常のアプリケーションでは避けるべきです: これらのメソッドはオブジェクトのカプセル化の原則に反しており、コードの理解や保守の邪魔になります。
- `SystemNavigation` はクラス階層の探索や閲覧に便利な多くの問い合わせ方法を持つユーティリティクラスです。例えば、ある文字列をソースコードに含むメソッドをすべて見つけ出すには `SystemNavigation default browseMethodsWithSourceString: 'pharo'` を使います (遅いですが、完璧です! )。
- どのクラスも `MethodDictionary` のインスタンスを参照しています。メソッド辞書はセレクタとコンパイル済みメソッドのインスタンスを結び付け



ています。コンパイル済みメソッドは自身が所属するクラスを知っており、お互いの参照が循環しています。

- **MethodReference**はコンパイル済みメソッドのための軽量プロキシです。便利なメソッドを提供しており、多くの **Smalltalk** のツールに使われています。
- リファクタリングブラウザの基盤の一部である **BrowserEnvironment**を使えば、**SystemNavigation**よりも洗練されたインターフェースでシステムに問い合わせられます。問い合わせた結果を、次の新しい問い合わせの対象範囲として使えます。GUI にとプログラマ的なインターフェースのどちらも利用できます。
- **thisContext**は仮想マシンの実行時スタックを具現化した擬似変数です。主にデバッガで使われ、スタックの対話的な操作を動的に実現しています。メッセージのセンダを動的に調べる場合にも便利です。
- 賢いブレークポイントを設定するには **haltIf:**を使います。このメソッドは引数にメソッドセレクタを取り、実行時スタックのセンダが同名のメソッドである場合のみ停止します。
- 対象へのメッセージをインターセプトする一般的な方法は「最小限のオブジェクト」をプロキシとして使うことです。プロキシが実装するメソッドをできるだけ減らし、**doesNotUnderstand:**を実装してすべてのメッセージ送信を捕捉します。これで追加のアクションを実行してから対象にメッセージを転送できます。
- プロキシと主体のような二つのオブジェクトへの参照を交換するには **become:**を使います。
- **class**や **yourself**のように、実際には送られず仮想マシンによって解釈されるメッセージに気をつけてください。他にも **+-**、**ifTrue:**などのメッセージは、レシーバによっては仮想マシンに直接解釈されたり、インライン展開されたりする可能性があります。
- 欠けているメソッドの遅延ロードまたは遅延コンパイルのために **doesNotUnderstand:**をオーバーライドすることもよくあります。
- **doesNotUnderstand:**は **self** 送信を捕捉できません。
- より正確にメッセージをインターセプトするには、オブジェクトをメソッドラッパーとして使います。メソッドラッパーはコンパイル済みメソッドの代わりにメソッド辞書にインストールされます。メソッドラッパーは **run:with:in:**を実装すべきです。このメッセージは仮想マシンがメソッド辞書からコンパイル済みメソッドの代わりに普通のオブジェクトを検出したときに送られます。この方法は **SUnit** テストランナーでカバレッジデータを集めるのに使われています。



Part IV

# **Appendices**



# Appendix A

## よくある質問

### A.1 はじめに

**FAQ 1** 最新の *Pharo* はどこで入手できますか？

**Answer** <http://www.pharo-project.org/>

**FAQ 2** この本ではどの *Pharo* のイメージを使う べきですか？

**Answer** どの *Pharo* イメージを使っても大丈夫ですが、*Pharo by Example* の web サイト (<http://PharoByExample.org>) に置いてあるイメージを使うことをお勧めします。ほとんどの場合、他のイメージも使用できますが、演習によっては驚くほど動作が異なることがあります。

### A.2 コレクション

**FAQ 3** `OrderedCollection` を並べ替えるにはどう すればいいですか？

**Answer** インスタンスに `asSortedCollection` メッセージを送ります。

```
#(7 2 6 1) asSortedCollection → a SortedCollection(1 2 6 7)
```

**FAQ 4** 文字の配列を文字列 (`String`) に変換するにはどう すればいいですか？

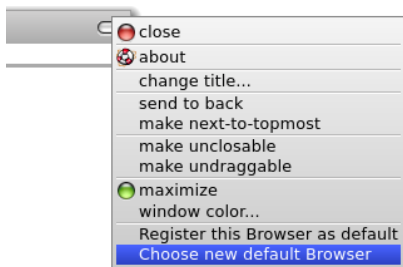
## Answer

```
String streamContents: [:str | str nextPutAll: 'hello' asSet] → 'hlelo'
```

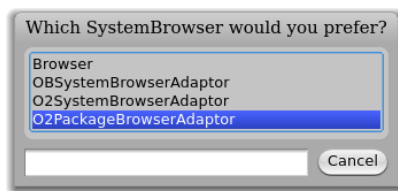
### A.3 システムブラウザ

**FAQ 5** ブラウザがこの本で使用されているものとは違います。どうすればいいですか？

**Answer** おそらくデフォルトのブラウザとして、異なるバージョンの **OmniBrowser** がインストールされたイメージを使用しているからです。この本では、デフォルトとして **OmniBrowser** の (*Package Browser*) がインストールされていることを前提としています。デフォルトブラウザはブラウザのメニューバーをクリックすると変更できます。ウィンドウの右上隅にある灰色の角丸長方形のボタンをクリックし、“Choose new default Browser” を選択し、**O2PackageBrowserAdaptor** を選んでください。次回以降開かれるブラウザはパッケージブラウザになります。



(a) Choose new default browser を選択



(b) O2PackageBrowserAdaptor を選択

Figure A.1: デフォルトブラウザの変更

**FAQ 6** クラスを検索するにはどうすればいいですか？

**Answer** クラス名の文字列を選択して **CMD-b (browse)** か、ブラウザのカテゴリペイン上にマウスカーソルがある状態で **CMD-f** です。

**FAQ 7** *super* にメッセージを送っているすべてのメソッドを検索したりブラウザするにはどうすればいいですか？

**Answer** 二つ目の手法の方が高速です:

```
SystemNavigation default browseMethodsWithSourceString: 'super'.
SystemNavigation default browseAllSelect: [:method | method sendsToSuper ].
```

**FAQ 8** あるクラス階層内で、*super* にメッセージを送っているメソッドをすべてブラウズするにはどうすればいいですか?

**Answer**

```
browseSuperSends := [:aClass | SystemNavigation default
  browseMessageList: (aClass withAllSubclasses gather: [ :each |
    (each methodDict associations
      select: [ :assoc | assoc value sendsToSuper ])
      collect: [ :assoc | MethodReference class: each selector: assoc key ] ])
  name: 'Supersends of ', aClass name, ' and its subclasses'].
browseSuperSends value: OrderedCollection.
```

**FAQ 9** サブクラスで新たに追加されたメソッド (親クラスのメソッドをオーバーライドしていないメソッド) を見つけるには、どうすればいいですか?

**Answer** Trueクラスで追加されたメソッドを調べる例です:

```
newMethods := [:aClass | aClass methodDict keys select:
  [:aMethod | (aClass superclass canUnderstand: aMethod) not ]].
newMethods value: True → an IdentitySet(#asBit #xor:)
```

**FAQ 10** クラスの抽象メソッドはどうすればわかりますか?

**Answer**

```
abstractMethods :=
  [:aClass | aClass methodDict keys select:
    [:aMethod | (aClass>>aMethod) isAbstract ]].
abstractMethods value: Collection → an IdentitySet(#remove:ifAbsent: #add: #do:)
```

**FAQ 11** AST (抽象構文木) のビューを生成するにはどうすればいいですか?

**Answer** squeaksource.com から AST パッケージをロードし、

```
(RBParser parseExpression: '3+4') explore
```

を実行します。(または *explore it* します)

**FAQ 12** システムにあるすべてのトレイトを見つけるにはどうすればいいですか?

**Answer**

```
Smalltalk allTraits
```

**FAQ 13** トレイトを使用しているクラスを探すにはどうすればいいですか？

**Answer**

```
Smalltalk allClasses select: [:each | each hasTraitComposition and: [each  
traitComposition notEmpty ]]
```

## A.4 Monticello と SqueakSource を使う

**FAQ 14** *SqueakSource* からプロジェクトをロードするにはどうすればいいですか？

**Answer**

1. ロードしたいプロジェクトを <http://squeaksource.com> から探します
2. プロジェクトのリポジトリ情報をクリップボードにコピーします
3. **open** ▾ **Monticello browser** を選択して Monticello ブラウザを開きます
4. **+Repository** ▾ **HTTP** を選択します
5. リポジトリ情報をペーストし、必要ならパスワードを入力して OK します
6. できたりポジトリを選択して **Open** を選択します
7. 最新のバージョンのファイル名を選択してロードします

**FAQ 15** *SqueakSource* にプロジェクトを新規作成するにはどうすればいいですか？

**Answer**

1. ウェブブラウザで <http://squeaksource.com> を開きます
2. メンバ登録を行います
3. プロジェクトを登録します (名前=カテゴリとなります)
4. リポジトリ情報をクリップボードにコピーします



5. **open** ▷ **Monticello browser** を選択して Monticello ブラウザを開きます
6. **+Package** を選択してカテゴリを登録します
7. パッケージを選択します
8. **+Repository** ▷ **HTTP** を選択します
9. リポジトリ情報をペーストし、必要ならパスワードを入力して OK します
10. **Save** ボタンで最初のバージョンを保存します

**FAQ 16** 既存の Number クラスに追加した Number»chf メソッドを、自分の Money プロジェクトのパッケージに入れるにはどうすればいいですか？

**Answer** そのメソッドのメソッドカテゴリを \*Money にします。Monticello は '\*package' のような名前で始まるカテゴリのメソッドをまとめて、そのパッケージに入れます。

## A.5 ツール

**FAQ 17** プログラムソース中から *SUnit TestRunner* を開くにはどうすればいいですか？

**Answer** TestRunner open を実行します

**FAQ 18** *Refactoring Browser* はどこにありますか？

**Answer** squeaksource.com から AST パッケージをロードし、その後リファクタリングエンジンをロードしてください: <http://www.squeaksource.com/AST>  
<http://www.squeaksource.com/RefactoringEngine>

**FAQ 19** デフォルトのブラウザを設定するにはどうすればいいですか？

**Answer** ブラウザウィンドウの右上隅にある灰色のボタンをクリックします。

## A.6 正規表現と構文解析

**FAQ 20** *RegEx(正規表現)* パッケージに関するドキュメントはどこにありますか？

**Answer** VB-Regex カテゴリにある RxParser class クラスの **DOCUMENTATION** プロトコルを参照してください。

**FAQ 21** パーサを書くための何かいいツールはありますか？

**Answer** SmaCC (the Smalltalk Compiler Compiler) を使ってください。その場合少なくとも SmaCC-lr.13 以降をインストールする必要があります。http://www.squeaksource.com/SmaccDevelopment.html からロードできます。ここにはとてもよくできたオンラインチュートリアルがあります: http://www.refactory.com/Software/SmaCC/Tutorial.html

**FAQ 22** パーサを書くためには *SqueakSource* の *SmaccDevelopment* にあるどのパッケージをロードすればいいですか？

**Answer** SmaCCDev の最新バージョンをロードしてください。ランタイムも含まれています (SmaCC-Development は Squeak 3.8 用です)。

# Bibliography

- Sherman R. Alpert, Kyle Brown and Bobby Woolf:** The Design Patterns Smalltalk Companion. Addison Wesley, 1998, ISBN 0-201-18462-1
- Kent Beck:** Smalltalk Best Practice Patterns. Prentice-Hall, 1997
- Kent Beck:** Test Driven Development: By Example. Addison-Wesley, 2003, ISBN 0-321-14653-0
- John Brant et al.:** Wrappers to the Rescue. In Proceedings European Conference on Object Oriented Programming (ECOOP'98). Volume 1445, Springer-Verlag 1998, 396-417
- Erich Gamma et al.:** Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Mass.: Addison Wesley, 1995, ISBN 0-201-63361-2-(3)
- Adele Goldberg and David Robson:** Smalltalk 80: the Language and its Implementation. Reading, Mass.: Addison Wesley, May 1983, ISBN 0-201-13688-0
- Dan Ingalls et al.:** Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97). ACM Press, November 1997 (URL: <http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html>), 318-326
- Wilf LaLonde and John Pugh:** Inside Smalltalk: Volume 1. Prentice Hall, 1990, ISBN 0-13-468414-1
- Alec Sharp:** Smalltalk by Example. McGraw-Hill, 1997 (URL: <http://stephane.ducasse.free.fr/FreeBooks/ByExample/>)
- Bobby Woolf:** Null Object. In **Robert Martin, Dirk Riehle and Frank Buschmann, editors:** Pattern Languages of Program Design 3. Addison Wesley, 1998, 5-18









