

Pharo par l'Exemple

Andrew Black Stéphane Ducasse
Oscar Nierstrasz Damien Pollet

avec l'aide de Damien Cassou et Marcus Denker

Traduit en français par :

A VENIR

Version du 10 novembre 2010

Ce livre est disponible en libre téléchargement depuis <http://PharoByExample.org/fr> sous le titre *Pharo par l'exemple, ISBN XXX-X-XXXXXX-X-X. La première édition a été publiée en Décembre 2009. Couverture par Samuel Morello.* Vous pouvez vous procurer une copie à l'adresse : <http://PharoByExample.org/fr>.

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

Le contenu de ce livre est protégé par la licence Creative Commons Paternité Version 3.0 de la licence générique - Partage des Conditions Initiales à l'Identique.

Vous êtes libres :

de reproduire, distribuer et communiquer cette création au public

de modifier cette création

Selon les conditions suivantes :

Paternité. Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).

Partage des Conditions Initiales à l'Identique. Si vous transformez ou modifiez cette œuvre pour en créer une nouvelle, vous devez la distribuer selon les termes du même contrat ou avec une licence similaire ou compatible.

- À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web :
<http://creativecommons.org/licenses/by-sa/3.0/deed.fr>
- Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre.
- Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Ce qui précède n'affecte en rien vos droits en tant qu'utilisateur (exceptions au droit d'auteur : copies réservées à l'usage privé du copiste, courtes citations, parodie, ...). Ceci est le Résumé Explicatif du Code Juridique (la version intégrale du contrat) :

<http://creativecommons.org/licenses/by-sa/3.0/legalcode>



Table des matières

	Préface	ix
I	Comment démarrer	
1	Une visite de Pharo	3
1.1	Premiers pas.	3
1.2	Le menu World.	8
1.3	Envoyer des messages	9
1.4	Enregistrer, quitter et redémarrer une session Pharo.	10
1.5	Les fenêtres Workspace et Transcript	12
1.6	Les raccourcis-clavier	13
1.7	Le navigateur de classes Class Browser	16
1.8	Trouver des classes	18
1.9	Trouver des méthodes	20
1.10	Définir une nouvelle méthode	22
1.11	Résumé du chapitre	27
2	Une première application	29
2.1	Le jeu Lights Out	29
2.2	Créer un nouveau paquetage.	30
2.3	Définir la classe LOCell.	30
2.4	Ajouter des méthodes à la classe	33
2.5	Inspecter un objet	35
2.6	Définir la classe LOGame	36
2.7	Organiser les méthodes en protocoles	39

2.8	Essayons notre code	43
2.9	Sauvegarder et partager le code Smalltalk	46
2.10	Résumé du chapitre	51
3	Un résumé de la syntaxe	53
3.1	Les éléments syntaxiques	53
3.2	Les pseudo-variables	56
3.3	Les envois de messages	57
3.4	Syntaxe relative aux méthodes	58
3.5	La syntaxe des blocs	60
3.6	Conditions et itérations	61
3.7	Primitives et Pragmas	63
3.8	Résumé du chapitre	64

4	Comprendre la syntaxe des messages	65
4.1	Identifier les messages	65
4.2	Trois sortes de messages	67
4.3	Composition de messages	69
4.4	Quelques astuces pour identifier les messages à mots-clés	76
4.5	Séquences d'expression	78
4.6	Cascades de messages	78
4.7	Résumé du chapitre	79

II Développer avec Pharo

5	Le modèle objet de Smalltalk	83
5.1	Les règles du modèle	83
5.2	Tout est objet	84
5.3	Tout objet est instance de classe	84
5.4	Toute classe a une super-classe	92
5.5	Tout se passe par envoi de messages	96
5.6	La recherche de méthode suit la chaîne d'héritage	98
5.7	Les variables partagées	104
5.8	Résumé du chapitre	110

6	L'environnement de programmation de Pharo	113
6.1	Une vue générale	114
6.2	Le Browser	115
6.3	Monticello	128
6.4	L'inspecteur et l'explorateur	135
6.5	Le débogueur	139
6.6	Le navigateur de processus	148
6.7	Trouver les méthodes	149
6.8	Change set et son gestionnaire Change Sorter.	151
6.9	Le navigateur de fichiers File List Browser	153
6.10	En Smalltalk, pas de perte de codes	155
6.11	Résumé du chapitre	156
7	SUnit	159
7.1	Introduction	159
7.2	Pourquoi tester est important	160
7.3	De quoi est fait un bon test ?	161
7.4	SUnit par l'exemple	162
7.5	Les recettes pour SUnit	167
7.6	Le <i>framework</i> SUnit	168
7.7	Caractéristiques avancées de SUnit	171
7.8	La mise en œuvre de SUnit	172
7.9	Quelques conseils sur les tests	175
7.10	Résumé du chapitre	177
8	Les classes de base	179
8.1	Object	179
8.2	Les nombres.	189
8.3	Les caractères	193
8.4	Les chaînes de caractères	194
8.5	Les booléens.	195
8.6	Résumé du chapitre	197

9	Les collections	199
9.1	Introduction	199
9.2	Des collections très variées	200
9.3	Les implémentations des collections	203
9.4	Exemples de classes importantes	204
9.5	Les collections itératrices ou iterators	215
9.6	Astuces pour tirer profit des collections	219
9.7	Résumé du chapitre	220
10	Streams : les flux de données	223
10.1	Deux séquences d'éléments	223
10.2	Streams contre Collections	224
10.3	Utiliser les streams avec les collections	225
10.4	Utiliser les streams pour accéder aux fichiers	233
10.5	Résumé du chapitre	236
11	L'interface Morphic	237
11.1	Première immersion dans Morphic	237
11.2	Manipuler les morphs	239
11.3	Composer des morphs	240
11.4	Dessiner ses propres morphs	241
11.5	Interaction et animation	245
11.6	Le glisser-déposer	250
11.7	Le jeu du dé	253
11.8	Gros plan sur le canevas	257
11.9	Résumé du chapitre	258
12	Seaside par l'exemple	259
12.1	Pourquoi avons-nous besoin de Seaside ?	259
12.2	Démarrer avec Seaside	260
12.3	Les composants Seaside	266
12.4	Le rendu XHTML	269
12.5	Les feuilles de style CSS	276
12.6	Gérer les flux de contrôle	278

12.7	Un tutoriel complet	285
12.8	Un bref coup d'œil sur la technologie AJAX	293
12.9	Résumé du chapitre	296

III Pharo avancé

13	Classes et métaclasses	301
13.1	Les règles pour les classes et les métaclasses	301
13.2	Retour sur le modèle objet de Smalltalk	302
13.3	Toute classe est une instance d'une métaclass	304
13.4	La hiérarchie des métaclasses est parallèle à celle des classes .	305
13.5	Toute métaclass hérite de Class et de Behavior	307
13.6	Toute métaclass est une instance de Metaclass	310
13.7	La métaclass de Metaclass est une instance de Metaclass . .	311
13.8	Résumé du chapitre	312
14	La réflexivité	315
14.1	Introspection	316
14.2	Parcourir le code	321
14.3	Classes, dictionnaires de méthodes et méthodes	323
14.4	Environnements de navigation du code	325
14.5	Accéder au contexte d'exécution	327
14.6	Intercepter les messages non compris.	330
14.7	Objects as method wrappers	334
14.8	Pragmas	337
14.9	Résumé du chapitre	338

IV Annexes

A	Foire Aux Questions	343
A.1	Prémises.	343
A.2	Collections	343
A.3	Naviguer dans le système	344
A.4	Utilisation de Monticello et de SqueakSource	346
A.5	Outils	347

A.6	Expressions régulières et analyse grammaticale	348
Bibliographie		349

Préface

Qu'est ce que Pharo ?

Pharo est une implémentation moderne, libre et complète du langage de programmation Smalltalk et de son environnement. Pharo est un *fork*¹ de Squeak², une réécriture du classique système Smalltalk-80. Alors que Squeak fut développé principalement en tant que plateforme pour le développement de logiciels éducatifs expérimentaux, Pharo tend à offrir une plateforme, à la fois, *open-source* et épurée pour le développement de logiciels professionnels et aussi, stable et robuste pour la recherche et le développement dans le domaine des langages et environnement dynamiques. Pharo est le système de référence de Seaside : le framework³ (dit aussi “cadre d’applications”) destiné au développement web.

Pharo résout les problèmes de licence inhérent à Squeak. Contrairement aux versions précédentes de Squeak, le noyau de Pharo ne contient que du code sous licence MIT. Le projet Pharo a débuté en mars 2008 depuis un *fork* de la version 3.9 de Squeak et la première version 1.0 *beta* a été publiée le 31 juillet 2009.

Bien que dépourvu de nombreux paquetages présents dans Squeak, Pharo est fourni avec beaucoup de fonctionnalités optionnelles dans Squeak. Par exemple, les fontes TrueType sont inclus dans Pharo. Pharo dispose aussi du support pour de véritables fermetures lexicales ou *block closures*. Les éléments d’interface utilisateurs ont été revus et simplifiés.

Pharo est extrêmement portable — même sa machine virtuelle est entièrement écrite en Smalltalk, ce qui facilite son débogage, son analyse et les

1. Projet dérivé d'une version d'un logiciel pré-existant.

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

3. Un *framework* est un ensemble de bibliothèques, d’outils et de conventions permettant le développement d’applications.

modifications à venir. Pharo est le véhicule de tout un ensemble de projets innovants, des applications multimédias et éducatives aux environnements de développement pour le web. Il est important de préciser le fait suivant concernant Pharo : Pharo ne devrait pas être qu'une simple copie du passé mais véritablement une *réinvention* de Smalltalk. Pour autant, les approches en *tabula rasa* fonctionnent rarement. Pharo encourage les changements évolutifs et incrémentaux. Nous voulons être capable d'expérimenter via les nouvelles fonctionnalités et autre bibliothèques. Par *évolution*, nous disons que Pharo tolèrent les erreurs et n'a pas pour objectif de devenir la prochaine solution de rêve d'un bond — même si nous le désirons. Pharo favorisera toutes les évolutions à caractère incrémental. Le succès de Pharo dépend des contributions de sa communauté.

Qui devrait lire ce livre ?

Ce livre est fondé sur *Squeak Par l'Exemple*⁴, une introduction à Squeak éditée en *open-source*. Il a néanmoins été librement adapté pour refléter les différences entre Pharo et Squeak. Ce livre présente différents aspects de Pharo, en commençant par les concepts de base et en poursuivant vers des sujets plus avancés.

Ce livre ne vous apprendra pas à programmer. Le lecteur doit avoir quelques notions concernant les langages de programmation. Quelques connaissances sur la programmation orientée objet seront utiles.

Ce livre introduit l'environnement de programmation, le langage et les outils de Pharo. Vous serez confronté à de nombreuses bonnes pratiques de Smalltalk, mais l'accent sera mis plus particulièrement sur les aspects techniques et non sur la conception orientée objet. Nous vous présenterons, autant que possible, une foule d'exemples (nous avons été inspiré par l'excellent livre de Alec Sharp sur Smalltalk⁵).

Il y a plusieurs autres livres sur Smalltalk disponibles gratuitement sur le web mais aucun d'entre eux ne se concentrent sur Pharo. Voyez par exemple : <http://stephane.ducasse.free.fr/FreeBooks.html>

Un petit conseil

Ne soyez pas frustré par des éléments de Smalltalk que vous ne comprenez pas immédiatement. Vous n'avez pas tout à connaître ! Alan Knight

4. <http://SqueakByExample.org/fr>; traduction française de Squeak By Example (<http://SqueakByExample.org>).

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

exprime ce principe comme suit⁶ :

Ne vous en préoccuez pas !* Les développeurs Small-talk débutants ont souvent beaucoup de difficultés car ils pensent qu'il est nécessaire de connaître tous les détails d'une chose avant de l'utiliser. Cela signifie qu'il leur faut un moment avant de maîtriser un simple : Transcript show: 'Hello World'. Une des grandes avancées de la programmation par objets est de pouvoir répondre à la question "Comment ceci marche ?" avec "Je ne m'en préoccupe pas".

*. Dans sa version originale : "Try not to care".

Un livre ouvert

Ce livre est ouvert dans plusieurs sens :

- Le contenu de ce livre est diffusé sous la licence Creative Commons Partenité - Partage des Conditions Initiales à l'Identique. En résumé, vous êtes autorisé à partager librement et à adapter ce livre, tant que vous respectez les conditions de la licence disponible à l'adresse suivante : <http://creativecommons.org/licenses/by-sa/3.0/>.
- Ce livre décrit simplement les concepts de base de Pharo. Idéalement, nous voulons encourager de nouvelles personnes à contribuer à des chapitres sur des parties de Pharo qui ne sont pas encore décrites. Si vous voulez participer à ce travail, merci de nous contacter. Nous voulons voir ce livre se développer !

Plus de détails concernant ce livre sont disponibles sur le site web <http://PharoByExample.org/fr>.

La communauté Pharo

La communauté Pharo est amicale et active. Voici une courte liste de ressources que vous pourrez trouver utiles :

- <http://www.pharo-project.org> est le site web principal de Pharo.
- <http://www.squeaksource.com> : SqueakSource est l'équivalent de SourceForge pour les projets Pharo. De nombreux paquetages optionnels se trouvent ici.

6. <http://www.surfscranton.com/architecture/KnightsPrinciples.htm>

Exemples et exercices

Nous utilisons deux conventions typographiques dans ce livre.

Nous avons essayé de fournir autant d'exemples que possible. Il y a notamment plusieurs exemples avec des fragments de code qui peuvent être évalués. Nous utilisons le symbole → afin d'indiquer le résultat qui peut être obtenu en sélectionnant l'expression et en utilisant l'option print it du menu contextuel :

`3 + 4 → 7 "Si vous sélectionnez 3+4 et 'print it', 7 s'affichera"`

Si vous voulez découvrir Pharo en vous amusant avec ces morceaux de code, sachez que vous pouvez charger un fichier texte avec la totalité des codes d'exemples via le site web du livre : <http://PharoByExample.org/fr>.

La deuxième convention que nous utilisons est l'icône pour vous indiquer que vous avez quelque chose à faire :

Avancez et lisez le prochain chapitre !

Remerciements pour l'édition anglaise

Nous voulons remercier Hilaire Fernandes et Serge Stinckwich qui nous ont autorisé à traduire des parties de leurs articles sur Smalltalk et Damien Cassou pour sa contribution au chapitre sur les flots de données ou streams.

Nous remercions particulièrement Alexandre Bergel, Orla Greevy, Fabrizio Perin, Lukas Renggli, Jorge Ressia et Erwann Wernli pour leurs corrections détaillées de l'édition originale.

Nous remercions l'Université de Berne en Suisse pour le soutien gracieusement offert à cette entreprise Open Source et pour les facilités d'hébergement web de ce livre.

Nous remercions aussi la communauté Squeak pour son soutien et son enthousiasme sur ce projet et pour sa communication quant à l'aide à la correction de la première édition de la version originale de ce livre.

Remerciements pour l'édition française

L'édition française de ce livre a été réalisée par l'équipe de traducteurs et de relecteurs suivantes : [A VENIR](#).

Première partie

Comment démarrer

Chapitre 1

Une visite de Pharo

Nous vous proposons dans ce chapitre une première visite de Pharo afin de vous familiariser avec son environnement. De nombreux aspects seront abordés ; il est conseillé d'avoir une machine prête à l'emploi pour suivre ce chapitre.

Cette icône  dans le texte signalera les étapes où vous devrez essayer quelque chose par vous-même. Vous apprendrez à lancer Pharo et les différentes manières d'utiliser l'environnement et les outils de base. La création des méthodes, des objets et les envois de messages seront également abordés.

1.1 Premiers pas

Pharo est librement disponible au téléchargement depuis le site web : <http://pharo-project.org>. Vous devez y télécharger 3 archives (pour 4 fichiers principaux qui constituent une installation courante de Pharo; voir la figure 1.1)

1. La *machine virtuelle* (abrégée en VM pour *virtual machine*) est la seule partie de l'environnement qui est particulière à chaque **couple système d'exploitation et processeur**. Des machines virtuelles pré-compilées sont disponibles pour la plupart des systèmes (Linux, Mac OS X, Win32). Dans la figure 1.1, **vous pouvez voir que la machine virtuelle pour la plateforme choisie est appelée *Pharo.exe*.**
2. Le fichier source contient le code source du système Pharo. Ce fichier ne change pas très fréquemment. Dans la figure 1.1, il correspond au fichier *SqueakV39.sources*¹.

1. Pharo est dérivé de Squeak 3.9 et partage actuellement la machine virtuelle avec Squeak.

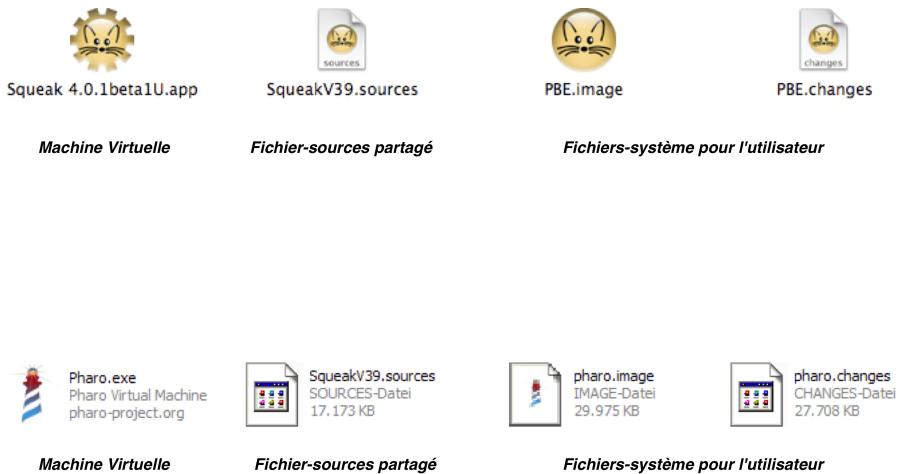


FIGURE 1.1 – Les fichiers à télécharger de Pharo **pour une des plateformes supportées**.

3. L'*image système* est un cliché d'un système Pharo en fonctionnement, figé à un instant donné. Il est composé de deux fichiers : le premier nommé avec l'extension *.image* contient l'état de tous les objets du système dont les classes et les méthodes (qui sont aussi des objets). Le second avec l'extension *.changes* contient le journal de toutes les modifications apportées au code source du système (contenu dans le fichier source). **Dans la figure 1.1, ces fichiers sont appelés *pharo.image* et *pharo.changes*.**

Téléchargez et installez Pharo sur votre ordinateur.

Nous vous recommandons d'utiliser l'image fournie sur la page web du livre².

Sachez que si vous avez déjà une autre version de Pharo qui fonctionne sur votre machine, la plupart des exemples d'introduction de ce livre fonctionneront. Il n'est donc pas nécessaire de mettre à jour Pharo. Dès lors, ne soyez pas surpris de constater parfois des différences dans l'apparence ou le comportement que nous décrirons.

Pendant que vous travaillez avec Pharo les fichiers *.image* et *.changes* sont modifiés, vous devez vous assurer qu'ils sont accessibles en écriture. Conservez toujours ces deux fichiers ensemble, *c-à-d.* dans le même dossier. Et surtout, ne tentez pas de les modifier avec un éditeur de texte, Pharo les utilise pour stocker vos objets de travail et vos changements dans le code

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

source. Faire une copie de sauvegarde de vos images téléchargées et de vos fichiers *changes* est une bonne idée ; vous pourrez ainsi toujours démarrer à partir d'une image propre et y recharger votre code.

Les fichiers *sources* et l'exécutable de la VM peuvent être en lecture seule—il est donc possible de les partager entre plusieurs utilisateurs. Ces quatre fichiers peuvent résider dans le même dossier, mais vous pouvez également placer la machine virtuelle et les fichiers sources dans un dossier partagé distinct. Vous pouvez adapter l'installation de Pharo à vos habitudes de travail et à votre système d'exploitation.



FIGURE 1.2 – Une image <http://PharoByExample.org/fr> fraîchement démarrée.

Lancement. Pour lancer Pharo, selon votre système : glissez le fichier *.image* sur l'icône de l'exécutable de la machine virtuelle, ou double-cliquez sur le fichier *.image*, ou encore, depuis une ligne de commande, tapez le nom du fichier binaire correspondant à la machine virtuelle suivi du chemin d'accès au fichier *.image* (si vous avez installé plusieurs machines virtuelles, le système ne choisira pas forcément celle qui convient, il sera préférable de glisser-déposer l'image sur la VM ou d'utiliser la ligne de commande).

Une fois lancé, Pharo vous présente une large fenêtre qui contient des espaces de travail nommés *Workspace* (voir la figure 1.2). **Vous pourriez remarquer un barre de menus mais Pharo emploie principalement des menus contextuels.**

❶ Lancez Pharo. Vous pouvez fermer les fenêtres déjà ouvertes en cli-

quant sur la bulle rouge dans le coin supérieur gauche des fenêtres.

Vous pouvez minimiser les fenêtres (ce qui les masque dans la barre de tâches située dans le bas de l'écran) en cliquant sur la bulle orange. Cliquer sur la bulle verte entraîne l'agrandissement maximal de la fenêtre.

Première interaction. Les options du menu World ("Monde" en anglais) présentées dans la figure 1.3 (a) sont un bon point de départ.

❶ Cliquez à l'aide de la souris dans l'arrière plan de la fenêtre principale pour afficher le menu World, puis sélectionnez **Workspace** pour créer un nouvel espace de travail ou **Workspace**.

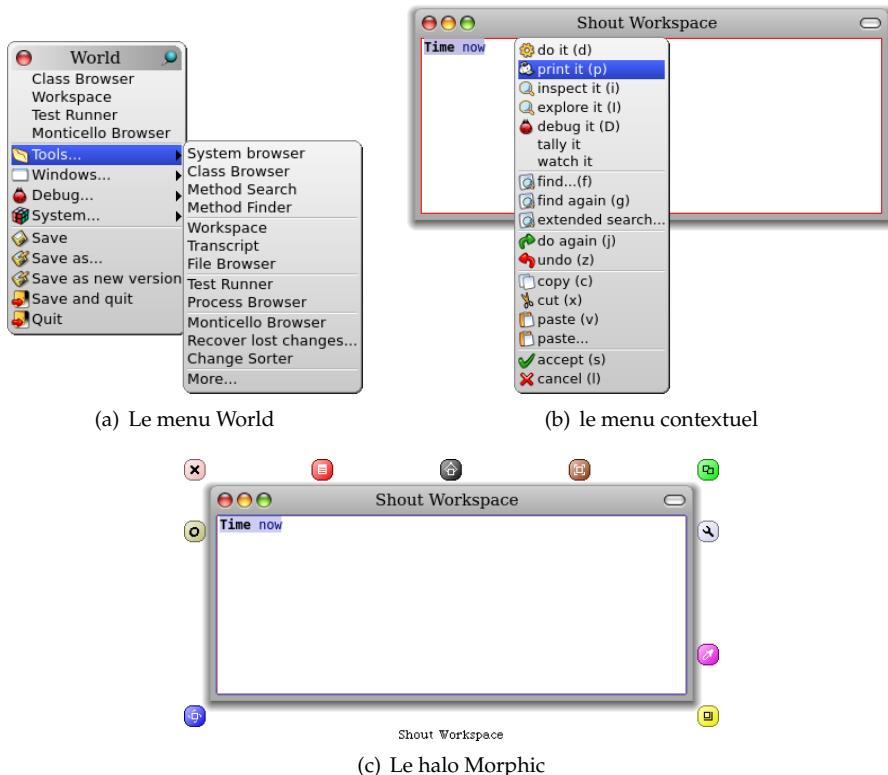


FIGURE 1.3 – Le menu World (affiché en cliquant avec la souris), un menu contextuel (affiché en cliquant avec le bouton d'action) et un halo Morphic (affiché en meta-cliquant).

Smalltalk a été conçu à l'origine pour être utilisé avec une souris à trois boutons. Si votre souris en a moins, vous pourrez utiliser des touches du cl-

vier en complément de la souris pour simuler les boutons manquants. Une souris à deux boutons fonctionne bien avec Pharo, mais si la vôtre n'a qu'un seul bouton vous devriez songer à adopter un modèle récent avec une molette qui fera office de troisième bouton : votre travail avec Pharo n'en sera que plus agréable.

Pharo évite les termes "clic gauche" ou "clic droit" car leurs effets peuvent varier selon les systèmes, le matériel ou les réglages utilisateur. Originellement, Smalltalk introduit des couleurs pour définir les différents boutons de souris³. Puisque de nombreux utilisateurs utiliseront diverses touches de modifications (*Ctrl*, *Alt*, *Meta* etc) pour réaliser les mêmes actions, nous utiliserons plutôt les termes suivants :

clic : il s'agit du bouton de la souris le plus fréquemment utilisé et correspond au fait de cliquer avec une souris à un seul bouton sans aucun touche de modifications ; cliquer sur l'arrière-plan de l'image fait apparaître le menu "World" (voir la figure 1.3(a)) ; nous utiliserons le terme *cliquer* pour définir cette action ;

clic d'action : c'est le second bouton le plus utilisé ; il est utilisé pour afficher un menu contextuel c-à-d. un menu qui fournit différentes actions dépendant de où se trouve la souris comme le montre la figure 1.3(b). Si vous n'avez pas de souris à multiples boutons, vous configurerez normalement la touche de modifications *Ctrl* pour effectuer cette même action avec votre unique bouton de souris ; nous utiliserons l'expression "*cliquer avec le bouton d'action*"⁴.

meta-clic : vous pouvez finalement *meta-cliquer* sur un objet affiché dans l'image pour activer le "halo Morphic" qui est une constellation d'icônes autour de l'objet actif à l'écran ; chaque icône représentant une poignée de contrôle permettant des actions telles que *changer la taille* ou *faire pivoter l'objet*, comme vous pouvez le voir sur la figure 1.3(c)⁵. En survolant lentement une icône avec le pointeur de votre souris, une bulle d'aide en affichera un descriptif de sa fonction. Dans Pharo, meta-cliquer dépend de votre système d'exploitation : Soit vous devez maintenir *SHIFT Ctrl* ou *SHIFT Option* tout en cliquant.

 *Saisissez Time now (expression renvoyant l'heure actuelle) dans le Work-space. Puis cliquez avec le bouton d'action dans le Workspace et sélectionnez print it (en français, "imprimez-le") dans le menu qui apparaît.*

Nous recommandons aux droitiers de configurer leur souris pour cliquer avec le bouton gauche (qui devient donc le bouton de clic), cliquer avec le

3. Les couleurs de boutons sont *rouge*, *jaune* et *bleu*. Les auteurs de ce livre n'ont jamais pu se souvenir à quelle couleur se réfère chaque bouton.

4. En anglais, le terme utilisé est "to actclick".

5. Notez que les icônes Morphic sont inactives par défaut dans Pharo, mais vous pouvez les activer via le Preferences Browser que nous verrons plus loin.

bouton d'action avec le bouton droit et meta-cliquer avec la molette de défilement cliquable, si elle est disponible. Si vous utilisez un Macintosh avec une souris à un bouton, vous pouvez simuler le second bouton en maintenant la touche **⌘** enfoncee en cliquant. Cependant, si vous prévoyez d'utiliser Pharo souvent, nous vous recommandons d'investir dans un modèle à deux boutons au minimum.

Vous pouvez configurer votre souris selon vos souhaits en utilisant les préférences de votre système ou le pilote de votre dispositif de pointage. Pharo vous propose des réglages pour adapter votre souris et les touches spéciales de votre clavier. Dans l'outil de réglage des préférences nommé Preference Browser (System > Preferences ... > Preference Browser... dans le menu World), la catégorie **keyboard** contient une option **swapControlAndAltKeys** permettant de **permuter les fonctions "cliquer avec le bouton d'action" et "meta-cliquer"**. Cette catégorie propose aussi des options afin de dupliquer les touches de modifications.

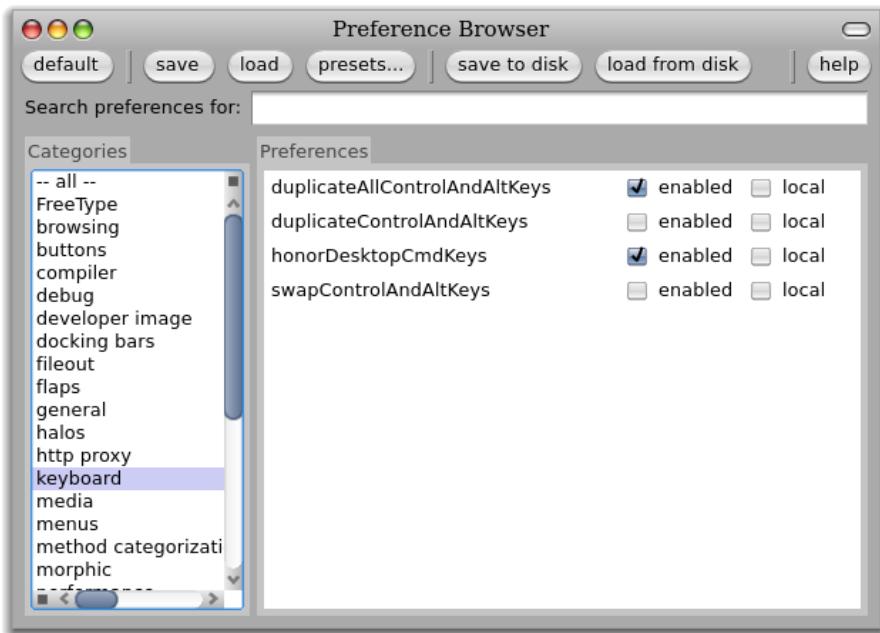


FIGURE 1.4 – Le Preference Browser.

1.2 Le menu World

 Cliquez dans l'arrière plan de Pharo.

Le menu `World` apparaît à nouveau. La plupart des menus de Pharo ne sont pas modaux ; ils ne bloquent pas le système dans l'attente d'une réponse. Avec Pharo vous pouvez maintenir ces menus sur l'écran en cliquant sur l'icône en forme d'épingle au coin supérieur droit. Essayez !

Le menu `World` vous offre un moyen simple d'accéder à la plupart des outils disponibles dans Pharo.

❶ Étudiez attentivement *le menu `World` et, en particulier, son sous-menu `Tools`* (voir la figure 1.3(a)).

Vous y trouverez une liste des principaux outils de Pharo. Nous aurons affaire à eux dans les prochains chapitres.

1.3 Envoyer des messages

❷ Ouvrez un espace de travail `Workspace` et saisissez-y le texte suivant :

```
BouncingAtomsMorph new openInWorld
```

❸ Maintenant cliquez avec le bouton d'action. Un menu devrait apparaître. Sélectionnez l'option `do it (d)` (en français, “faîtes-le !”) comme le montre la figure 1.5.

Une fenêtre contenant un grand nombre d'atomes rebondissants (en anglais, “*bouncing atoms*”) s'ouvre dans le coin supérieur gauche de votre image Pharo.

Vous venez tout simplement d'évaluer votre première expression Smalltalk. Vous avez juste envoyé le message `new` à la classe `BouncingAtomsMorph` ce qui résulte de la création d'une nouvelle instance qui à son tour reçoit le message `openInWorld`. La classe `BouncingAtomsMorph` a décidé de ce qu'il fallait faire avec le message `new` : elle recherche dans ses méthodes pour répondre de façon appropriée au message `new` (c-à-d. “nouveau” en français ; ce que nous traduirons par nouvelle instance). De même, l'instance `BouncingAtomsMorph` recherchera dans ses méthodes comment répondre à `openInWorld`.

Si vous discutez avec des habitués de Smalltalk, vous constaterez rapidement qu'ils n'emploient généralement pas les expressions comme “faire appel à une opération” ou “invoquer une méthode” : ils diront “envoyer un message”. Ceci reflète l'idée que les objets sont responsables de leurs propres actions. Vous ne direz jamais à un objet quoi faire — vous lui demanderez poliment de faire quelque chose en lui envoyant un message. C'est l'objet, et non pas vous, qui choisit la méthode appropriée pour répondre à votre message.

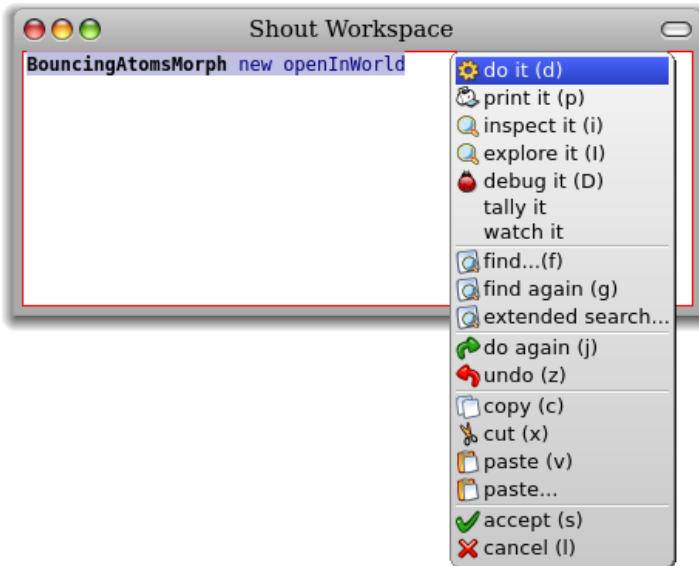


FIGURE 1.5 – Évaluer une expression avec `do it.`

1.4 Enregistrer, quitter et redémarrer une session Pharo.

Cliquez sur la fenêtre de démo des atomes rebondissants et déplacez-la où vous voulez. Vous avons maintenant la démo “dans la main”. Posez-la en cliquant.

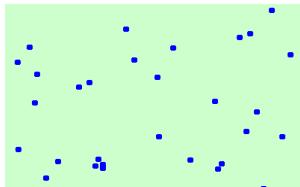


FIGURE 1.6 – Une instance de `BouncingAtomsMorph`.

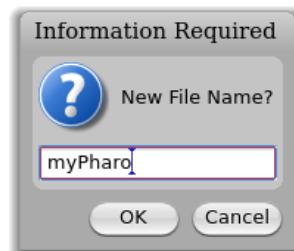


FIGURE 1.7 – La boîte de dialogue `save as...`.

Sélectionnez `World>Save as...` et entrez le nom “myPharo”, puis cliquez sur le bouton `OK`. pour sauvegarder sous un nouveau nom d’image.

Pour quitter, sélectionnez World> Save and quit.

Le dossier qui contenait les fichiers *image* et *changes* lorsque vous avez lancé cette session de travail avec Pharo contient désormais deux nouveaux fichiers : “*myPharo.image*” et “*myPharo.changes*”. Ils représentent l’image “vivante” de votre session Pharo au moment qui précédait votre enregistrement avec *Save and quit*. Ces deux fichiers peuvent être copiés à votre convenance dans les dossiers de votre disque pour y être utilisés plus tard. à vous de les invoquer en prenant soin (selon votre système de fichiers) de déplacer, copier ou lier le fichier *sources* correspondant, tout en veillant à exécuter la bonne machine virtuelle.

❶ Lancez Pharo avec l’image que vous venez de créer c-à-d. le fichier “*myPharo.image*”.

Vous retrouvez l’état de votre session exactement tel qu’il était avant que vous quittiez Pharo. La démo des atomes rebondissants est toujours sur votre fenêtre de travail et les atomes continuent de rebondir depuis la position qu’ils avaient lorsque vous avez quitté.

En lançant pour la première fois Pharo, la machine virtuelle charge le fichier *image* que vous spécifiez. Ce fichier contient l’instantané d’un grand nombre d’objets et surtout le code pré-existant accompagné des outils de développement qui sont d’ailleurs des objets comme les autres. En travaillant dans Pharo, vous allez envoyer des messages à ces objets, en créer de nouveaux, et certains seront supprimés et l’espace-mémoire utilisé sera récupéré (c-à-d. passé au ramasse-miettes ou *garbage collector*).

En quittant Pharo vous sauvegardez un instantané de tous vos objets. En sauvegardant (par “*Save*”), vous remplacerez l’image courante par l’instantané de votre session. Pour préserver l’image courante, vous devez enregistrer sous un nouveau nom comme nous venons de le faire.

Chaque fichier *.image* est accompagné d’un fichier *.changes*. Ce fichier contient un journal de toutes les modifications que vous avez faites en utilisant l’environnement de développement. Vous n’avez pas à vous soucier de ce fichier la plupart du temps. Mais comme nous allons le voir plus tard, le fichier *.changes* pourra être utilisé pour rétablir votre système Pharo à la suite d’erreurs.

L’image sur laquelle vous travaillez provient d’une image de Smalltalk-80 créée à la fin des années 1970. Beaucoup des objets qu’elle contient sont là depuis des décennies !

Vous pourriez penser que l’utilisation d’une image est incontournable pour stocker et gérer des projets, mais comme nous le verrons bientôt il existe des outils plus adaptés pour gérer le code et travailler en équipe sur des projets. Les images sont très utiles mais nous les considérons comme une pratique un peu dépassée et fragile pour diffuser et partager vos projets alors

qu'il existe des outils tels que Monticello qui proposent de biens meilleurs moyens de suivre les évolutions du code et de le partager entre plusieurs développeurs.

❶ *Meta-cliquez (en utilisant les touches de modifications appropriées conjointement avec votre souris) sur la fenêtre d'atomes rebondissants⁶.*

Vous verrez tout autour une collection d'icônes **circulaires** colorées nommée halo de BouncingAtomsMorph ; l'icône halo est aussi appelée *poignée*. Cliquez sur la poignée rose pâle qui contient une croix ; la fenêtre de démo disparaît.

1.5 Les fenêtres Workspace et Transcript

❷ *Fermez toutes fenêtres actuellement ouvertes. Ouvrez un Transcript (via le menu World > Tools) et un Workspace. Positionnez et redimensionnez le Transcript et le Workspace pour ce dernier recouvre le Transcript.*

Vous pouvez redimensionner les fenêtres en glissant l'un de leurs coins ou **en meta-cliquant pour afficher le halo Morphic** : utilisez alors l'icône jaune située en bas à droite.

Une seule fenêtre est active à la fois ; elle s'affiche au premier plan et **son contour est alors mis en relief**.

Le Transcript est un objet qui est couramment utilisé pour afficher des messages du système. C'est un genre de "console".

Les fenêtres Workspace (ou espace de travail) sont destinées à y saisir vos expressions de code Smalltalk à expérimenter. Vous pouvez aussi les utiliser simplement pour taper une quelconque note de texte à retenir, comme une liste de choses à faire (en anglais, *todo-list*) ou des instructions pour quiconque est amené à utiliser votre image. Les Workspaces sont souvent employés pour maintenir une documentation à propos de l'image courante, comme c'est le cas dans l'image standard précédemment chargée (voir la figure 1.2).

❸ *Saisissez le texte suivant dans l'espace de travail Workspace :*

Transcript show: 'hello world'; cr.

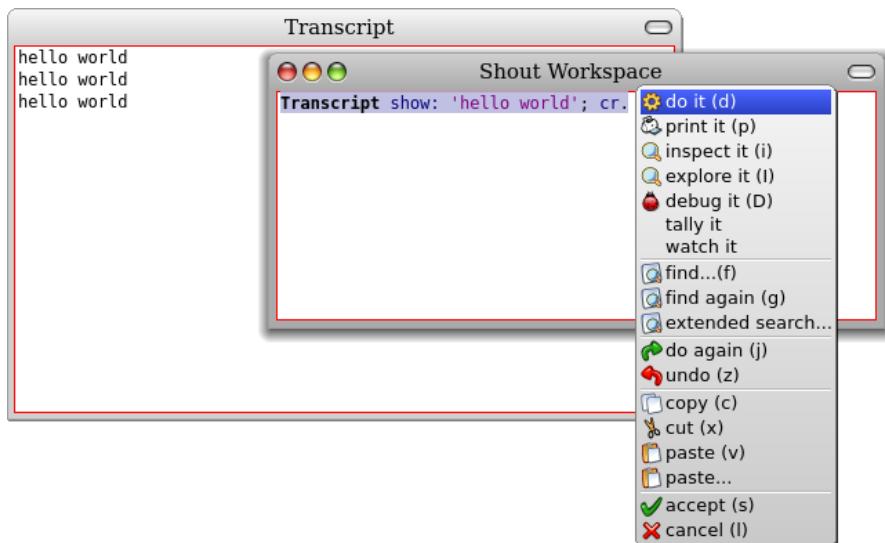
Expérimitez la sélection en double-cliquant dans l'espace de travail à différents points dans le texte que vous venez de saisir. Remarquez comment

6. Souvenez-vous que vous pourriez avoir besoin d'activer l'option `halosEnabled` dans le Preference Browser.

un mot entier ou tout un texte est sélectionné selon que vous cliquez sur un mot, à la fin d'une chaîne de caractères ou à la fin d'une expression entière.

 Sélectionnez le texte que vous avez saisi puis cliquez avec le bouton d'action. Choisissez `do it (d)` (dans le sens “faites-le !”, c-à-d. évaluer le code sélectionné) dans le menu contextuel.

Notez que le texte “hello world”⁷ apparaît dans la fenêtre Transcript (voir la figure 1.5). Refaites encore un `do it (d)` (Le `(d)` dans l'option de menu `do it (d)` vous indique que le raccourci-clavier correspondant est CMD-d. Pour plus d'informations, rendez-vous dans la prochaine section !).



Les fenêtres sont superposées. Le Workspace est actif.

1.6 Les raccourcis-clavier

Si vous voulez évaluer une expression, vous n'avez pas besoin de toujours passer par le menu accessible en cliquant avec le bouton d'action : les raccourcis-clavier sont là pour vous. Ils sont mentionnés dans les expressions parenthésées des options des menus. Selon votre plateforme, vous pouvez être amené à presser l'une des touches de modifications soit `Control`, `Alt`, `Command` ou `Meta` (nous les indiquerons de manière générique par `CMD-touche`).  Réévaluez l'expression dans le Workspace en utilisant

7. NdT : C'est une tradition de la programmation : tout premier programme dans un nouveau langage de programmation consiste à afficher la phrase en anglais “hello world” signifiant “bonjour le monde”.

cette fois-ci le raccourci-clavier : CMD-d.

En plus de do it, vous aurez noté la présence de print it (pour évaluer et afficher le résultat dans le même espace de travail), de inspect it (pour inspecter) et de explore it (pour explorer). Jetons un coup d'œil à ceux-ci.

 Entrez l'expression $3 + 4$ dans le Workspace. Maintenant évaluez en faisant un do it avec le raccourci-clavier.

Ne soyez pas surpris que rien ne se passe ! Ce que vous venez de faire, c'est d'envoyer le message + avec l'argument 4 au nombre 3. Le résultat 7 aura normalement été calculé et retourné, mais puisque votre espace de travail Workspace ne savait que faire de ce résultat, la réponse a simplement été jetée dans le vide. Si vous voulez voir le résultat, vous devriez faire print it au lieu de do it. En fait, print it compile l'expression, l'exécute et envoie le message printString au résultat puis affiche la chaîne de caractère résultante.

 Sélectionnez $3+4$ et faites print it (CMD-p).

Cette fois, nous pouvons lire le résultat que nous attendions (voir la figure 1.8).

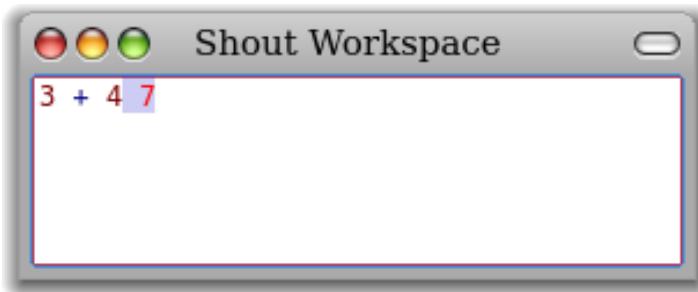


FIGURE 1.8 – Afficher le résultat sous forme de chaîne de caractères avec print it plutôt que de simplement évaluer avec do it.

$3 + 4 \longrightarrow 7$

Nous utilisons la notation \longrightarrow comme convention dans tout le livre pour indiquer qu'une expression particulière donne un certain résultat quand vous l'évaluez avec print it.

 Effacez le texte souligné "7"; comme Pharo devrait l'avoir sélectionné pour vous, vous n'avez qu'à presser sur la touche de suppression (suivant votre type de clavier Suppr. ou Del.). Sélectionnez $3+4$ à nouveau et, cette fois, faites une inspection avec inspect it (CMD-i).

Vous devriez maintenant voir une nouvelle fenêtre appelée *inspecteur* avec pour titre `SmallInteger: 7` (voir la figure 1.9). L'inspecteur ou (sous son nom de classe) Inspector est un outil extrêmement utile : il vous permet de naviguer et d'interagir avec n'importe quel objet du système. Le titre nous dit que 7 est une instance de la classe `SmallInteger` (classe des entiers sur 31 bits). Le panneau de gauche nous offre une vue des variables d'instance de l'objet en cours d'inspection. Nous pouvons naviguer entre ces variables et le panneau de droite nous affiche leur valeur. Le panneau inférieur peut être utilisé pour écrire des expressions envoyant des messages à l'objet.

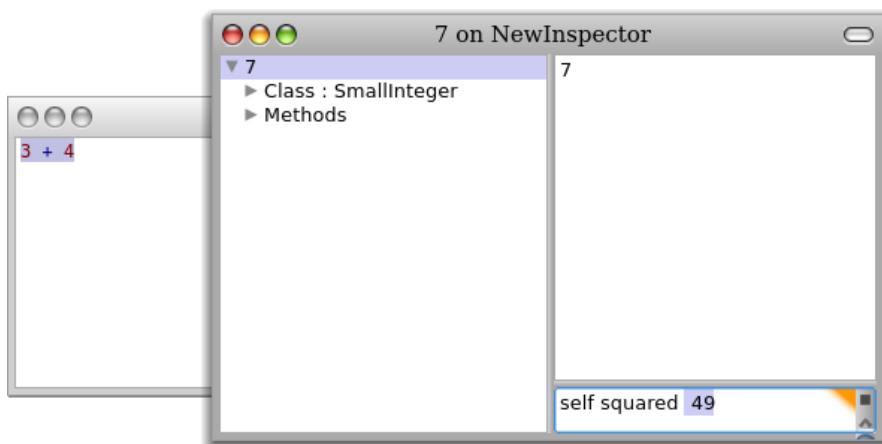


FIGURE 1.9 – Inspecter un objet.

Saisissez `self squared` dans le panneau inférieur de l'inspecteur que vous avez ouvert sur l'entier 7 et faites un `print it`. Le message `squared` (carré) va éléver le nombre 7 lui-même (`self`).

Fermez l'inspecteur. Saisissez dans un Workspace le mot-expression `Object` et explorez-le via `explore it` (CMD-I, i majuscule).

Vous devriez voir maintenant une fenêtre intitulée `Object` contenant le texte `> root: Object`. Cliquez sur le triangle pour l'ouvrir (voir la figure 1.10).

Cet explorateur (ou Explorer) est similaire à l'inspecteur mais il offre une vue arborescente d'un objet complexe. Dans notre cas, l'objet que nous observons est la classe `Object`. Nous pouvons voir directement toutes les informations stockées dans cette classe et naviguer facilement dans toutes ses parties.

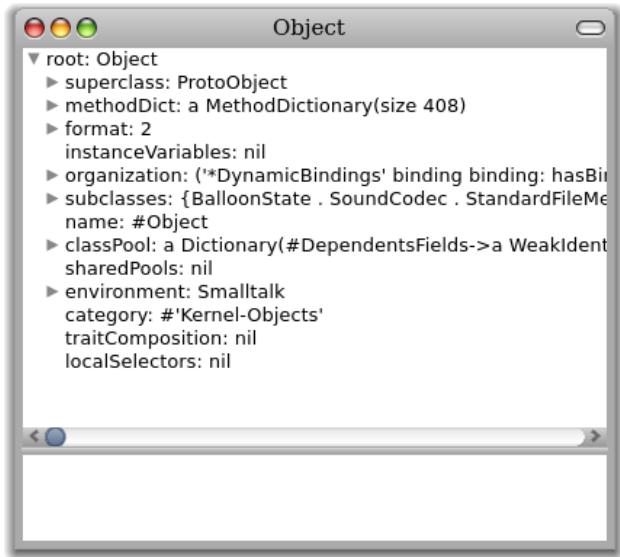


FIGURE 1.10 – Explorer Object.

1.7 Le navigateur de classes Class Browser

Le navigateur de classes nommé Class Browser⁸ est un des outils-clé pour programmer. Comme nous le verrons bientôt, il y a plusieurs navigateurs ou *browsers* intéressants disponibles pour Pharo, mais c'est le plus simple que vous pourrez trouver dans n'importe quelle image, que nous allons utiliser ici.

Ouvrez un navigateur de classes en sélectionnant World > Class Browser⁹.

Nous pouvons voir un navigateur de classes sur la figure 1.11. La barre de titre indique que nous sommes en train de parcourir la classe Object.

à l'ouverture du Browser, tous les panneaux sont vides excepté le premier à gauche. Ce premier panneau liste tous les *paquetages* (en anglais, *packages*) connus ; ils contiennent des groupes de *classes parentes*.

cliquez sur le paquetage Kernel .

8. Ce navigateur est confusément référencé sous les noms “System Browser” ou “Code Browser”. Pharo utilise l’implémentation OmniBrowser du navigateur connue aussi comme “OB” ou “Package Browser”. Dans ce livre, nous utiliserons simplement le terme de Browser ou, s’il y a ambiguïté, nous parlerons de navigateur de classes.

9. Si votre Browser ne ressemble pas à celui visible sur la figure 1.11, vous pourriez avoir besoin de changer le navigateur par défaut. Veuillez la FAQ ??, p. ??

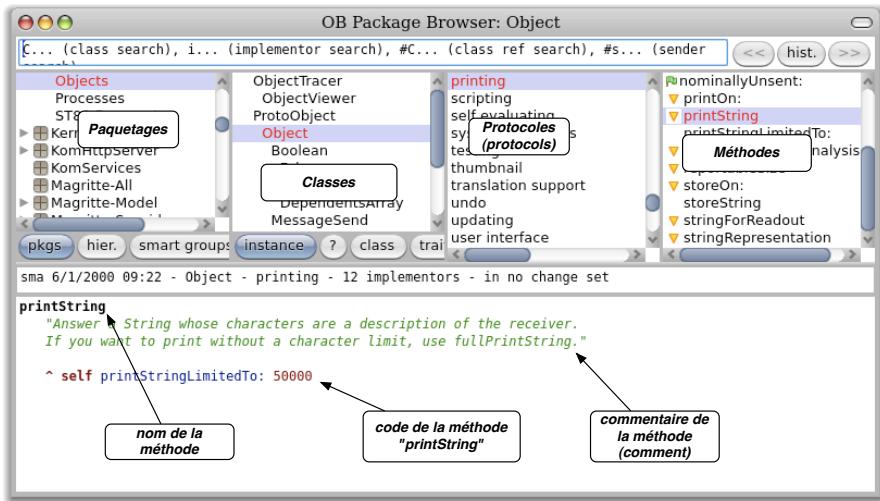


FIGURE 1.11 – Le navigateur de classes (ou Browser) affichant la méthode printString de la classe Object.

Cette manipulation permet l'affichage dans le second panneau de toutes les classes du paquetage sélectionné.

➊ Sélectionnez la classe Object.

Désormais les deux panneaux restants se remplissent. Le troisième panneau affiche les protocoles de la classe sélectionnée. Ce sont des regroupements commodes pour relier des méthodes connexes. Si aucun protocole n'est sélectionné, vous devriez voir toutes les méthodes disponibles de la classe dans le quatrième panneau.

➋ Sélectionnez le protocole printing , protocole de l'affichage.

Vous pourriez avoir besoin de faire défiler (avec la barre de défilement) la liste des protocoles pour le trouver. Vous ne voyez maintenant que les méthodes relatives à l'affichage.

➌ Sélectionnez la méthode printString.

Dès lors, vous voyez dans la partie inférieure du Browser le code source de la méthode printString partagé par tous les objets (tous dérivés de la classe Object, exception faite de ceux qui la surchargent).

1.8 Trouver des classes

Il existe plusieurs moyens de trouver une classe dans Pharo. Tout d'abord, comme nous l'avons vu plus haut, nous pouvons savoir (ou deviner) dans quelle catégorie elle se trouve et, de là, naviguer jusqu'à elle via le navigateur de classes.

Une seconde technique consiste à envoyer le message `browse` (ce mot a le sens de "naviguer") à la classe, ce qui a pour effet d'ouvrir un navigateur de classes sur celle-ci (si elle existe bien sûr). Supposons que nous voulions naviguer dans la classe `Boolean` (la classe des booléens).

Saisissez Boolean browse dans un Workspace et faites un do it.

Un navigateur s'ouvrira sur la classe `Boolean` (voir la figure 1.12). Il existe aussi un raccourci-clavier `CMD-b` (`browse`) que vous pouvez utiliser dans n'importe quel outil où vous trouvez un nom de classe ; sélectionnez le nom de la classe (par ex., `Boolean`) puis tapez `CMD-b`.

Utilisez le raccourci-clavier pour naviguer dans la classe Boolean.

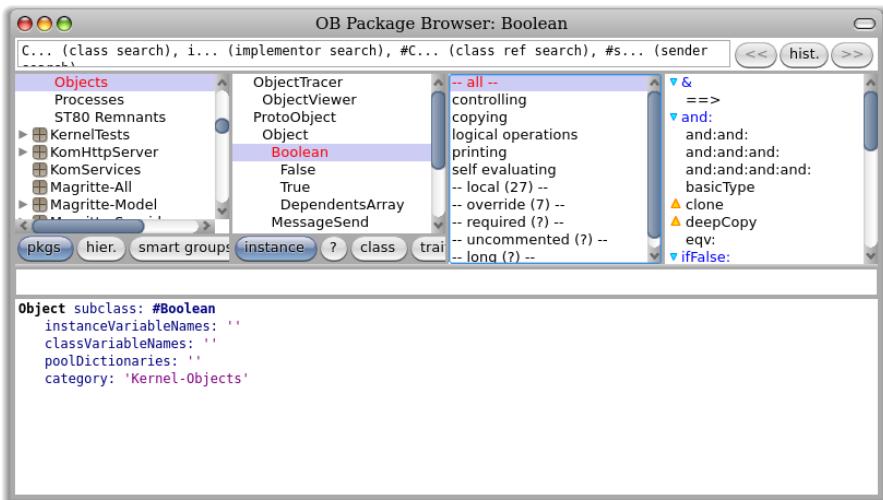


FIGURE 1.12 – Le navigateur de classes affichant la définition de la classe `Boolean`.

Remarquez que nous voyons une *définition de classe* quand la classe `Boolean` est sélectionnée mais sans qu'aucun protocole ni aucune méthode ne le soit (voir la figure 1.12). Ce n'est rien de plus qu'un message Smalltalk ordinaire qui est envoyé à la classe parente lui réclamant de créer une sous-classe. Ici nous voyons qu'il est demandé à la classe `Object` de créer une

sous-classe nommée Boolean sans aucune variables d'instance, ni variables de classe ou "pool dictionaries" et de mettre la classe Boolean dans la catégorie *Kernel-Objects*. Si vous cliquez sur le bouton **?** en bas du panneau de classes, vous verrez le commentaire de classe dans un panneau dédié comme le montre la figure 1.13.

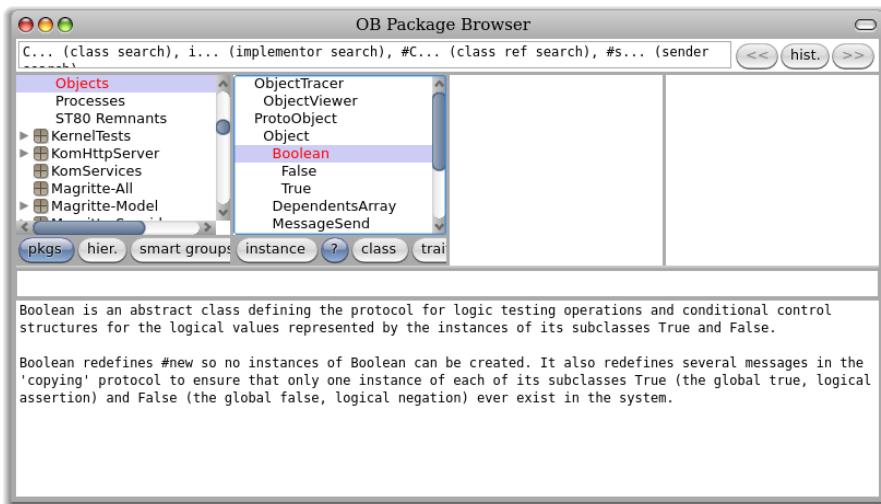


FIGURE 1.13 – Le commentaire de classe de Boolean.

Souvent, la méthode la plus rapide de trouver une classe consiste à la rechercher par son nom. Par exemple, supposons que vous êtes à la recherche d'une classe inconnue qui représente les jours et les heures.

💡 Placez la souris dans le panneau des paquetages du Browser et tapez CMD-f ou sélectionnez find class... (f) dans le menu contextuel accessible en cliquant avec le bouton d'action. Saisissez "time" (c-à-d. le temps, puisque c'est l'objet de notre quête) dans la boîte de dialogue et acceptez cette entrée.

Une liste de classes dont le nom contient "time" vous sera présentée (voir la figure 1.14). Choisissez-en une, disons, Time ; un navigateur l'affichera avec un commentaire de classe suggérant d'autres classes pouvant être utiles. Si vous voulez naviguer dans l'une des autres classes, sélectionnez son nom (dans n'importe quelle zone de texte) et tapez CMD-b.

Notez que si vous tapez le nom complet (et correctement capitalisé *c-à-d.* en respectant la casse) de la classe dans la boîte de dialogue de recherche (find), le navigateur ira directement à cette classe sans montrer aucune liste de classes à choisir.

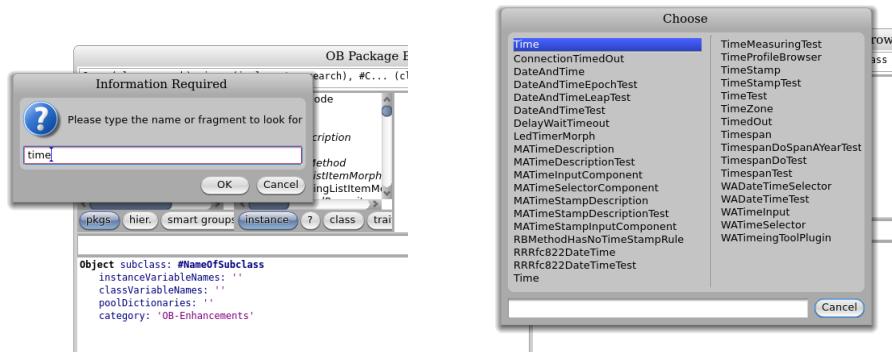


FIGURE 1.14 – Rechercher une classe d'après son nom.

1.9 Trouver des méthodes

Vous pouvez parfois deviner le nom de la méthode, ou tout au moins une partie de son nom plus facilement que le nom d'une classe. Par exemple, si vous êtes intéressé par la connaissance du temps actuel, vous pouvez vous attendre à ce qu'il y ait une méthode affichant le temps *maintenant* : comme la langue de Smalltalk est l'anglais et que *maintenant* se dit "now", une méthode contenant le mot "now" a de forte chance d'exister. Mais où pourrait-elle être ? L'outil *Method Finder* peut vous aider à la trouver.

Sélectionnez World > Tools > Method Finder.

Saisissez "now" dans le panneau supérieur gauche et cliquez sur accept (ou tapez simplement la touche ENTRÉE). Le chercheur de méthodes Method Finder affichera une liste de tous les noms de méthodes contenant la sous-chaine de caractères "now".

Pour défiler jusqu'à now lui-même, tapez "n" ; cette astuce fonctionne sur toutes les zones à défilement de n'importe quelle fenêtre. En sélectionnant "now", le panneau de droite vous présentera les classes qui définissent une méthode avec ce nom, comme le montre la figure 1.15. Sélectionner une de ces classes vous ouvrira un navigateur sur celle-ci.

à d'autres moments, vous pourriez avoir en tête qu'une méthode existe bien sans savoir comment elle s'appelle. Le Method Finder peut encore vous aider ! Par exemple, partons de la situation suivante : vous voulez trouver une méthode qui transforme une chaîne de caractères en sa version majuscule, c-à-d. qui transforme 'eureka' en 'EUREKA'.

Saisissez 'eureka' . 'EUREKA' dans le Method Finder, comme le montre la figure 1.16.

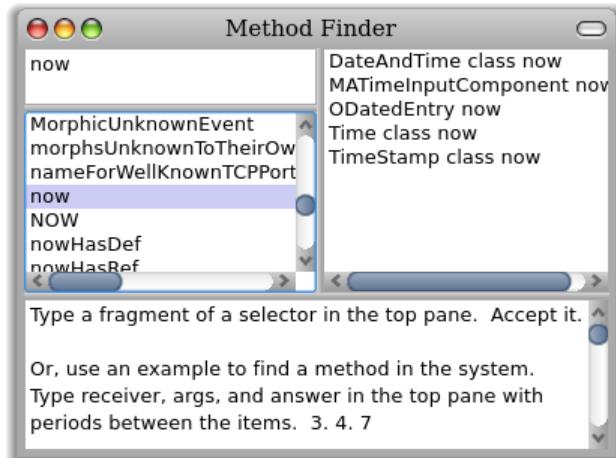


FIGURE 1.15 – Le Method Finder affichant toutes les classes qui définissent une méthode appelée now.

Le Method Finder vous suggère une méthode qui fait ce que vous voulez¹⁰.

Un astérisque au début d'une ligne dans le panneau de droite du Method Finder vous indique que cette méthode est celle qui a été effectivement utilisée pour obtenir le résultat requis. Ainsi, l'astérisque devant String asUppercase vous fait savoir que la méthode asUppercase (traduisible par “en tant que majuscule”) définie dans la classe String (la classe des chaînes de caractères) a été exécutée et a renvoyé le résultat voulu. Les méthodes qui n’ont pas d’astérisque ne sont que d’autres méthodes que celles qui retournent le résultat attendu. Character>asUppercase n’a pas été exécutée dans notre exemple, parce que ‘eureka’ n’est pas un caractère de classe Character.

Vous pouvez aussi utiliser le Method Finder pour trouver des méthodes avec plusieurs arguments ; par exemple, si vous recherchez une méthode qui trouve le plus grand commun diviseur de deux entiers, vous pouvez essayer de saisir 25. 35. 5 comme exemple. Vous pouvez aussi donner au Method Finder de multiples exemples pour restreindre le champ des recherches ; le texte d'aide situé dans le panneau inférieur vous apprendra [comment faire](#).

10. Si une fenêtre s’ouvre soudain avec un message d’alerte à propos d’une méthode obsolète — le terme anglais est *deprecated method* — ne paniquez pas : le Method Finder est simplement en train d’essayer de chercher parmi tous les candidats incluant ainsi les méthodes obsolètes. Cliquez alors sur le bouton [Proceed](#).

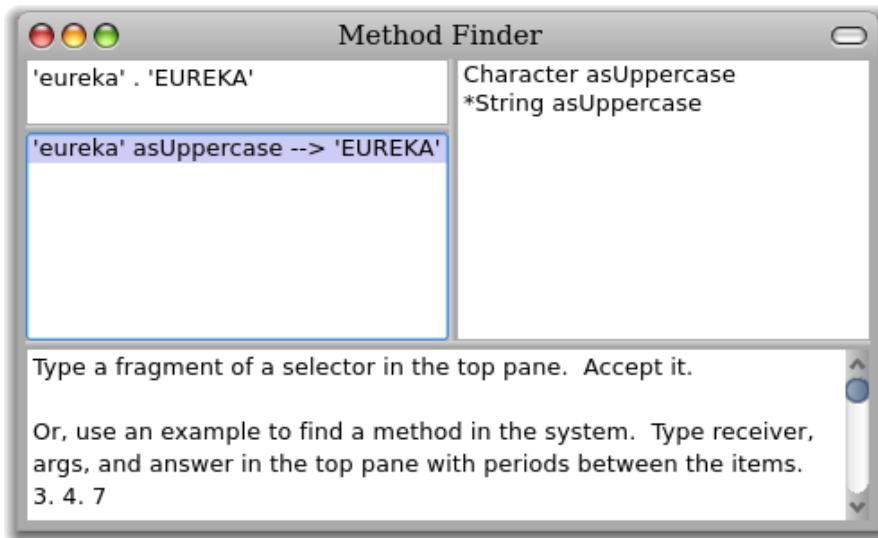


FIGURE 1.16 – Trouver une méthode par l'exemple.

1.10 Définir une nouvelle méthode

L'avènement de la méthodologie de développement orientée tests ou *Test Driven Development*¹¹ a changé la façon d'écrire du code. L'idée derrière cette technique aussi appelée TDD se résume par l'écriture du test qui définit le comportement désiré de notre code *avant* celle du code proprement dit. à partir de là seulement, nous écrivons le code qui satisfait au test.

Supposons que nous voulions écrire une méthode qui “hurle quelque chose”. Qu'est-ce que cela veut dire au juste ? Quel serait le nom le plus convenable pour une telle méthode ? Comment pourrions-nous être sûrs que les programmeurs en charge de la maintenance future du code auront une description sans ambiguïté de ce que ce code est censé faire ? Nous pouvons répondre à toutes ces questions en proposant l'exemple suivant :

Quand nous envoyons le message `shout` (qui veut dire “crier” en anglais) à la chaîne de caractères “Pas de panique”, le résultat devrait être “PAS DE PANIQUE !”.

Pour faire de cet exemple quelque chose que le système peut utiliser, nous le transformons en méthode de test :

11. Kent Beck, *Test Driven Development : By Example*. Addison-Wesley, 2003, ISBN 0-321-14653-0.

Méthode 1.1 – Un test pour la méthode shout

```
testShout
self assert: ('Pas de panique' shout = 'PAS DE PANIQUE!')
```

Comment créons-nous une nouvelle méthode dans Pharo ? Premièrement, nous devons décider quelle classe va accueillir la méthode. Dans ce cas, la méthode shout que nous testons ira dans la classe String car c'est la classe des chaînes de caractères et "Pas de panique" en est une. Donc, par convention, le test correspondant ira dans une classe nommée StringTest.

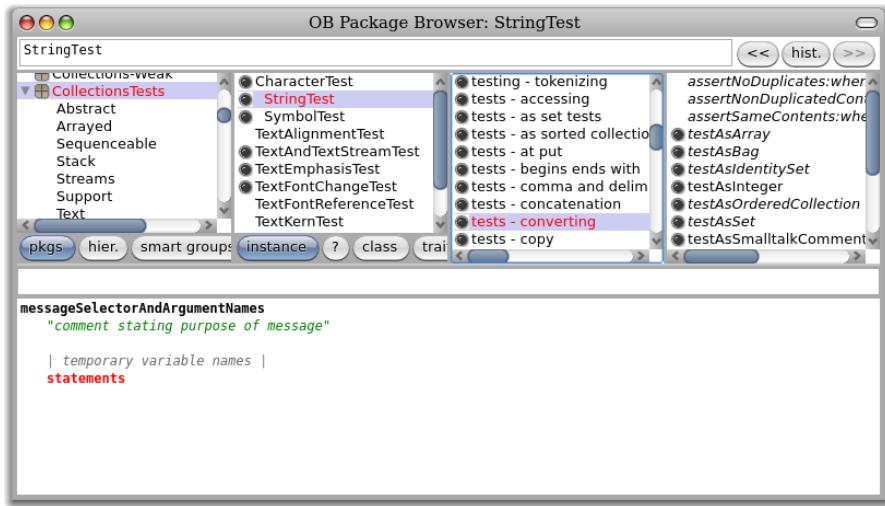


FIGURE 1.17 – Le patron de la nouvelle méthode dans la classe StringTest.

Ouvrez un navigateur de classes sur la classe StringTest. Sélectionnez un protocole approprié pour notre méthode ; dans notre cas, tests - converting (signifiant tests de conversion, puisque notre méthode modifiera le texte en retour), comme nous pouvons le voir sur la figure 1.17. Le texte surligné dans le panneau inférieur est un patron de méthode qui vous rappelle ce à quoi ressemble une méthode. Effacez-le et saisissez le code de la méthode 1.1.

Une fois que vous avez commencé à entrer le texte dans le navigateur, l'espace de saisie est entouré de rouge pour vous rappeler que ce panneau contient des changements non-sauvegardés. Lorsque vous avez fini de saisir le texte de la méthode de test, sélectionnez accept (s) via le menu activé en cliquant avec le bouton d'action dans ce panneau ou utilisez le raccourci-clavier CMD-s : ainsi, vous compilerez et sauvegarderez votre méthode.

Si c'est la première fois que vous acceptez du code dans votre image, vous serez invité à saisir votre nom dans une fenêtre spécifique. Beaucoup de

personnes ont contribué au code de l'image ; c'est important de garder une trace de tous ceux qui créent ou modifient les méthodes. Entrez simplement votre prénom suivi de votre nom sans espaces ni point de séparation.

Puisqu'il n'y a pas encore de méthode nommée shout, le Browser vous demandera confirmation que c'est bien le nom que vous désirez — il vous suggèrera d'ailleurs d'autres noms de méthodes existantes dans le système (voir la figure 1.19). Ce comportement du navigateur est utile si vous aviez effectivement fait une erreur de frappe. Mais ici, nous voulons vraiment écrire shout puisque c'est la méthode que nous voulons créer. Dès lors, nous n'avons qu'à confirmer cela en sélectionnant la première option parmi celles du menu, comme vous le voyez sur la figure 1.19.

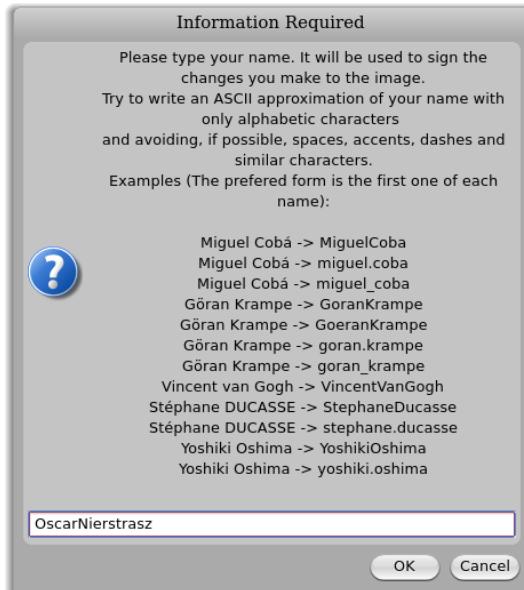


FIGURE 1.18 – Saisir son nom.

❶ Lancez votre test nouvellement créé : ouvrez le programme SUnit nommé TestRunner depuis le menu **World**.

Les deux panneaux les plus à gauche se présentent un peu comme les panneaux supérieurs du Browser. Le panneau de gauche contient une liste de catégories restreintes aux catégories qui contiennent des classes de test.

❷ Sélectionnez CollectionsTests-Text et le panneau juste à droite vous affichera alors toutes les classes de test de cette catégorie dont la classe StringTest . Les classes sont déjà sélectionnées dans cette catégorie ; cliquez alors sur Run Selected pour lancer tous ces tests.

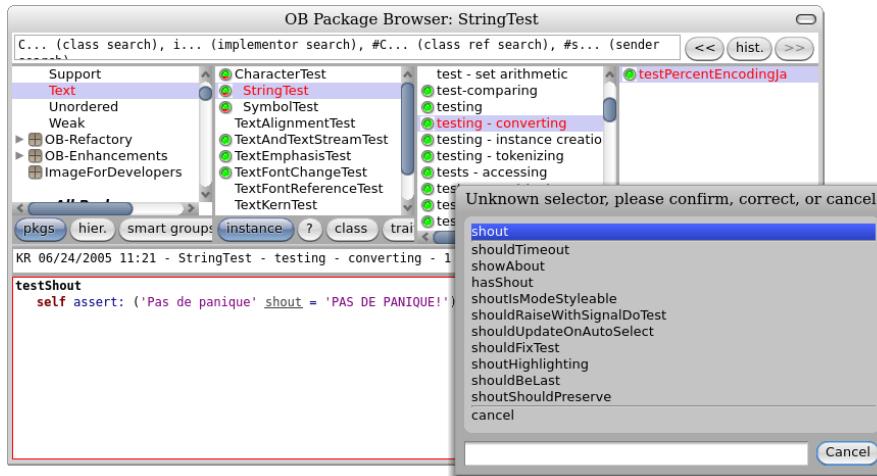


FIGURE 1.19 – Accepter la méthode testShout dans la classe StringTest.

Vous devriez voir un message comme celui de la figure 1.20, vous indiquant qu'il y a eu une erreur lors de l'exécution des tests. La liste des tests qui donne naissance à une erreur est affichée dans le panneau inférieur de droite ; comme vous pouvez le voir, c'est bien StringTest»#testShout le coupable (remarquez que la notation StringTest»#testShout est la convention Smalltalk pour identifier la méthode de la classe StringTest). Si vous cliquez sur cette ligne de texte, le test erroné sera lancé à nouveau mais, cette fois-ci, de telle façon que vous voyez l'erreur surgir : "MessageNotUnderstood: ByteString»shout".

La fenêtre qui s'ouvre avec le message d'erreur est le débogueur Smalltalk (voir la figure 1.21). Nous verrons le débogueur nommé Debugger et ses fonctionnalités dans le chapitre 6.

L'erreur était bien sûr attendue ; lancer le test génère une erreur parce que nous n'avons pas encore écrit la méthode qui dit aux chaînes de caractères comment hurler *c-à-d.* comment répondre au message shout. De toutes façons, c'est une bonne pratique de s'assurer que le test échoue ; cela confirme que nous avons correctement configuré notre machine à tests et que le nouveau test est actuellement en cours d'exécution. Une fois que vous avez vu l'erreur, vous pouvez cliquer sur le bouton **Abandon** pour abandonner le test en cours, ce qui fermera la fenêtre du débogueur. Sachez qu'en Smalltalk vous pouvez souvent définir la méthode manquante directement depuis le débogueur en utilisant le bouton **Create**, en y éditant la méthode nouvellement créée puis, *in fine*, en appuyant sur le bouton **Proceed** pour poursuivre le test.

Définissons maintenant la méthode qui fera du test un succès !

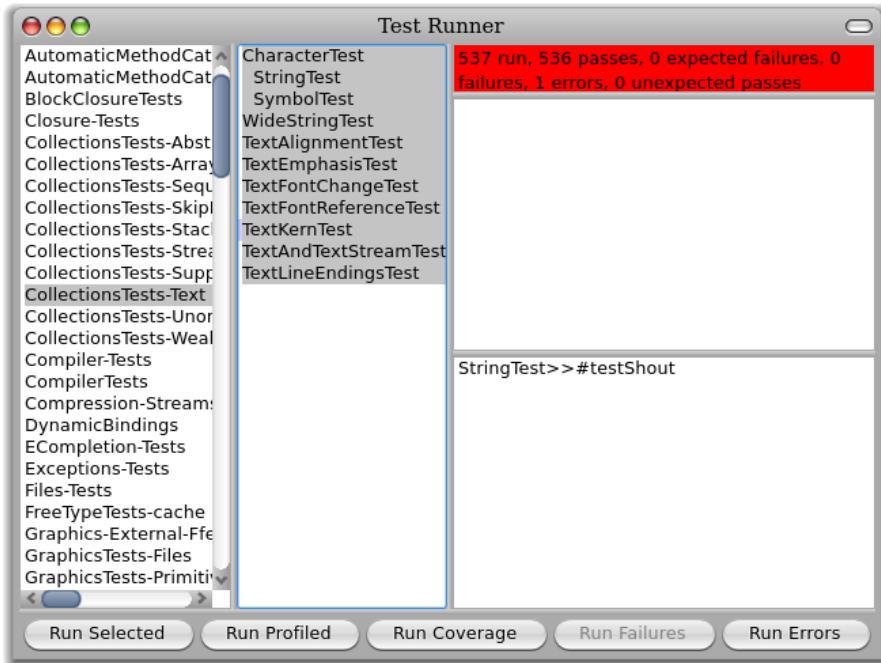


FIGURE 1.20 – Lancer les tests de String.

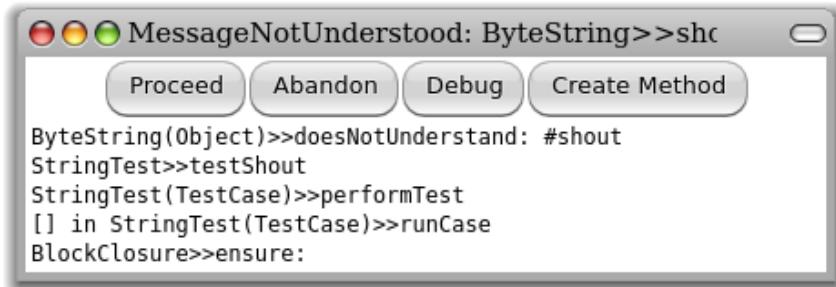


FIGURE 1.21 – La fenêtre de démarrage du débogueur.

Sélectionnez la classe String dans le Browser et rendez-vous dans le protocole déjà existant des méthodes de conversion et appellé converting. à la place du patron de création de méthode, saisissez le texte de la méthode 1.2 et faites accept (saisissez ^ pour obtenir un ↑)

Méthode 1.2 – La méthode shout

```
shout
↑ self asUppercase, '!'
```

La virgule est un opérateur de concaténation de chaînes de caractères, donc, le corps de cette méthode ajoute un point d'exclamation à la version majuscule (obtenue avec la méthode `asUppercase`) de l'objet `String` auquel le message `shout` a été envoyé. Le ↑ dit à Pharo que l'expression qui suit est la réponse que la méthode doit retourner ; dans notre cas, il s'agit de la nouvelle chaîne concaténée.

Est-ce que cette méthode fonctionne ? Lançons tout simplement notre test afin de le savoir.

 Cliquez encore sur le bouton `Run Selected` du Test Runner. Cette fois vous devriez obtenir une barre de signalisation verte (et non plus rouge) et son texte vous confirmera que tous les tests lancés se feront sans aucun échec (*ni failures, ni errors*).

Vous voyez une barre verte¹² dans le Test Runner ? Bravo ! Sauvegardez votre image et faites une pause. Vous l'avez bien mérité.

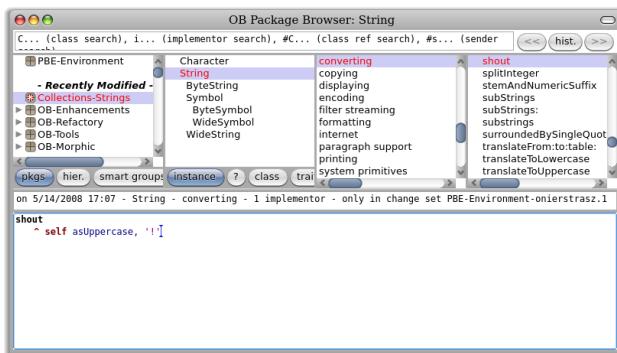


FIGURE 1.22 – La méthode `shout` dans la classe `String`.

1.11 Résumé du chapitre

Dans ce chapitre, nous vous avons introduit à l'environnement de Pharo et nous vous avons montré comment utiliser certains de ses principaux outils comme le Browser, le Method Finder et le Test Runner. Vous avez pu avoir un aperçu de la syntaxe sans que vous puissiez encore la comprendre suffisamment à ce stade.

- Un système Pharo fonctionnel comprend une *machine virtuelle* (souvent abrégée par VM), un fichier *sources* et un couple de fichiers : une *image* et un fichier *changes*. Ces deux derniers sont les seuls à être susceptibles de changer, puisqu'ils sauvegardent un cliché du système actif.
- Quand vous restaurez une image Pharo, vous vous retrouvez exactement dans le même état — avec les mêmes objets lancés — que lorsque vous l'avez laissée au moment de votre dernière sauvegarde de cette image.
- Pharo est destiné à fonctionner avec une souris à trois boutons [pour cliquer, cliquer avec le bouton d'action ou meta-cliquer](#). Si vous n'avez pas de souris à trois boutons, vous pouvez utiliser des touches de modifications au clavier pour obtenir le même effet.
- Vous cliquez sur l'arrière-plan de Pharo pour faire apparaître le *menu World* et pouvoir lancer depuis celui-ci divers outils.
- Un *Workspace* ou espace de travail est un outil destiné à écrire et évaluer des fragments de code. Vous pouvez aussi l'utiliser pour y stocker un texte quelconque.
- Vous pouvez utiliser des raccourcis-clavier sur du texte dans un *Workspace* ou tout autre outil pour en évaluer le code. Les plus importants sont `do it` (CMD-d), `print it` (CMD-p), `inspect it` (CMD-i) et `explore it` (CMD-I).
- SqueakMap est un outil pour télécharger des paquetages utiles depuis Internet.
- Le navigateur de classes *Browser* est le principal outil pour naviguer dans le code Pharo et pour développer du nouveau code.
- Le *Test Runner* permet d'effectuer des tests unitaires. Il supporte pleinement la méthodologie de programmation orientée tests connue sous le nom de *Test Driven Development*.

Chapitre 2

Une première application

Dans ce chapitre, nous allons développer un jeu simple de réflexion, le jeu Lights Out¹. En cours de route, nous allons faire la démonstration de la plupart des outils que les développeurs Pharo utilisent pour construire et déboguer leurs programmes et comment les programmes sont échangés entre les développeurs. Nous verrons notamment le navigateur de classes, l'inspecteur d'objet, le débogueur et le navigateur de paquetages Monticello. Le développement avec Smalltalk est efficace : vous découvrirez que vous passerez beaucoup plus de temps à écrire du code et beaucoup moins à gérer le processus de développement. Ceci est en partie au fait que Smalltalk est un langage très simple, et d'autre part que les outils qui forment l'environnement de programmation sont très intégrés avec le langage.

2.1 Le jeu Lights Out

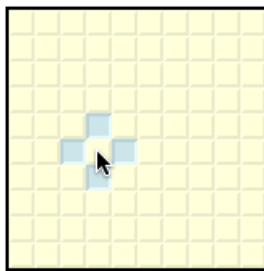


FIGURE 2.1 – Le plateau de jeu Lights Out. L'utilisateur vient de cliquer sur une case avec la souris comme le montre le curseur.

Pour vous montrer comment utiliser les outils de développement de

1. En anglais, [http://en.wikipedia.org/wiki/Lights_Out_\(game\)](http://en.wikipedia.org/wiki/Lights_Out_(game)).

Pharo, nous allons construire un jeu très simple nommé *Lights Out*. Le plateau de jeu est montré dans la figure 2.1 ; il consiste en un tableau rectangulaire de *cellules* jaunes claires. Lorsque l'on clique sur l'une de ces cellules avec la souris, les quatre qui l'entourent deviennent bleues. Cliquez de nouveau et elles repassent au jaune pâle. Le but du jeu est de passer au bleu autant de cellules que possible.

Le jeu *Lights Out* montré dans la figure 2.1 est fait de deux types d'objets : le plateau de jeu lui-même et une centaine de cellule-objets individuelles. Le code Pharo pour réaliser ce jeu va contenir deux classes : une pour le jeu et une autre pour les cellules. Nous allons voir maintenant comment définir ces deux classes en utilisant les outils de programmation de Pharo.

2.2 Créer un nouveau paquetage

Nous avons déjà vu le Browser dans le chapitre 1, où nous avons appris à naviguer dans les classes et méthodes, et à définir de nouvelles méthodes. Nous allons maintenant voir comment créer des paquetages (ou *packages*), des catégories et des classes.

 Ouvrez un Browser et cliquez avec le bouton d'action sur le panneau des paquetages. Sélectionnez `create package`².

Tapez le nom du nouveau paquetage (nous allons utiliser *PBE-LightsOut*) dans la boîte de dialogue et cliquez sur `accept` (ou appuyez simplement sur la touche entrée) ; le nouveau paquetage est créé et s'affiche dans la liste des paquetages en respectant l'ordre alphabétique.

2.3 Définir la classe LOCell

Pour l'instant, il n'y a aucune classe dans le nouveau paquetage. Cependant le panneau de code inférieur — qui est la zone principale d'édition — affiche un patron pour faciliter la création d'une nouvelle classe (voir la figure 2.3).

Ce modèle nous montre une expression Smalltalk qui envoie un message à la classe appelée `Object`, lui demandant de créer une sous-classe appelée `NameOfSubClass`. La nouvelle classe n'a pas de variables et devrait appartenir à la catégorie *PBE-LightsOut*.

2. Nous supposons que le **Package** Browser est installé en tant que navigateur de classes par défaut. Si le Browser ne ressemble pas à celui de la la figure 2.2, vous aurez besoin de changer le navigateur par défaut. Voyez la FAQ ??, p. ??.

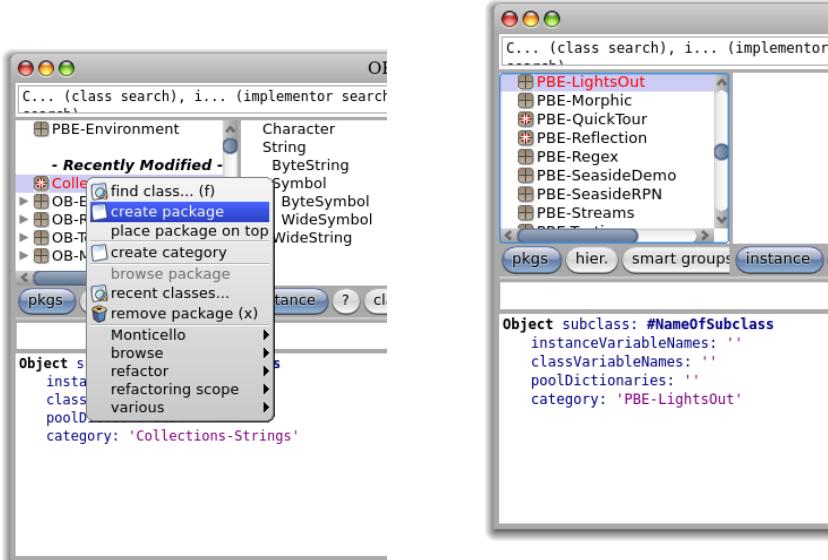


FIGURE 2.2 – Ajouter un paquetage.

FIGURE 2.3 – Le patron de création d'une classe.

À propos des catégories et des paquetages

Historiquement, Smalltalk ne connaît que les *catégories*. Vous pouvez vous interroger sur la différence qui peut exister entre catégories et paquetages. Une catégorie est simplement une collection de classes apparentées dans une image Smalltalk. Un *paquetage* (ou *package*) est une collection de classes apparentées *et de méthodes d'extension* qui peuvent être versionnées via l'outil de versionnage Monticello. Par convention, les noms de paquetages et les noms de catégories sont les mêmes. La plupart du temps, nous n'accordons pas de différence mais, dans ce livre, nous serons attentifs à utiliser la terminologie exacte car il y a des cas où la différence est cruciale. Vous en apprendrez plus lorsque nous aborderons le travail avec Monticello.

Créer une nouvelle classe

Nous modifions simplement le modèle afin de créer la classe que nous souhaitons.

Modifiez le modèle de création d'une classe comme suit :

- remplacez Object par SimpleSwitchMorph ;
- remplacez NameOfSubClass par LOCell ;

- ajoutez mouseAction dans la liste de variables d'instances.
- Le résultat doit ressembler à la classe 2.1.

Classe 2.1 – Définition de la classe LOCell

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

Cette nouvelle définition consiste en une expression Smalltalk qui envoie un message à une classe existante SimpleSwitchMorph, lui demandant de créer une sous-classe appelée LOCell (en fait, comme LOCell n'existe pas encore, nous passons comme argument le *symbole* #LOCell qui correspond au nom de la classe à créer). Nous indiquons également que les instances de cette nouvelle classe doivent avoir une variable d'instance mouseAction, que nous utiliserons pour définir l'action que la cellule doit effectuer lorsque l'utilisateur clique dessus avec la souris.

À ce point, nous n'avons encore rien construit. Notez que le bord du panneau du modèle de la classe est passé en rouge (voir la figure 2.4). Cela signifie qu'il y a des *modifications non sauvegardées*. Pour effectivement envoyer ce message, vous devez faire accept.

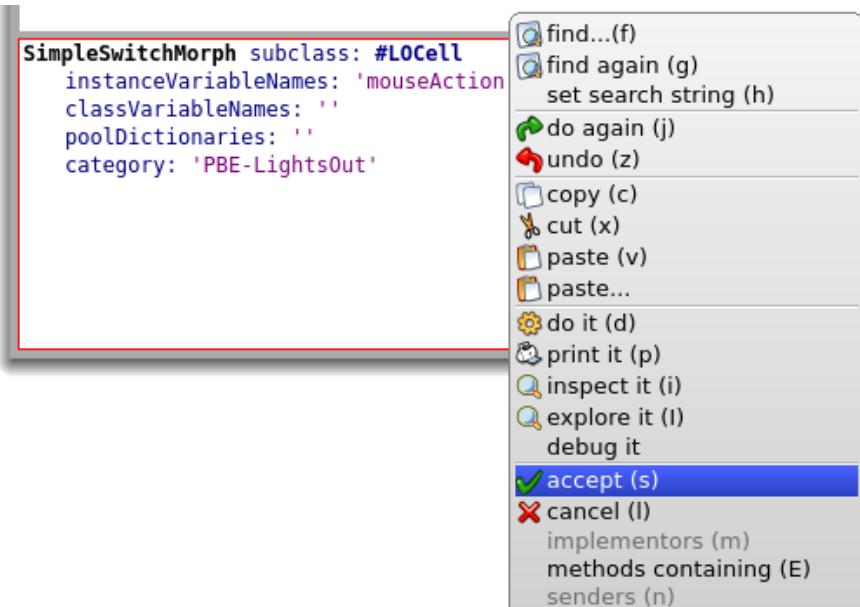


FIGURE 2.4 – Le modèle de création d'une classe.

⌚ Acceptez la nouvelle définition de classe.

Cliquez avec le bouton d'action et sélectionnez `accept` ou encore utilisez le raccourci-clavier CMD-s (pour “save” c-à-d. sauvegarder). Ce message sera envoyé à `SimpleSwitchMorph`, ce qui aura pour effet de compiler la nouvelle classe.

Une fois la définition de classe acceptée, la classe va être créée et apparaîtra dans le panneau des classes du navigateur (voir la figure 2.5). Le panneau d'édition montre maintenant la définition de la classe et un petit panneau dessous vous invite à écrire quelques mots décrivant l'objectif de la classe. Nous appelons cela un *commentaire de classe* ; il est assez important d'en écrire un qui donnera aux autres développeurs une vision globale de votre classe. Les Smalltalkiens accordent une grande valeur à la lisibilité de leur code et il n'est pas habituel de trouver des commentaires détaillés dans leurs méthodes ; la philosophie est plutôt d'avoir un code qui parle de lui-même (si cela n'est pas le cas, vous devrez le refactoriser jusqu'à ce que ça le soit!). Un commentaire de classe ne nécessite pas une description détaillée de la classe, mais quelques mots la décrivant sont vitaux si les développeurs qui viennent après vous souhaitent passer un peu de temps sur votre classe.

⌚ Tapez un commentaire de classe pour `LOCCell` et acceptez-le ; vous aurez tout le loisir de l'améliorer par la suite.

FIGURE 2.5 – La classe nouvellement créée `LOCCell`. Le panneau inférieur est le panneau de commentaires ; par défaut, il dit : “CETTE CLASSE N'A PAS DE COMMENTAIRE !”.

2.4 Ajouter des méthodes à la classe

Ajoutons maintenant quelques méthodes à notre classe.

⌚ Sélectionnez le protocole `--all--` dans le panneau des protocoles.

Vous voyez maintenant un modèle pour la création d'une méthode dans le panneau d'édition. Sélectionnez-le et remplacez-le par le texte de la méthode 2.2.

Méthode 2.2 – Initialiser les instances de LOCell.

```

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

```

Notez que les caractères " de la ligne 3 sont deux apostrophes³ sans espace entre elles, et non un guillemet (")! " représente la chaîne de caractères vide.

Faites un accept de cette définition de méthode.

Que fait le code ci-dessus ? Nous n'allons pas rentrer dans tous les détails maintenant (ce sera l'objet du reste de ce livre !), mais nous allons vous en donner un bref aperçu. Reprenons le code ligne par ligne.

Notons que la méthode s'appelle `initialize`. Ce nom dit bien ce qu'il veut dire⁴ ! Par convention, si une classe définit une méthode nommée `initialize`, cette méthode sera appelée dès que l'objet aura été créé. Ainsi dès que nous évaluons `LOCell new`, le message `initialize` sera envoyé automatiquement à cet objet nouvellement créé. Les méthodes d'initialisation sont utilisées pour définir l'état des objets, généralement pour donner une valeur à leurs variables d'instances ; c'est exactement ce que nous faisons ici.

La première action de cette méthode (ligne 2) est d'exécuter la méthode `initialize` de sa super-classe, `SimpleSwitchMorph`. L'idée est que tout état hérité sera initialisé correctement par la méthode `initialize` de la super-classe. C'est toujours une bonne idée d'initialiser l'état hérité en envoyant `super initialize` avant de faire tout autre chose ; nous ne savons pas exactement ce que la méthode `initialize` de `SimpleSwitchMorph` va faire, et nous ne nous en soucions pas, mais il est raisonnable de penser que cette méthode va initialiser quelques variables d'instance avec des valeurs par défaut, et qu'il vaut mieux le faire au risque de se retrouver dans un état incorrect.

Le reste de la méthode donne un état à cet objet. Par exemple, envoyer `self label: ""` affecte le label de cet objet avec la chaîne de caractères vide.

L'expression `0@0 corner: 16@16` nécessite probablement plus d'explications. `0@0` représente un objet `Point` dont les coordonnées *x* et *y* ont été fixées à 0. En fait, `0@0` envoie le message `@` au nombre 0 avec l'argument 0. L'effet produit sera que le nombre 0 va demander à la classe `Point` de créer une nouvelle instance de coordonnées (0,0). Puis, nous envoyons à ce nouveau point

3. Nous utilisons le terme “quote” en anglais.

4. En anglais, puisque c'est la langue conventionnelle en Smalltalk.

le message corner: 16@16, ce qui cause la création d'un Rectangle de coins 0@0 et 16@16. Ce nouveau rectangle va être affecté à la variable bounds héritée de la super-classe.

Notez que l'origine de l'écran Pharo est en *haut à gauche* et que les coordonnées en *y* augmentent vers *le bas*.

Le reste de la méthode doit être compréhensible de lui-même. Une partie de l'art d'écrire du bon code Smalltalk est de choisir les bons noms de méthodes de telle sorte que le code Smalltalk peut être lu comme de l'anglais simplifié (*English pidgin*). Vous devriez être capable d'imaginer l'objet se parlant à lui-même et dire : "Utilise des bords carrés !" (d'où `useSquareCorners`), "Éteins les cellules !" (en anglais, `turnOff`).

2.5 Inspecter un objet

Vous pouvez tester l'effet du code que vous avez écrit en créant un nouvel objet `LOCCell` et en l'inspectant avec l'inspecteur nommé **Inspector**. ⓘ

Ouvrez un espace de travail (*Workspace*). Tapez l'expression `LOCCell new` et choisissez `inspect it`.

FIGURE 2.6 – L'inspecteur utilisé pour examiner l'objet `LOCCell`.

Le panneau gauche de l'inspecteur montre une liste de variables d'instances ; si vous en sélectionnez une (par exemple `bounds`), la valeur de la variable d'instance est affichée dans le panneau droit.

Le panneau en bas d'un inspecteur est un mini-espace de travail. C'est très utile car, dans cet espace de travail, la pseudo-variable `self` est liée à l'objet sélectionné.

ⓘ Sélectionnez `LOCCell` à la racine de la fenêtre de l'inspecteur. Saisissez l'expression `self bounds: (200@200 corner: 250@250)` dans le panneau inférieur et faites un `do it` (via le menu contextuel ou le raccourci-clavier).

La variable `bounds` devrait changer dans l'inspecteur. Saisissez maintenant `self openInWorld` dans ce même panneau et évaluez le code avec `do it`. La cellule doit apparaître près du coin supérieur gauche, là où les coordonnées `bounds` doivent le faire apparaître. Meta-cliquez sur la cellule afin de faire apparaître son halo Morphic.

Déplacez la cellule avec la poignée marron (à gauche de l'icône du coin supérieur droit) et redimensionnez-la avec la poignée jaune (en bas à droite). Vérifiez que les limites indiquées par l'inspecteur sont modifiées en conséquence (il faudra peut-être cliquer avec le bouton d'action sur `refresh` pour

voir les nouvelles valeurs).

FIGURE 2.7 – Redimensionner la cellule.

- ⌚ Détruisez la cellule en cliquant sur le x de la poignée rose pâle (en haut à gauche).

2.6 Définir la classe LOGame

Créons maintenant l'autre classe dont nous avons besoin dans le jeu ; nous l'appellerons LOGame.

- ⌚ Faites apparaître le modèle de définition de classe dans la fenêtre principale du navigateur.

Pour cela, cliquez sur le nom du paquetage. Éditez le code de telle sorte qu'il puisse être lu comme suit puis faites `accept`.

Classe 2.3 – Définition de la classe LOGame

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

Ici nous sous-classons BorderedMorph ; Morph est la super-classe de toutes les formes graphiques de Pharo, et (surprise !) un BorderedMorph est un Morph avec un bord. Nous pourrions également insérer les noms des variables d'instances entre apostrophes sur la seconde ligne, mais pour l'instant laissons cette liste vide.

Définissons maintenant une méthode `initialize` pour LOGame.

- ⌚ Tapez ce qui suit dans le navigateur comme une méthode de LOGame et faites ensuite `accept` :

Méthode 2.4 – Initialisation du jeu

```
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 va se plaindre qu'il ne connaît pas la signification de certains termes. Il vous indique alors qu'il ne connaît pas le message `cellsPerSide` (en français, “cellules par côté”) et vous suggère un certain nombre de propositions, dans le cas où il s'agirait d'une erreur de frappe.

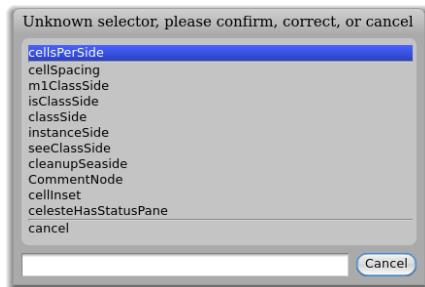


FIGURE 2.8 – Pharo détecte un sélecteur inconnu.

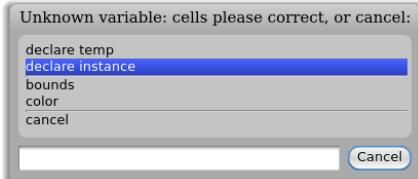


FIGURE 2.9 – Déclaration d'une nouvelle variable d'instance.

Mais `cellsPerSide` n'est pas une erreur — c'est juste le nom d'une méthode que nous n'avons pas encore définie — que nous allons écrire dans une minute ou deux.

Sélectionnez la première option du menu, afin de confirmer que nous parlons bien de `cellsPerSide`.

Puis, Pharo va se plaindre de ne pas connaître la signification de `cells`. Il vous offre plusieurs possibilités de correction.

Choisissez `declare instance` parce que nous souhaitons que `cells` soit une variable d'instance.

Enfin, Pharo va se plaindre à propos du message `newCellAt:at:` envoyé à la dernière ligne ; ce n'est pas non plus une erreur, confirmez donc ce message aussi.

Si vous regardez maintenant de nouveau la définition de classe (en cliquant sur le bouton `[instance]`), vous allez voir que la définition a été modifiée pour inclure la variable d'instance `cells`.

Examinons plus précisément cette méthode `initialize`. La ligne `| sampleCell width height n |` déclare 4 variables temporaires. Elles sont appelées variables temporaires car leur portée et leur durée de vie sont limitées à cette méthode. Des variables temporaires avec des noms explicites sont utiles afin de rendre le code plus lisible. Smalltalk n'a pas de syntaxe spéciale pour dis-

tinguer les constantes et les variables et en fait, ces 4 “variables” sont ici des constantes. Les lignes 4 à 7 définissent ces constantes.

Quelle doit être la taille de notre plateau de jeu ? Assez grande pour pouvoir contenir un certain nombre de cellules et pour pouvoir dessiner un bord autour d’elles. Quel est le bon nombre de cellules ? 5 ? 10 ? 100 ? Nous ne le savons pas pour l’instant et si nous le savions, il y aurait des chances pour que nous changions d’idée par la suite. Nous déléguons donc la responsabilité de connaître ce nombre à une autre méthode, que nous appelons `cellsPerSide` et que nous écrirons bientôt. C’est parce que nous envoyons le message `cellsPerSide` avant de définir une méthode avec ce nom que Pharo nous demande “*confirm, correct, or cancel*” (c-à-d. “confirmez, corrigez ou annulez”) lorsque nous acceptons le corps de la méthode `initialize`. Que cela ne vous inquiète pas : c’est en fait une bonne pratique d’écrire en fonction d’autres méthodes qui ne sont pas encore définies. Pourquoi ? En fait, ce n’est que quand nous avons commencé à écrire la méthode `initialize` que nous nous sommes rendu compte que nous en avions besoin, et à ce point, nous lui avons donné un nom significatif et nous avons poursuivi, sans nous interrompre.

La quatrième ligne utilise cette méthode : le code Smalltalk `self cellsPerSide` envoie le message `cellsPerSide` à `self`, c-à-d. à l’objet lui-même. La réponse, qui sera le nombre de cellules par côté du plateau de jeu, est affectée à `n`.

Les trois lignes suivantes créent un nouvel objet `LOCCell` et assignent sa largeur et sa hauteur aux variables temporaires appropriées.

La ligne 8 fixe la valeur de `bounds` (définissant les limites) du nouvel objet. Ne vous inquiétez pas trop sur les détails pour l’instant. Croyez-nous : l’expression entre parenthèses crée un carré avec comme origine (c-à-d. son coin haut à gauche) le point (5,5) et son coin bas droit suffisamment loin afin d’avoir de l’espace pour le bon nombre de cellules.

La dernière ligne affecte la variable d’instance `cells` de l’objet `LOGame` à un nouvel objet `Matrix` avec le bon nombre de lignes et de colonnes. Nous réalisons cela en envoyant le message `new:tabulate:` à la classe `Matrix` (les classes sont des objets aussi, nous pouvons leur envoyer des messages). Nous savons que `new:tabulate:` prend deux arguments parce qu’il y a deux fois deux points (:) dans son nom. Les arguments arrivent à droite après les deux points. Si vous êtes habitué à des langages de programmation où les arguments sont tous mis à l’intérieur de parenthèses, ceci peut sembler surprenant dans un premier temps. Ne vous inquiétez pas, c’est juste de la syntaxe ! Cela s’avère être une excellente syntaxe car le nom de la méthode peut être utilisé pour expliquer le rôle des arguments. Par exemple, il est très clair que `Matrix rows:5 columns:2` a 5 lignes et 2 colonnes et non pas 2 lignes et 5 colonnes.

`Matrix new: n tabulate: [:i :j | self newCellAt: i at: j]` crée une nouvelle matrice de

taille $n \times n$ et initialise ses éléments. La valeur initiale de chaque élément dépend de ses coordonnées. L'élément (i,j) sera initialisé avec le résultat de l'évaluation de self newCellAt: i at: j.

2.7 Organiser les méthodes en protocoles

Avant de définir de nouvelles méthodes, attardons-nous un peu sur le troisième panneau en haut du navigateur. De la même façon que le premier panneau du navigateur nous permet de catégoriser les classes dans des paquetages de telle sorte que nous ne soyons pas submergés par une liste de noms de classes trop longue dans le second panneau, le troisième panneau nous permet de catégoriser les méthodes de telle sorte que n'ayons pas une liste de méthodes trop longue dans le quatrième panneau. Ces catégories de méthodes sont appelées "protocoles".

S'il n'y avait que quelques méthodes par classe, ce niveau hiérarchique supplémentaire ne serait pas vraiment nécessaire. C'est pour cela que le navigateur offre un protocole virtuel `--all--` (*c-à-d.* "tout" en français) qui, vous ne serez pas surpris de l'apprendre, contient toutes les méthodes de la classe.

Si vous avez suivi l'exemple jusqu'à présent, le troisième panneau doit contenir le protocole *as yet unclassified*⁵.

 Cliquez avec le bouton d'action dans le panneau des protocoles et sélectionnez `various > categorize automatically` afin de régler ce problème et déplacer les méthodes `initialize` vers un nouveau protocole appelé `initialization`.

Comment Pharo sait que c'est le bon protocole ? En général, Pharo ne peut pas le savoir mais dans notre cas, il y a aussi une méthode `initialize` dans la super-classe et Pharo suppose que notre méthode `initialize` doit être rangée dans la même catégorie que celle qu'elle surcharge.

Une convention typographique. Les Smalltalkiens utilisent fréquemment la notation "`>>`" afin d'identifier la classe à laquelle la méthode appartient, ainsi par exemple, la méthode `cellsPerSide` de la classe `LOGame` sera référencée par `LOGame>>cellsPerSide`. Afin d'indiquer que cela ne fait pas partie de la syntaxe de Smalltalk, nous utiliserons plutôt le symbole spécial » de telle sorte que cette méthode apparaîtra dans le texte comme `LOGame»cellsPerSide`

À partir de maintenant, lorsque nous voudrons montrer une méthode dans ce livre, nous écrirons le nom de cette méthode sous cette forme. Bien sûr, lorsque vous tapez le code dans un navigateur, vous n'avez pas à taper

5. NdT : non encore classifié.

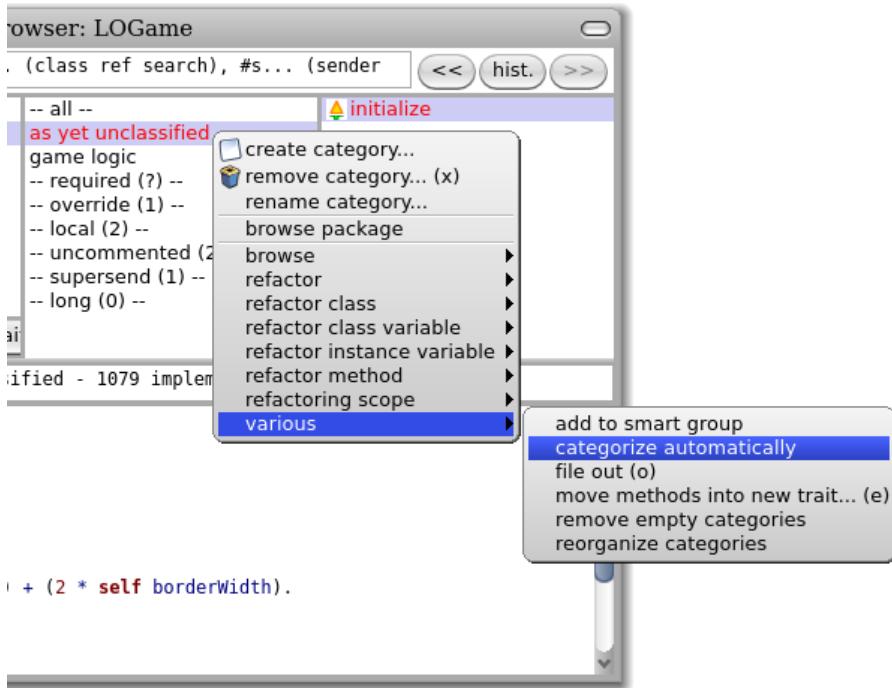


FIGURE 2.10 – Catégoriser de façon automatique toutes les méthodes non catégorisées.

le nom de la classe ou le » ; vous devrez juste vous assurer que la classe appropriée est sélectionnée dans le panneau des classes.

Définissons maintenant les autres méthodes qui sont utilisées par la méthode `LOGame»initialize`. Les deux peuvent être mises dans le protocole *initialization*.

Méthode 2.5 – Une méthode constante

```
LOGame»cellsPerSide
  "Le nombre de cellules le long de chaque côté du jeu"
  ↑ 10
```

Cette méthode ne peut pas être plus simple : elle retourne la constante 10. Représenter les constantes comme des méthodes a comme avantage que si le programme évolue de telle sorte que la constante dépende d'autres propriétés, la méthode peut être modifiée pour calculer la valeur.

Méthode 2.6 – Une méthode d'aide à l'initialisation

```
LOGame»newCellAt: i at: j
    "Crée une cellule à la position (i,j) et l'ajoute dans ma représentation graphique à la
     position correcte. Retourne une nouvelle cellule"
| 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]
```

 Ajoutez les méthodes LOGame»cellsPerSide et LOGame»newCellAt:at:.

Confirmez que les sélecteurs `toggleNeighboursOfCellAt:at:` et `mouseAction:` s'épellent correctement.

La méthode 2.6 retourne une nouvelle cellule `LOCell` à la position (*i,j*) dans la matrice (`Matrix`) de cellules. La dernière ligne définit l'action de la souris (`mouseAction`) associée à la cellule comme le *bloc* `[self toggleNeighboursOfCellAt:i at:j]`. En effet, ceci définit le comportement de rappel ou *callback* à effectuer lorsque nous cliquons à la souris. La méthode correspondante doit être aussi définie.

Méthode 2.7 – La méthode callback

```
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]
```

La méthode 2.7 (traduisible par “change les voisins de la cellule...”) change l'état des 4 cellules au nord, sud, ouest et est de la cellule (*i, j*). La seule complication est que le plateau de jeu est fini. Il faut donc s'assurer qu'une cellule voisine existe avant de changer son état.

 Placez cette méthode dans un nouveau protocole appelé `game logic` (pour “logique du jeu”) et créé en cliquant avec le bouton d'action dans le panneau des protocoles.

Pour déplacer cette méthode, vous devez simplement cliquer sur son nom puis la glisser-déposer sur le nouveau protocole (voir la figure 2.11).

Afin de compléter le jeu `Lights Out`, nous avons besoin de définir encore deux méthodes dans la classe `LOCell` pour gérer les événements souris.

Méthode 2.8 – Un mutateur typique

```
LOCell»mouseAction: aBlock
↑ mouseAction := aBlock
```

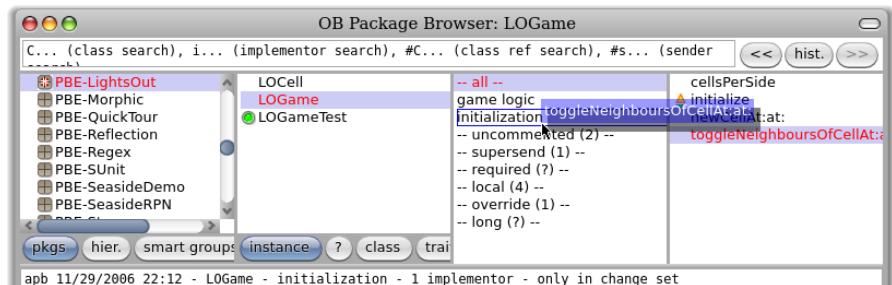


FIGURE 2.11 – Faire un glisser-déposer de la méthode dans un protocole.

La seule action de la méthode 2.8 consiste à donner comme valeur à la variable `mouseAction` celle de l'argument puis, à en retourner la nouvelle valeur. Toute méthode qui *change* la valeur d'une variable d'instance de cette façon est appelée une *méthode d'accès en écriture* ou *mutateur* (vous pourrez trouver dans la littérature le terme anglais *setter*) ; une méthode qui *retourne* la valeur courante d'une variable d'instance est appelée une *méthode d'accès en lecture* ou *accesseur* (le mot anglais équivalent est *getter*).

Si vous êtes habitués aux méthodes d'accès en lecture (*getter*) et écriture (*setter*) dans d'autres langages de programmation, vous vous attendez à avoir deux méthodes nommées `getMouseAction` et `setMouseAction`. La convention en Smalltalk est différente. Une méthode d'accès en lecture a toujours le même nom que la variable correspondante et la méthode d'accès en écriture est nommée de la même manière avec un ":" à la fin ; ici nous avons donc `mouseAction` et `mouseAction:`.

Une méthode d'accès (en lecture ou en écriture) est appelée en anglais *accessor* et par convention, elle doit être placée dans le protocole *accessing*. En Smalltalk, toutes les variables d'instances sont privées à l'objet qui les possède, ainsi la seule façon pour un autre objet de lire ou de modifier ces variables en Smalltalk se fait au travers de ces méthodes d'accès comme ici⁶.

Allez à la classe LOCell, définissez LOCell»mouseAction: et mettez-la dans le protocole accessing.

Finalement, vous avez besoin de définir la méthode `mouseUp:` ; elle sera appellée automatiquement par l'infrastructure (ou *framework*) graphique si le bouton de la souris est pressé lorsque le pointeur de celle-ci est au-dessus d'une cellule sur l'écran.

Méthode 2.9 – Un gestionnaire d'événement

LOCell»mouseUp: anEvent

6. En fait, les variables d'instances peuvent être accédées également dans les sous-classes.

```
mouseAction value
```

-  Ajoutez la méthode LOCell»mouseUp: définissant l'action lorsque le bouton de la souris est relâché puis, faites categorize automatically.

Que fait cette méthode ? Elle envoie le message value à l'objet stocké dans la variable d'instance mouseAction. Rappelez-vous que dans la méthode LOGame»newCellAt: i at: j nous avons affecté le fragment de code qui suit à mouseAction :

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

Envoyer le message value provoque l'évaluation de ce bloc (toujours entre crochets, voir le chapitre 3) et, par voie de conséquence, est responsable du changement d'état des cellules.

2.8 Essayons notre code

Voilà, le jeu Lights Out est complet !

Si vous avez suivi toutes les étapes, vous devriez pouvoir jouer au jeu qui comprend 2 classes et 7 méthodes.

-  Dans un espace de travail, tapez LOGame new openInWorld et faites do it.

Le jeu devrait s'ouvrir et vous devriez pouvoir cliquer sur les cellules et vérifier si le jeu fonctionne.

Du moins en théorie... Lorsque vous cliquez sur une cellule une fenêtre de notification appelée la fenêtre PreDebugWindow devrait apparaître avec un message d'erreur ! Comme nous pouvons le voir sur la figure 2.12, elle dit MessageNotUnderstood: LOGame»toggleState.

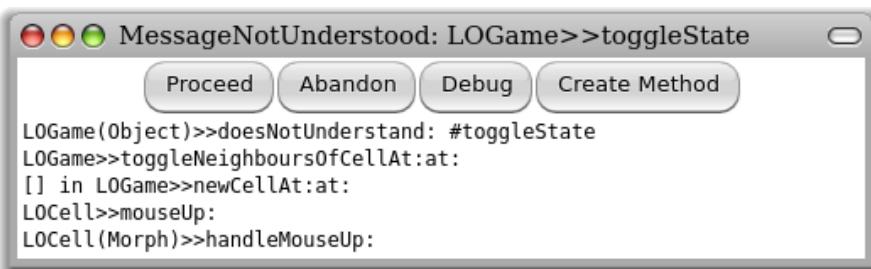


FIGURE 2.12 – Il y a une erreur dans notre jeu lorsqu'une cellule est sélectionnée !

Que se passe-t-il ? Afin de le découvrir, utilisons l'un des outils les plus puissants de Smalltalk, le débogueur.

 Cliquez sur le bouton `debug` de la fenêtre de notification.

Le débogueur nommé Debugger devrait apparaître. Dans la partie supérieure de la fenêtre du débogueur, nous pouvons voir la pile d'exécution, affichant toutes les méthodes actives ; en sélectionnant l'une d'entre elles, nous voyons dans le panneau du milieu le code Smalltalk en cours d'exécution dans cette méthode, avec la partie qui a déclenchée l'erreur en caractère gras.

 Cliquez sur la ligne nommée `LOGame»toggleNeighboursOfCellAt:at:` (près du haut).

Le débogueur vous montrera le contexte d'exécution à l'intérieur de la méthode où l'erreur s'est déclenchée (voir la figure 2.13).

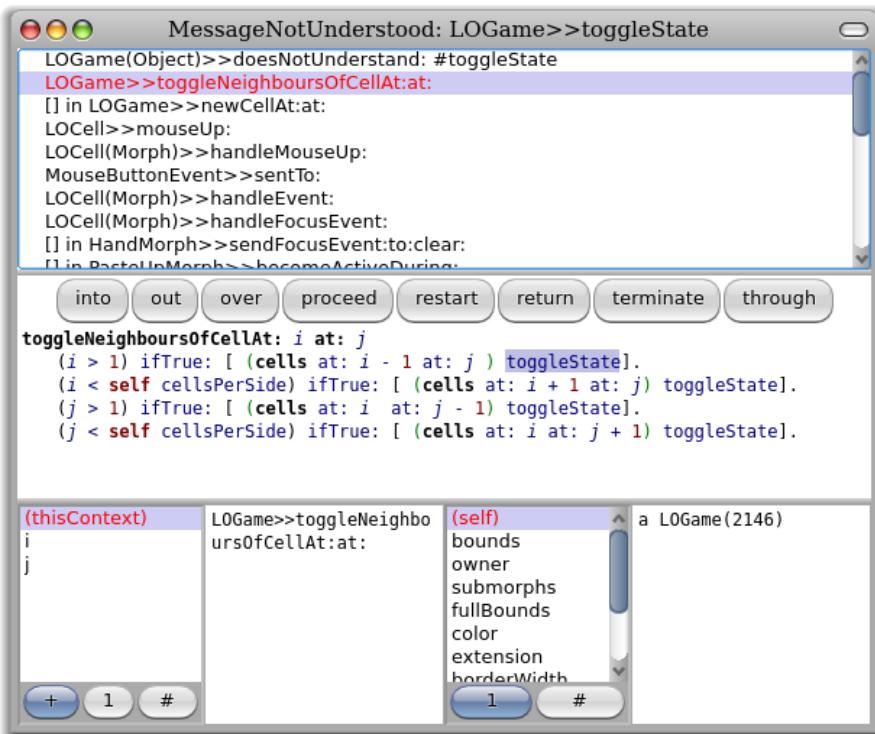


FIGURE 2.13 – Le débogueur avec la méthode `toggleNeighboursOfCellAt:` sélectionnée.

Dans la partie inférieure du débogueur, il y a deux petites fenêtres d'ins-

pection. Sur la gauche, vous pouvez inspecter l'objet-recepteur du message qui cause l'exécution de la méthode sélectionnée. Vous pouvez voir ici les valeurs des variables d'instance. Sur la droite, vous pouvez inspecter l'objet qui représente la méthode en cours d'exécution. Il est possible d'examiner ici les valeurs des paramètres et les variables temporaires.

En utilisant le débogueur, vous pouvez exécuter du code pas à pas, inspecter les objets dans les paramètres et les variables locales, évaluer du code comme vous le faites dans le Workspace et, de manière surprenante pour ceux qui sont déjà habitués à d'autres débogueurs, il est possible de modifier le code en cours de débogage ! Certains Smalltalkiens programment la plupart du temps dans le débogueur, plutôt que dans le navigateur de classes. L'avantage est certain : la méthode que vous écrivez est telle qu'elle sera exécutée *c-à-d.* avec ses paramètres dans son contexte actuel d'exécution.

Dans notre cas, vous pouvez voir dans la première ligne du panneau du haut que le message `toggleState` a été envoyé à une instance de `LOGame`, alors qu'il était clairement destiné à une instance de `LOCcell`. Le problème se situe vraisemblablement dans l'initialisation de la matrice `cells`. En parcourant le code de `LOGame»initialize`, nous pouvons voir que `cells` est rempli avec les valeurs rentrées par `newCellAt:at:`, mais lorsque nous regardons cette méthode, nous constatons qu'il n'y a pas de valeur rentrée ici ! Par défaut, une méthode retourne `self`, ce qui dans le cas de `newCellAt:at:` est effectivement une instance de `LOGame`.

 Fermez la fenêtre du débogueur. Ajoutez l'expression "`↑ c`" à la fin de la méthode `LOGame»newCellAt:at:` de telle sorte qu'elle retourne `c` (voir la méthode 2.10).

Méthode 2.10 – Corriger l'erreur

```
LOGame»newCellAt: i at: j
```

"Crée une cellule à la position (i,j) et l'ajoute dans ma représentation graphique à la position correcte. Retourne une nouvelle cellule"

```
| c origin |
c := LOCcell 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
```

Rappelez-vous ce que nous avons vu dans le chapitre 1 : pour renvoyer une valeur d'une méthode en Smalltalk, nous utilisons `↑`, que nous pouvons obtenir en tapant `^`.

Il est souvent possible de corriger le code directement dans la fenêtre du débogueur et de poursuivre l'application en cliquant sur `Proceed`. Dans notre cas, la chose la plus simple à faire est de fermer la fenêtre du débogueur,

détruire l’instance en cours d’exécution (avec le halo Morphic) et d’en créer une nouvelle, parce que le bug ne se situe pas dans une méthode erronée mais dans l’initialisation de l’objet.

 **Exécutez** `LOGame new openInWorld` *de nouveau.*

Le jeu doit maintenant se dérouler sans problèmes ... ou presque ! S’il vous arrive de bouger la souris entre le moment où vous cliquez et le moment où vous relâchez le bouton de la souris, la cellule sur laquelle se trouve la souris sera aussi changée. Ceci résulte du comportement hérité de `SimpleSwitchMorph`. Nous pouvons simplement corriger cela en surchargeant `mouseMove:` pour lui dire de ne rien faire :

Méthode 2.11 – *Surcharger les actions associées aux déplacements de la souris*

```
LOGame»mouseMove: anEvent
```

Et voilà !

2.9 Sauvegarder et partager le code Smalltalk

Maintenant que nous avons un jeu Lights Out fonctionnel, vous avez probablement envie de le sauvegarder quelque part de telle sorte à pouvoir le partager avec des amis. Bien sûr, vous pouvez sauvegarder l’ensemble de votre image Pharo et montrer votre premier programme en l’exécutant, mais vos amis ont probablement leur propre code dans leurs images et ne veulent pas s’en passer pour utiliser votre image. Nous avons donc besoin de pouvoir extraire le code source d’une image Pharo afin que d’autres développeurs puissent le charger dans leurs images.

La façon la plus simple de le faire est d’effectuer une exportation ou sortie-fichier (*filing out*) de votre code. Le menu activé en cliquant avec le bouton d’action dans le panneau des paquetages vous permet de générer un fichier correspondant au paquetage `PBE-LightsOut` tout entier via l’option `various > file out`. Le fichier résultant est plus lisible par tout un chacun, même si son contenu est plutôt destiné aux machines qu’aux hommes. Vous pouvez envoyer par email ce fichier à vos amis et ils peuvent le charger dans leurs propres images Pharo en utilisant le navigateur de fichiers File List Browser.

 **Cliquez avec le bouton d’action sur le paquetage** `PBE-LightsOut` **et choisissez** `various > file out` **pour exporter le contenu.**

Vous devriez trouver maintenant un fichier `PBE-LightsOut.st` dans le même répertoire où votre image a été sauvegardée. Jetez un coup d’œil à ce fichier avec un éditeur de texte.

 Ouvrez une nouvelle image Pharo et utilisez l'outil File Browser (Tools > File Browser) pour faire une importation de fichier via l'option de menu file in dans le fichier PBE-LightsOut.st. Vérifiez que le jeu fonctionne maintenant dans une nouvelle image.

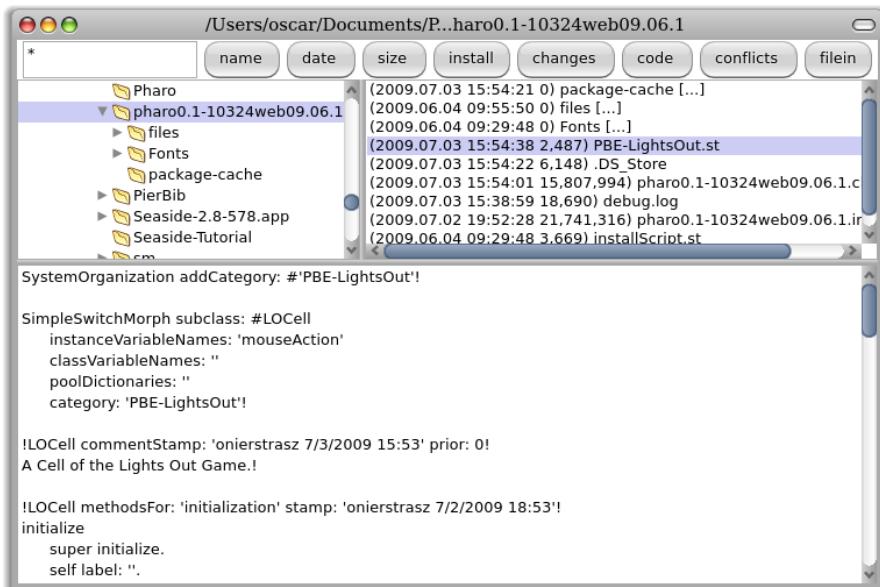


FIGURE 2.14 – Charger le code source dans Pharo.

Les paquetages Monticello

Bien que les exportations de fichiers soient une façon convenable de faire des sauvegardes du code que vous avez écrit, elles font maintenant partie du passé. Tout comme la plupart des développeurs de projets libres Open-Source qui trouvent plus utile de maintenir leur code dans des dépôts en utilisant CVS⁷ ou Subversion⁸, les programmeurs sur Pharo gèrent maintenant leur code au moyen de paquetages Monticello (dit, en anglais, *packages*) : ces paquetages sont représentés comme des fichiers dont le nom se termine en .mcz ; ce sont en fait des fichiers compressés en zip qui contiennent le code complet de votre paquetage.

En utilisant le navigateur de paquetages Monticello, vous pouvez sauver les paquetages dans des dépôts en utilisant de nombreux types de serveurs,

7. <http://www.nongnu.org/cvs>

8. <http://subversion.tigris.org>

notamment des serveurs FTP et HTTP ; vous pouvez également écrire vos paquetages dans un dépôt qui se trouve dans un répertoire de votre système local de fichiers. Une copie de votre paquetage est toujours en cache sur disque local dans le répertoire *package-cache*. Monticello vous permet de sauver de multiples versions de votre programme, fusionner des versions, revenir à une ancienne version et voir les différences entre plusieurs versions. En fait, nous retrouvons les mêmes types d'opérations auxquelles vous pourriez être habitués en utilisant CVS ou Subversion pour partager votre travail.

Vous pouvez également envoyer un fichier .mcz par email. Le destinataire devra le placer dans son répertoire *package-cache* ; il sera alors capable d'utiliser Monticello pour le parcourir et le charger.

 Ouvrez le navigateur Monticello ou Monticello Browser depuis le menu World.

Dans la partie droite du navigateur (voir la figure 2.15), il y a une liste des dépôts Monticello incluant tous les dépôts dans lesquels du code a été chargé dans l'image que vous utilisez.

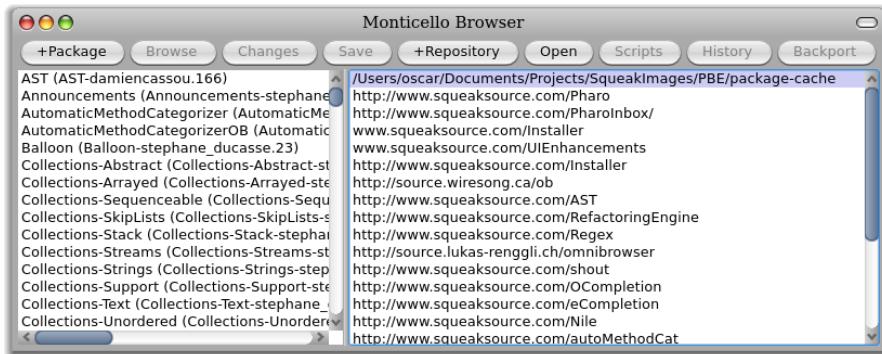


FIGURE 2.15 – Le navigateur Monticello.

En haut de la liste dans le navigateur Monticello, il y a un dépôt dans un répertoire local appelé *package cache* : il s'agit d'un répertoire-cache pour des copies de paquetages que vous avez chargées ou publiées sur le réseau. Ce cache est vraiment utile car il vous permet de garder votre historique local. Il vous permet également de travailler là où vous n'avez pas d'accès Internet ou lorsque l'accès est si lent que vous n'avez pas envie de sauver fréquemment sur un dépôt distant.

Sauvegarder et charger du code avec Monticello

Dans la partie gauche du navigateur Monticello, il y a une liste de paquetages dont vous avez une version chargée dans votre image ; les paquetages qui ont été modifiés depuis qu'ils ont été chargés sont marqués d'une astérisque (ils sont parfois appelés des *dirty packages*). Si vous sélectionnez un paquetage, la liste des dépôts est restreinte à ceux qui contiennent une copie du paquetage sélectionné.

 Ajoutez le paquetage PBE–LightsOut à votre navigateur Monticello en utilisant le bouton **+Package**.

SqueakSource : un SourceForge pour Pharo

Nous pensons que la meilleure façon de sauvegarder votre code et de le partager est de créer un compte sur un serveur SqueakSource. SqueakSource est similaire à SourceForge⁹ : il s'agit d'un *portail web* à un serveur Monticello HTTP qui vous permet de gérer vos projets. Il y a un serveur public SqueakSource à l'adresse <http://www.squeaksource.com> et une copie du code concernant ce livre est enregistrée sur <http://www.squeaksource.com/PharoByExample.html>. Vous pouvez consulter ce projet à l'aide d'un navigateur internet, mais il est beaucoup plus productif de le faire depuis Pharo en utilisant l'outil *ad hoc*, le navigateur Monticello, qui vous permet de gérer vos paquetages.

 Ouvrez un navigateur web à l'adresse <http://www.squeaksource.com>. Ouvrir un compte et ensuite, créez un projet (c-à-d. via "register") pour le jeu Lights Out.

SqueakSource va vous montrer l'information que vous devez utiliser lorsque nous ajoutons un dépôt au moyen de Monticello.

Une fois que votre projet a été créé sur SqueakSource, vous devez indiquer au système Pharo de l'utiliser.

 Avec le paquetage PBE–LightsOut sélectionné, cliquez sur le bouton **+Repository** dans le navigateur Monticello.

Vous verrez une liste des différents types de dépôts disponibles ; pour ajouter un dépôt SqueakSource, sélectionner le menu **HTTP**. Une boîte de dialogue vous permettra de rentrer les informations nécessaires pour le serveur. Vous devez copier le modèle ci-dessous pour identifier votre projet SqueakSource, copiez-le dans Monticello en y ajoutant vos initiales et votre mot de passe :

9. <http://www.sourceforge.net>

```
MCHttpRepository
location: 'http://www.squeaksource.com/VotreProjet'
user: 'vosInitiales'
password: 'votreMotDePasse'
```

Si vous passez en paramètres des initiales et un mot de passe vide, vous pouvez toujours charger le projet, mais vous ne serez pas autorisé à le mettre à jour :

```
MCHttpRepository
location: 'http://www.squeaksource.com/SqueakByExample'
user: ""
password: "
```

Une fois que vous avez accepté ce modèle, un nouveau dépôt doit apparaître dans la partie droite du navigateur Monticello.

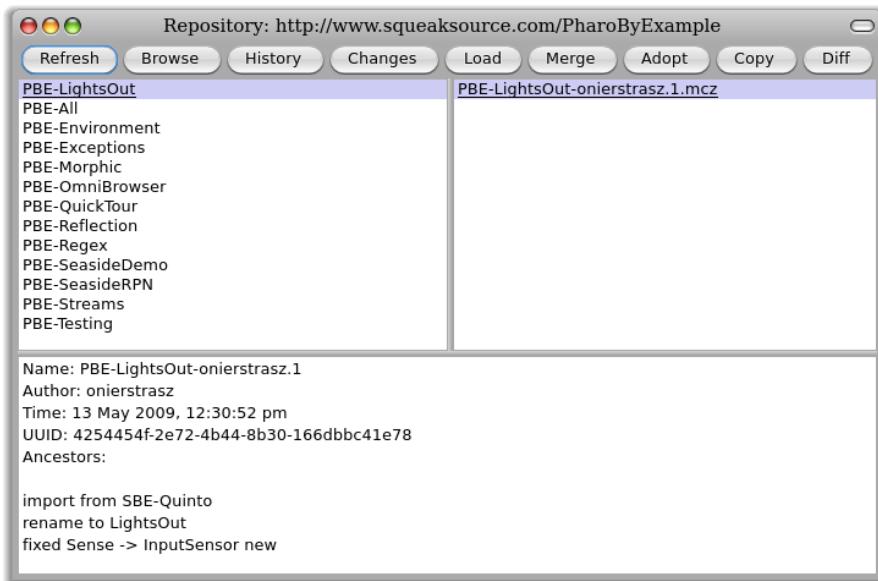


FIGURE 2.16 – Parcourir un dépôt Monticello.

Cliquez sur le bouton **Save** pour faire une première sauvegarde du jeu *Lights Out* sur SqueakSource.

Pour charger un paquetage dans votre image, vous devez d'abord sélectionner une version particulière. Vous pouvez faire cela dans le navigateur de dépôts *Repository Browser*, que vous pouvez ouvrir avec le bouton **Open**

ou en cliquant avec le bouton d'action pour choisir `open repository` dans le menu contextuel. Une fois que vous avez sélectionné une version, vous pouvez la charger dans votre image.

❸ *Ouvrez le dépôt PBE–LightsOut que vous venez de sauvegarder.*

Monticello a beaucoup d'autres fonctionnalités qui seront discutées plus en détail dans le chapitre 6. Vous pouvez également consulter la documentation en ligne de Monticello à l'adresse <http://www.wiresong.ca/Monticello/>.

2.10 Résumé du chapitre

Dans ce chapitre, nous avons vu comment créer des catégories, des classes et des méthodes. Nous avons vu aussi comment utiliser le navigateur de classes (Browser), l'inspecteur (Inspector), le débogueur (Debugger) et le navigateur Monticello.

- Les catégories sont des groupes de classes connexes.
- Une nouvelle classe est créée en envoyant un message à sa super-classe.
- Les protocoles sont des groupes de méthodes apparentées.
- Une nouvelle méthode est créée ou modifiée en éditant la définition dans le navigateur de classes et en acceptant les modifications.
- L'inspecteur offre une manière simple et générale pour inspecter et interagir avec des objets arbitraires.
- Le navigateur de classes détecte l'utilisation de méthodes et de variables non déclarées et propose d'éventuelles corrections.
- La méthode `initialize` est automatiquement exécutée après la création d'un objet dans Pharo. Vous pouvez y mettre le code d'initialisation que vous voulez.
- Le débogueur est une interface de haut niveau pour inspecter et modifier l'état d'un programme en cours d'exécution.
- Vous pouvez partager le code source en sauvegardant une catégorie sous forme d'un fichier d'exportation.
- Une meilleure façon de partager le code consiste à faire appel à Monticello afin de gérer un dépôt externe défini, par exemple, comme un projet SqueakSource.

Chapitre 3

Un résumé de la syntaxe

Pharo, comme la plupart des dialectes modernes de Smalltalk, adopte une syntaxe proche de celle de Smalltalk-80. La syntaxe est conçue de telle sorte que le texte d'un programme lu à haute voix ressemble à de l'*English pidgin* ou "anglais simplifié" :

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

La syntaxe de Pharo (c-à-d. les expressions) est minimaliste ; pour l'essentiel, conçue uniquement pour envoyer des messages. Les expressions sont construites à partir d'un nombre très réduit de primitives. Smalltalk dispose seulement de 6 mots-clés et d'aucune syntaxe pour les structures de contrôle, ni pour les déclarations de nouvelles classes. En revanche, tout ou presque est réalisable en envoyant des messages à des objets. Par exemple, à la place de la structure de contrôle conditionnelle *si-alors-sinon*, Smalltalk envoie des messages comme `ifTrue:` à des objets de la classe Boolean. Les nouvelles (sous-)classes sont créées en envoyant un message à leur super-classe.

3.1 Les éléments syntaxiques

Les expressions sont composées des blocs constructeurs suivants :

- (i) six mots-clés réservés ou *pseudo-variables* : `self`, `super`, `nil`, `true`, `false`, `and` `thisContext` ;
- (ii) des expressions constantes pour des *objets littéraux* comprenant les nombres, les caractères, les chaînes de caractères, les symboles et les tableaux ;
- (iii) des déclarations de variables ;
- (iv) des affectations ;

- (v) des blocs ou fermetures lexicales – *block closures* en anglais – et ;
- (vi) des messages.

Syntaxe	ce qu'elle représente
startPoint	un nom de variable
Transcript	un nom de variable globale
self	une pseudo-variable
1	un entier décimal
2r101	un entier binaire
1.5	un nombre flottant
2.4e7	une notation exponentielle
\$a	le caractère 'a'
'Bonjour'	la chaîne "Bonjour"
#Bonjour	le symbole #Bonjour
#{1 2 3}	un tableau de littéraux
{1. 2. 1+2}	un tableau dynamique
"c'est mon commentaire"	un commentaire
x y	une déclaration de 2 variables x et y
x := 1	affectation de 1 à x
[x + y]	un bloc qui évalue x+y
<primitive : 1>	une primitive de la VM ¹ ou annotation
3 factorial	un message unaire
3 + 4	un message binaire
2 raisedTo : 6 modulo : 10	un message à mots-clés
↑ true	retourne la valeur true pour vrai
Transcript show : 'bonjour'. Transcript cr	un séparateur d'expression (.)
Transcript show : 'bonjour' ; cr	un message en cascade (;)

TABLE 3.1 – Résumé de la syntaxe de Pharo

Dans la table 3.1, nous pouvons voir des exemples divers d'éléments syntaxiques.

Les variables locales. startPoint est un nom de variable ou identifiant. Par convention, les identifiants sont composés de mots au format d'écriture casse chameau ("camelCase") : chaque mot excepté le premier débute par une lettre majuscule. La première lettre d'une variable d'instance, d'une méthode ou d'un bloc argument ou d'une variable temporaire doit être en minuscule. Ce qui indique au lecteur que la portée de la variable est privée .

Les variables partagées. Les identifiants qui débutent par une lettre majuscule sont des variables globales, des variables de classes, des diction-

naires de pool ou des noms de classes. Transcript est une variable globale, une instance de la classe TranscriptStream.

Le receveur. self est un mot-clé qui pointe vers l'objet sur lequel la méthode courante s'exécute. Nous le nommons "le receveur" car cet objet devra normalement recevoir le message qui provoque l'exécution de la méthode. self est appelé une "pseudo-variable" puisque nous ne pouvons rien lui affecter.

Les entiers. En plus des entiers décimaux habituels comme 42, Pharo propose aussi une notation en base numérique ou *radix*. 2r101 est 101 en base 2 (c-à-d. en binaire), qui est égal à l'entier décimal 5.

Les nombres flottants. Ils peuvent être spécifiés avec leur exposant en base dix : 2.4e7 est 2.4×10^7 .

Les caractères. Un signe dollar définit un caractère : \$a est le littéral pour 'a'. Des instances de caractères non-imprimables peuvent être obtenues en envoyant des messages ad hoc à la classe Character, tel que Character space et Character tab.

Les chaînes de caractères. Les apostrophes sont utilisées pour définir un littéral chaîne. Si vous désirez qu'une chaîne comporte une apostrophe, il suffira de doubler l'apostrophe, comme dans 'aujourd"'hui'.

Les symboles. Ils ressemblent à des chaînes de caractères, en ce sens qu'ils comportent une suite de caractères. Mais contrairement à une chaîne, un symbole doit être globalement unique. Il y a seulement un objet symbole #Bonjour mais il peut y avoir plusieurs objets chaînes de caractères ayant la valeur 'Bonjour'.

Les tableaux définis à la compilation. Ils sont définis par #(), les objets littéraux sont séparés par des espaces. À l'intérieur des parenthèses, tout doit être constant durant la compilation. Par exemple, #(27 (true false) abc)² est un tableau littéral de trois éléments : l'entier 27, le tableau à la compilation contenant deux booléens et le symbole #abc.

Les tableaux définis à l'exécution. Les accolades {} définissent un tableau (dynamique) à l'exécution. Ses éléments sont des expressions séparées par des points. Ainsi { 1. 2. 1+2 } définit un tableau dont les éléments sont 1, 2 et le résultat de l'évaluation de 1+2 ([dans le monde de Smalltalk, la notation entre accolades est particulière aux dialectes Pharo et Squeak](#). Dans d'autres Smalltalks vous devez explicitement construire des tableaux dynamiques).

Les commentaires. Ils sont encadrés par des guillemets. "Bonjour le commentaire" est un commentaire et non une chaîne; donc il est ignoré par le compilateur de Pharo. Les commentaires peuvent se répartir sur plusieurs lignes.

2. Notez que c'est la même chose que #(27 #(true false) #abc).

Les définitions des variables locales. Des barres verticales | | limitent les déclarations d'une ou plusieurs variables locales dans une méthode (ainsi que dans un bloc).

L'affectation. := affecte un objet à une variable.

Les blocs. Des crochets [] définissent un bloc, aussi connu sous le nom de *block closure* ou fermeture lexicale, laquelle est un objet à part entière représentant une fonction. Comme nous le verrons, les blocs peuvent avoir des arguments et des variables locales.

Les primitives. <primitive: ...> marque l'invocation d'une primitive de la VM ou machine virtuelle (<primitive: 1> est la primitive de SmallInteger»+). Tout code suivant la primitive est exécuté seulement si la primitive échoue. La même syntaxe est aussi employée pour des annotations de méthode.

Les messages unaires. Ce sont des simples mots (comme factorial) envoyés à un receveur (comme 3).

Les messages binaires. Ce sont des opérateurs (comme +) envoyés à un receveur et ayant un seul argument. Dans 3+4, le receveur est 3 et l'argument est 4.

Les messages à mots-clés. Ce sont des mots-clés multiples (comme raisedTo:modulo:), chacun se terminant par un deux-points (:) et ayant un seul argument. Dans l'expression 2 raisedTo: 6 modulo: 10, le *sélecteur de message* raisedTo:modulo: prend les deux arguments 6 et 10, chacun suivant le :. Nous envoyons le message au receveur 2.

Le retour d'une méthode. ↑ est employé pour obtenir le *retour* ou *renvoi* d'une méthode. Il vous faut taper ^ pour obtenir le caractère ↑.

Les séquences d'instructions. Un point (.) est le *séparateur d'instructions*. Placer un point entre deux expressions les transforme en deux instructions indépendantes.

Les cascades. un point virgule peut être utilisé pour envoyer une *cascade* de messages à un receveur unique. Dans Transcript show: 'bonjour'; cr, nous envoyons d'abord le message à mots-clés show: 'bonjour' au receveur Transcript, puis nous envoyons au même receveur le message unaire cr.

Les classes Number, Character, String et Boolean sont décrites avec plus de détails dans le chapitre 8.

3.2 Les pseudo-variables

Dans Smalltalk, il y a 6 mots-clés réservés ou *pseudo-variables* : nil, true , false, self, super et thisContext. Ils sont appelés pseudo-variables car ils sont prédéfinis et ne peuvent pas être l'objet d'une affectation. true, false et nil sont

des constantes tandis que les valeurs de self, super et de thisContext varient de façon dynamique lorsque le code est exécuté.

true et false sont les uniques instances des classes Boolean : True et False (voir le chapitre 8 pour plus de détails).

self se réfère toujours au receveur de la méthode en cours d'exécution. super se réfère aussi au receveur de la méthode en cours, mais quand vous envoyez un message à super, la recherche de méthode change en démarrant de la super-classe relative à la classe contenant la méthode qui utilise super (pour plus de détails, voyez le chapitre 5).

nil est l'objet non défini. C'est l'unique instance de la classe UndefinedObject. Les variables d'instance, les variables de classe et les variables locales sont initialisées à nil.

thisContext est une pseudo-variable qui représente la structure du sommet de la pile d'exécution. En d'autres termes, il représente le MethodContext ou le BlockClosure en cours d'exécution. En temps normal, thisContext ne doit pas intéresser la plupart des programmeurs, mais il est essentiel pour implémenter des outils de développement tels que le débogueur et il est aussi utilisé pour gérer exceptions et continuations.

3.3 Les envois de messages

Il y a trois types de messages dans Pharo.

1. Les messages *unaires* : messages sans argument. 1 factorial envoie le message factorial à l'objet 1.
2. Les messages *binaires* : messages avec un seul argument. 1 + 2 envoie le message + avec l'argument 2 à l'objet 1.
3. Les messages à *mots-clés* : messages qui comportent un nombre arbitraire d'arguments. 2 raisedTo: 6 modulo: 10 envoie le message comprenant le sélecteur raisedTo:modulo: et les arguments 6 et 10 vers l'objet 2.

Les sélecteurs des messages unaires sont constitués de caractères alphanumériques et débutent par une lettre minuscule.

Les sélecteurs des messages binaires sont constitués par un ou plusieurs caractères de l'ensemble suivant :

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

Les sélecteurs des messages à mots-clés sont formés d'une suite de mots-clés alphanumériques qui commencent par une lettre minuscule et se terminent par :.

Les messages unaires ont la plus haute priorité, puis viennent les messages binaires et, pour finir, les messages à mots-clés ; ainsi :

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

D'abord nous envoyons factorial à 3, puis nous envoyons + 6 à 1, et pour finir, nous envoyons raisedTo: 7 à 2. Rappelons que nous utilisons la notation *expression* → *result* pour montrer le résultat de l'évaluation d'une expression.

Priorité mise à part, l'évaluation s'effectue strictement de la gauche vers la droite, donc :

```
1 + 2 * 3 → 9
```

et non 7. Les parenthèses permettent de modifier l'ordre d'une évaluation :

```
1 + (2 * 3) → 7
```

Les envois de message peuvent être composés grâce à des points et des points-virgules. Une suite d'expressions séparées par des points provoque l'évaluation de chaque expression dans la suite comme une *instruction*, une après l'autre.

Transcript cr.

Transcript show: 'Bonjour le monde'.

Transcript cr

Ce code enverra cr à l'objet Transcript, puis enverra show: 'Bonjour le monde', et enfin enverra un nouveau cr.

Quand une succession de messages doit être envoyée à un *même* receveur, ou pour dire les choses plus succinctement en *cascade*, le receveur est spécifié une seule fois et la suite des messages est séparée par des points-virgules :

```
Transcript cr;
show: 'Bonjour le monde';
cr
```

Ce code a précisément le même effet que celui de l'exemple précédent.

3.4 Syntaxe relative aux méthodes

Bien que les expressions peuvent être évaluées n'importe où dans Pharo (par exemple, dans un espace de travail (Workspace), dans un débogueur (Debugger) ou dans un navigateur de classes), les méthodes sont en principe définies dans une fenêtre du Browser ou du débogueur les méthodes

peuvent aussi être rentrées depuis une source externe, mais ce n'est pas une façon habituelle de programmer en Pharo).

Les programmes sont développés, une méthode à la fois, dans l'environnement d'une classe précise (une classe est définie en envoyant un message à une classe existante, en demandant de créer une sous-classe, de sorte qu'il n'y ait pas de syntaxe spécifique pour créer une classe).

Voilà la méthode `lineCount` (pour compter le nombre de lignes) dans la classe `String`. La convention habituelle consiste à se référer aux méthodes comme suit : `ClassName»methodName` ; ainsi nous nommerons cette méthode `String»lineCount`³.

Méthode 3.1 – Compteur de lignes

```
String»lineCount
"Answer the number of lines represented by the receiver,
where every cr adds one line."
| cr count |
cr := Character cr.
count := 1 min: self size.
self do:
[:c | c == cr ifTrue: [count := count + 1]].
^ count
```

Sur le plan de la syntaxe, une méthode comporte :

1. la structure de la méthode avec le nom (*c-à-d.* `lineCount`) et tous les arguments (aucun dans cet exemple) ;
2. les commentaires (qui peuvent être placés n'importe où, mais conventionnellement, un commentaire doit être placé au début afin d'expliquer le but de la méthode) ;
3. les déclarations des variables locales (*c-à-d.* `cr` et `count`) ;
4. un nombre quelconque d'expressions séparées par des points ; dans notre exemple, il y en a **quatre**.

L'évaluation de n'importe quelle expression précédée par un `↑` (saisi en tapant `^`) provoquera l'arrêt de la méthode à cet endroit, donnant en retour la valeur de cette expression. Une méthode qui se termine sans retourner explicitement une expression retournera de façon implicite `self`.

Les arguments et les variables locales doivent toujours débuter par une lettre minuscule. Les noms débutant par une majuscule sont réservés aux variables globales. Les noms des classes, comme par exemple `Character`, sont tout simplement des variables globales qui se réfèrent à l'objet représentant cette classe.

3. Le commentaire de la méthode dit : "Retourne le nombre de lignes représentées par le receveur, dans lequel chaque cr ajoute une ligne"

3.5 La syntaxe des blocs

Les blocs apportent un moyen de différer l'évaluation d'une expression. Un bloc est essentiellement une fonction anonyme. Un bloc est évalué en lui envoyant le message `value`. Le bloc retourne la valeur de la dernière expression de son corps, à moins qu'il y ait un retour explicite (avec `↑`) auquel cas il ne retourne aucune valeur.

```
[ 1 + 2 ] value → 3
```

Les blocs peuvent prendre des paramètres, chacun doit être déclaré en le précédent d'un deux-points. Une barre verticale sépare les déclarations des paramètres et le corps du bloc. Pour évaluer un bloc avec un paramètre, vous devez lui envoyer le message `value:` avec un argument. Un bloc à deux paramètres doit recevoir `value:value:;` et ainsi de suite, jusqu'à 4 arguments.

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

Si vous avez un bloc comportant plus de quatre paramètres, vous devez utiliser `valueWithArguments:` et passer les arguments à l'aide d'un tableau (un bloc comportant un grand nombre de paramètres étant souvent révélateur d'un problème au niveau de sa conception).

Des blocs peuvent aussi déclarer des variables locales, lesquelles seront entourées par des barres verticales, tout comme des déclarations de variables locales dans une méthode. Les variables locales sont déclarées après les éventuels arguments :

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

Les blocs sont en fait des *fermetures lexicales*, puisqu'ils peuvent faire référence à des variables de leur environnement immédiat. Le bloc suivant fait référence à la variable `x` voisine :

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

Les blocs sont des instances de la classe `BlockClosure` ; ce sont donc des objets, de sorte qu'ils peuvent être affectés à des variables et être passés comme arguments à l'instar de tout autre objet.

3.6 Conditions et itérations

Smalltalk n'offre aucune syntaxe spécifique pour les structures de contrôle. Typiquement celles-ci sont obtenues par l'envoi de messages à des booléens, des nombres ou des collections, avec pour arguments des blocs.

Les clauses conditionnelles sont obtenues par l'envoi des messages `ifTrue:`, `ifFalse:` ou `ifTrue:ifFalse:` au résultat d'une expression booléenne. Pour plus de détails sur les booléens, lisez le chapitre 8.

```
(17 * 13 > 220)
ifTrue: [ 'plus grand'
ifFalse: [ 'plus petit' ] → 'plus grand'
```

Les boucles (ou itérations) sont obtenues typiquement par l'envoi de messages à des blocs, des entiers ou des collections. Comme la condition de sortie d'une boucle peut être évaluée de façon répétitive, elle se présentera sous la forme d'un bloc plutôt que de celle d'une valeur booléenne. Voici précisément un exemple d'une boucle procédurale :

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

`whileFalse:` inverse la condition de sortie.

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

`timesRepeat:` offre un moyen simple pour implémenter un nombre donné d'itérations :

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

Nous pouvons aussi envoyer le message `to:do:` à un nombre qui deviendra alors la valeur initiale d'un compteur de boucle. Le premier argument est la borne supérieure ; le second est un bloc qui prend la valeur courante du compteur de boucle comme argument :

```
n := 0.
1 to: 10 do: [ :counter | n := n + counter ].
n → 55
```

Itérateurs d'ordre supérieur. Les collections comprennent un grand nombre de classes différentes dont beaucoup acceptent le même protocole. Les messages les plus importants pour itérer sur des collections sont `do:`, `collect:`, `select:`, `reject:`, `detect:` ainsi que `inject:into:`. Ces messages définissent des itérateurs d'ordre supérieur qui nous permettent d'écrire du code très compact.

Une instance `Interval` (*c-à-d.* un intervalle) est une collection qui définit un itérateur sur une suite de nombres depuis un début jusqu'à une fin. `1 to: 10` représente l'intervalle de 1 à 10. Comme il s'agit d'une collection, nous pouvons lui envoyer le message `do:`. L'argument est un bloc qui est évalué pour chaque élément de la collection.

```
n := 0.
(1 to: 10) do: [ :element | n := n + element ].
n → 55
```

`collect:` construit une nouvelle collection de la même taille, en transformant chaque élément.

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

`select:` et `reject:` construisent des collections nouvelles, contenant un sous-ensemble d'éléments satisfaisant (ou non) la condition du bloc booléen. `detect:` retourne le premier élément satisfaisant la condition. Ne perdez pas de vue que les chaînes sont aussi des collections, ainsi vous pouvez itérer aussi sur tous les caractères. La méthode `isVowel` renvoie `true` (*c-à-d.* vrai) lorsque le receveur-caractère est une voyelle⁴.

```
'bonjour Squeak' select: [ :char | char isVowel ] → 'ooouea'
'bonjour Squeak' reject: [ :char | char isVowel ] → 'bnjr Sqk'
'bonjour Squeak' detect: [ :char | char isVowel ] → $o
```

4. Note du traducteur : les voyelles accentuées ne sont pas considérées par défaut comme des voyelles ; Smalltalk-80 a le même défaut que la plupart des langages de programmation nés dans la culture anglo-saxonne.

Finalement, vous devez garder à l'esprit que les collections acceptent aussi l'équivalent de l'opérateur `fold` issu de la programmation fonctionnelle au travers de la méthode `inject:into:`. Cela vous amène à générer un résultat cumulatif utilisant une expression qui accepte une valeur initiale puis injecte chaque élément de la collection. Les sommes et les produits sont des exemples typiques.

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

Ce code est équivalent à $0+1+2+3+4+5+6+7+8+9+10$.

Pour plus de détails sur les collections et les flux de données, rendez-vous dans les chapitres 9 et 10.

3.7 Primitives et Pragmas

En Smalltalk, tout est objet et tout se passe par l'envoi de messages. Néanmoins, à certains niveaux, ce modèle a ses limites ; le fonctionnement de certains objets ne peut être achevé qu'en invoquant la machine virtuelle et les primitives.

Par exemple, les comportements suivants sont tous implémentés sous la forme de primitives : l'allocation de la mémoire (`new` et `new:`), la manipulation de bits (`bitAnd:`, `bitOr:` et `bitShift:`), l'arithmétique des pointeurs et des entiers (+, -, <, >, *, /, =, ==...) et l'accès des tableaux (`at:`, `at:put:`).

Les primitives sont invoquées avec la syntaxe `<primitive: aNumber>` (`aNumber` étant un nombre). Une méthode qui invoque une telle primitive peut aussi embarquer du code Smalltalk qui sera évalué *seulement* en cas d'échec de la primitive.

Examinons le code pour `SmallInteger»+`. Si la primitive échoue, l'expression `super + aNumber` sera évaluée et retournée⁵.

Méthode 3.2 – Une méthode primitive

```
+ aNumber
```

"Primitive. Add the receiver to the argument and answer with the result if it is a SmallInteger. Fail if the argument or the result is not a SmallInteger. Essential. No Lookup. See Object documentation whatIsAPrimitive."

```
<primitive: 1>
↑ super + aNumber
```

5. Le commentaire de la méthode dit : "Ajoute le receveur à l'argument et répond le résultat s'il s'agit d'un entier de classe `SmallInteger`. Échoue si l'argument ou le résultat n'est pas un `SmallInteger`. Essentiel Aucune recherche. Voir la documentation de la classe `Object` : `whatIsAPrimitive` (qu'est-ce qu'une primitive)."

Dans Pharo, la syntaxe avec <....> est aussi utilisée pour les annotations de méthode que l'on appelle des *pragmas*.

3.8 Résumé du chapitre

- Pharo a (seulement) six mots réservés aussi appelés *pseudo-variables* : true, false, nil, self, super et thisContext.
- Il y a cinq types d'objets littéraux : les nombres (5, 2.5, 1.9e15, 2r111), les caractères (\$a), les chaînes ('bonjour'), les symboles (#bonjour) et les tableaux (#('bonjour' #bonjour))
- Les chaînes sont délimitées par des apostrophes et les commentaires par des guillemets. Pour obtenir une apostrophe dans une chaîne, il suffit de la doubler.
- Contrairement aux chaînes, les symboles sont par essence globalement uniques.
- Employez #(...) pour définir un tableau littéral. Employez { ... } pour définir un tableau dynamique. Sachez que #(1 + 2) size → 3, mais que { 1 + 2 } size → 1
- Il y a trois types de messages :
 - *unaire* : par ex., 1 asString, Array new ;
 - *binaire* : par ex., 3 + 4, 'salut' , 'Squeak' ;
 - à *mots-clés* : par ex., 'salue' at: 5 put: \$t
- Un envoi de messages en cascade est une suite de messages envoyés à la même cible, tous séparés par des ; : OrderedCollection new add: #albert; add: #einstein; size → 2
- Les variables locales sont déclarées à l'aide de barres verticales. Employez := pour les affectations. |x| x:=1
- Les expressions sont les messages envoyés, les cascades et les affectations ; parfois regroupées avec des parenthèses. Les *instructions* sont des expressions séparées par des points.
- Les blocs ou fermetures lexicales sont des expressions limitées par des crochets. Les blocs peuvent prendre des arguments et peuvent contenir des variables locales dites aussi *variables temporaires*. Les expressions du bloc ne sont évaluées que lorsque vous envoyez un message de la forme value... avec le bon nombre d'arguments.
[:x | x + 2] value: 4 → 6.
- Il n'y a pas de syntaxe particulière pour les structures de contrôle ; ce ne sont que des messages qui, sous certaines conditions, évaluent des blocs.
(Smalltalk includes: Class) ifTrue: [Transcript show: Class superclass]

Chapitre 4

Comprendre la syntaxe des messages

Bien que la syntaxe des messages Smalltalk soit extrêmement simple, elle n'est pas habituelle et cela peut prendre un certain temps pour s'y habituer. Ce chapitre offre quelques conseils pour vous aider à mieux appréhender la syntaxe spéciale des envois de messages. Si vous vous sentez en confiance avec la syntaxe, vous pouvez choisir de sauter ce chapitre ou bien d'y revenir un peu plus tard.

4.1 Identifier les messages

En Smalltalk, exception faite des éléments syntaxiques rencontrés dans le chapitre 3 (`(:= ↑ . ; # () {} [: |])`), tout se passe par envoi de messages. Comme en C++, vous pouvez définir vos opérateurs comme `+` pour vos propres classes, mais tous les opérateurs ont la même précédence. De plus, il n'est pas possible de changer l'arité d'une méthode : `-` est toujours un message binaire, et il n'est pas possible d'avoir une forme unaire avec une surcharge différente.

Avec Smalltalk, l'ordre dans lequel les messages sont envoyés est déterminé par le type de message. Il n'y a que trois formes de messages : les messages *unaire*, *binaire* et à *mots-clés*. Les messages unaires sont toujours envoyés en premier, puis les messages binaires et enfin ceux à mots-clés. Comme dans la plupart des langages, les parenthèses peuvent être utilisées pour changer l'ordre d'évaluation. Ces règles rendent le code Smalltalk aussi facile à lire que possible. La plupart du temps, il n'est pas nécessaire de réfléchir à ces règles.

Comme la plupart des calculs en Smalltalk sont effectués par des envois de messages, identifier correctement les messages est crucial. La terminolo-

gie suivante va nous être utile :

- Un message est composé d'un sélecteur et d'arguments optionnels.
- Un message est envoyé au receveur.
- La combinaison d'un message et de son receveur est appelé un *envoi de message* comme il est montré dans la figure 4.1.

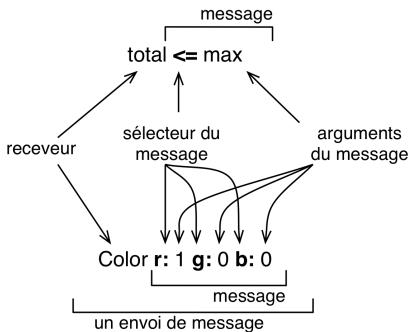


FIGURE 4.1 – Deux messages composés d'un receveur, d'un sélecteur de méthode et d'un ensemble d'arguments.

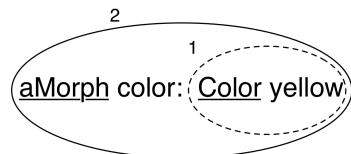


FIGURE 4.2 – aMorph color: Color yellow est composé de deux expressions : Color yellow et aMorph color: Color yellow.

Un message est toujours envoyé à un receveur qui peut être un simple littéral, une variable ou le résultat de l'évaluation d'une autre expression.

Nous vous proposons de vous faciliter la lecture au moyen d'une notation graphique : nous soulignerons le receveur afin de vous aider à l'identifier. Nous entourerons également chaque expression dans une ellipse et numérotterons les expressions à partir de la première à être évaluée afin de voir l'ordre d'envoi des messages.

La figure 4.2 représente deux envois de messages, Color yellow et aMorph color: Color yellow, de telle sorte qu'il y a deux ellipses. L'expression Color yellow est d'abord évalué en premier, ainsi son ellipse est numérotée à 1. Il y a deux receveurs : aMorph qui reçoit le message color: ... et Color qui reçoit le message yellow (yellow correspond à la couleur jaune en anglais). Chacun des receveurs est souligné.

Un receveur peut être le premier élément d'un message, comme 100 dans l'expression 100 + 200 ou Color (la classe des couleurs) dans l'expression Color yellow. Un objet receveur peut également être le résultat de l'évaluation d'autres messages. Par exemple, dans le message Pen new go: 100, le receveur de ce message go: 100 (littéralement, aller à 100) est l'objet retourné par cette expression Pen new (soit une instance de Pen, la classe crayon). Dans tous les

Expression	Type de messages	Résultat
Color yellow aPen go : 100	unaire à mots-clés	Crée une couleur. Le crayon receveur se déplace en avant de 100 pixels.
100 + 20	binaire	Le nombre 100 reçoit le message + avec le paramètre 20.
Browser open	unaire	Ouvre un nouveau navigateur de classes.
Pen new go : 100	unaire et à mots-clés	Un crayon est créé puis déplacé de 100 pixels.
aPen go : 100 + 20	à mots-clés et binaire	Le crayon receveur se déplace vers l'avant de 120 pixels.

TABLE 4.1 – Exemples de messages

cas, le message est envoyé à un objet appelé le *receveur* qui a pu être créé par un autre envoi de message.

La table 4.1 montre différents exemples de messages. Vous devez remarquer que tous les messages n'ont pas obligatoirement d'arguments. Un message unaire comme `open` (pour ouvrir) ne nécessite pas d'arguments. Les messages à mots-clés simples ou les messages binaires comme `go: 100` et `+ 20` ont chacun un argument. Il y a aussi des messages simples et des messages composés. `Color yellow` et `100 + 20` sont simples : un message est envoyé à un objet, tandis que l'expression `aPen go: 100 + 20` est composée de deux messages : `+ 20` est envoyé à `100` et `go:` est envoyé à `aPen` avec pour argument le résultat du premier message. Un receveur peut être une expression qui peut retourner un objet. Dans `Pen new go: 100`, le message `go: 100` est envoyé à l'objet qui résulte de l'évaluation de l'expression `Pen new`.

4.2 Trois sortes de messages

Smalltalk définit quelques règles simples pour déterminer l'ordre dans lequel les messages sont envoyés. Ces règles sont basées sur la distinction établie entre les 3 formes d'envoi de messages :

- *Les messages unaires* sont des messages qui sont envoyés à un objet sans autre information. Par exemple dans `3 factorial`, `factorial` (pour factorielle) est un message unaire.
- *Les messages binaires* sont des messages formés avec des opérateurs (souvent arithmétiques). Ils sont binaires car ils ne concernent que deux objets : le receveur et l'objet argument. Par exemple, dans `10 + 20`, `+` est un message binaire qui est envoyé au receveur `10` avec l'argument `20`.

- Les messages à mots-clés sont des messages formés avec plusieurs mots-clés, chacun d'entre eux se finissant par deux points (:) et prenant un paramètre. Par exemple, dans anArray at: 1 put: 10, le mot-clé at: prend un argument 1 et le mot-clé put: prend l'argument 10.

Messages unaires

Les messages unaires sont des messages qui ne nécessitent aucun argument. Ils suivent le modèle syntaxique suivant : receveur nomMessage. Le sélecteur est constitué d'une série de caractères ne contenant pas de deux points (:) (par ex., 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 "La classe de Object est Object class (!)"

Les messages unaires sont des messages qui ne nécessitent pas d'argument.

Ils suivent le moule syntaxique : receveur **sélecteur**

Messages binaires

Les messages binaires sont des messages qui nécessitent exactement un argument *et* dont le sélecteur consiste en une séquence de un ou plusieurs caractères de l'ensemble : +, -, *, /, &, =, >, |, <, ~, et @. Notez que -- n'est pas autorisé.

100@100	→	100@100 "crée un objet Point"
3 + 4	→	7
10 - 1	→	9
4 <= 3	→	false
(4/3) * 3 = 4	→	true "l'égalité est juste un message binaire et les fractions sont exactes"
(3/4) == (3/4)	→	false "deux fractions égales ne sont pas le même objet"

Les messages binaires sont des messages qui nécessitent exactement un argument *et* dont le sélecteur est composé d'une séquence de caractères parmi : +, -, *, /, &, =, >, |, <, ~, et @. -- n'est pas possible.

Ils suivent le moule syntaxique : receveur **sélecteur argument**

Messages à mots-clés

Les messages à mots-clés sont des messages qui nécessitent un ou plusieurs arguments et dont le sélecteur consiste en un ou plusieurs mots-clés se finissant par deux points ::. Les messages à mots-clés suivent le moule syntaxique : **receveur selecteurMotUn** : argumentUn **motDeux** : argumentDeux

Chaque mot-clé utilise un argument. Ainsi r:g:b: est une méthode avec 3 arguments, playFileNamed: et at: sont des méthodes avec un argument, et at:put: est une méthode avec deux arguments. Pour créer une instance de la classe Color on peut utiliser la méthode r:g:b: comme dans Color r: 1 g: 0 b: 0 créant ainsi la couleur rouge. Notez que les deux points ne font pas partie du sélecteur.

En Java ou C++, l'invocation de méthode Smalltalk Color r: 1 g: 0 b: 0 serait écrite Color.rgb(1,0,0).

```
1 to: 10          → (1 to: 10) "création d'un intervalle"
Color r: 1 g: 0 b: 0   → Color red "création d'une nouvelle
couleur (rouge)"
12 between: 8 and: 15 → true

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

Les messages basés sur les mots-clés sont des messages qui nécessitent un ou plusieurs arguments. Leurs sélecteurs consistent en un ou plusieurs mots-clés chacun se terminant par deux points (:). Ils suivent le moule syntaxique :

receveur selecteurMotUn : argumentUn **motDeux** : argumentDeux

4.3 Composition de messages

Les trois formes d'envoi de messages ont chacune des priorités différentes, ce qui permet de les composer de manière élégante.

1. Les messages unaires sont envoyés en premier, puis les messages binaires et enfin les messages à mots-clés.

2. Les messages entre parenthèses sont envoyés avant tout autre type de messages.
3. Les messages de même type sont envoyés de gauche à droite.

Ces règles ont un ordre de lecture très naturel. Maintenant si vous voulez être sûr que vos messages sont envoyés dans l'ordre que vous souhaitez, vous pouvez toujours mettre des parenthèses supplémentaires comme dans la figure 4.3. Dans cet exemple, le message `yellow` est un message unaire et le message `color:` est un message à mots-clés ; ainsi l'expression `Color yellow` est envoyé en premier. Néanmoins comme les expressions entre parenthèses sont envoyées en premier, mettre des parenthèses (normalement inutiles) autour de `Color yellow` permet d'accentuer le fait qu'elle doit être envoyée en premier. Le reste de cette section illustre chacun de ces différents points.

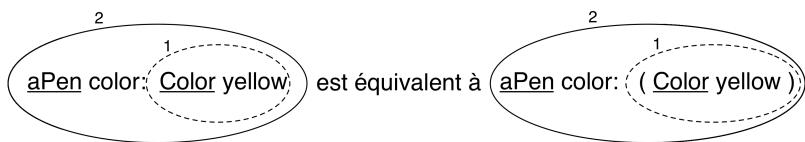


FIGURE 4.3 – Les messages unaires sont envoyés en premier ; donc ici le premier message est `Color yellow`. Il retourne un objet de couleur jaune qui est passé comme argument du message `aPen color:`.

Unaire > Binaire > Mots-clés

Les messages unaires sont d'abord envoyés, puis les messages binaires et enfin les messages à mots-clés. Nous pouvons également dire que les messages unaires ont une priorité plus importante que les autres types de messages.

Règle une. Les messages unaires sont envoyés en premier, puis les messages binaires et finalement les messages à mots-clés.

Unaire > Binaire > Mots-clés

Comme ces exemples suivants le montrent, les règles de syntaxe de Smalltalk permettent d'assurer une certaine lisibilité des expressions :

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

Malheureusement, les règles sont un peu trop simplistes pour les expressions arithmétiques. Dès lors, des parenthèses doivent être introduites

chaque fois que l'on veut imposer un ordre de priorité entre deux opérateurs binaires :

$$\begin{array}{lcl} 1 + 2 * 3 & \longrightarrow & 9 \\ 1 + (2 * 3) & \longrightarrow & 7 \end{array}$$

L'exemple suivant qui est un peu plus complexe (!) est l'illustration que même des expressions Smalltalk compliquées peuvent être lues de manière assez naturelle :

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

Ici nous voulons savoir quelles méthodes de la classe Boolean (classe des booléens) sont abstraites. Nous interrogeons la classe argument aClass pour récupérer les clés (via le message *unaire* keys) de son dictionnaire de méthodes (via le message *unaire* methodDict), puis nous en sélectionnons (via le message à mots-clés select:) les méthodes de la classe qui sont abstraites. Ensuite nous lions (par value:) l'argument aClass à la valeur concrète Boolean. Nous avons besoin des parenthèses uniquement pour le message binaire >>, qui sélectionne une méthode d'une classe, avant d'envoyer le message *unaire* isAbstract à cette méthode. Le résultat (sous la forme d'un ensemble de classe IdentitySet) nous montre quelles méthodes doivent être implémentées par les sous-classes concrètes de Boolean : True et False.

Exemple. Dans le message aPen color: Color yellow, il y a un message *unaire* yellow envoyé à la classe Color et un message à *mots-clés* color: envoyé à aPen. Les messages unaires sont d'abord envoyés, de telle sorte que l'expression Color yellow soit d'abord exécutée (1). Celle-ci retourne un objet couleur qui est passé en argument du message aPen color: aColor (2) comme indiqué dans l'exemple 4.1. La figure 4.3 montre graphiquement comment les messages sont envoyés.

Exemple 4.1 – Décomposition de l'évaluation de aPen color: Color yellow

(1)	aPen color: Color yellow	
	Color yellow	"message unaire envoyé en premier"
	→ aColor	
(2)	aPen color: aColor	"puis le message à mots-clés"

Exemple. Dans le message aPen go: 100 + 20, il y a le message *binaire* + 20 et un message à *mots-clés* go:. Les messages binaires sont d'abord envoyés avant les messages à mots-clés, ainsi 100 + 20 est envoyé en premier (1) : le message + 20 est envoyé à l'objet 100 et retourne le nombre 120. Ensuite le

message aPen go: 120 est envoyé avec comme argument 120 (2). L'exemple 4.2 nous montre comment l'expression est évaluée.

Exemple 4.2 – Décomposition de aPen go: 100 + 20

aPen go: 100 + 20		
(1)	100 + 20 → 120	"le message binaire en premier"
(2) aPen go: 120		"puis le message à mots-clés"

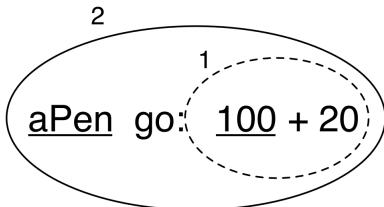


FIGURE 4.4 – Les messages unaires sont envoyés en premier, ainsi Color yellow est d'abord envoyé. Il retourne un objet de couleur jaune qui est passé en argument du message aPen color:..

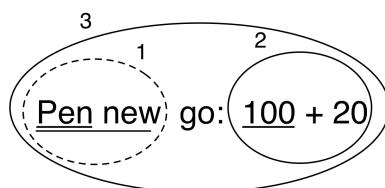


FIGURE 4.5 – Décomposition de Pen new go: 100 + 20.

Exemple. Comme exercice, nous vous laissons décomposer l'évaluation du message Pen new go: 100 + 20 qui est composé d'un message unaire, d'un message à mots-clés et d'un message binaire (voir la figure 4.5).

Les parenthèses en premier

Règle deux. Les messages parenthésés sont envoyés avant tout autre message.

(Msg) > Unaire > Binaire > Mots-clés

1.5 tan rounded asString = (((1.5 tan) rounded) asString) → true "les parenthèses sont nécessaires ici"
3 + 4 factorial → 27 "(et pas 5040)"
(3 + 4) factorial → 5040

Ici nous avons besoin des parenthèses pour forcer l'envoi de lowMajorScaleOn: avant play.

```
(FMSound lowMajorScaleOn: FMSound clarinet) play
"(1) envoie le message clarinet à la classe FMSound pour créer le son de clarinette.
(2) envoie le son à FMSound comme argument du message à mots-clés
    lowMajorScaleOn:
(3) joue le son résultant."
```

Exemple. Le message (65@325 extent: 134@100) center retourne le centre du rectangle dont le point supérieur gauche est (65, 325) et dont la taille est 134×100 . L'exemple 4.3 montre comment le message est décomposé et envoyé. Le message entre parenthèses est d'abord envoyé : il contient deux messages binaires 65@325 et 134@100 qui sont d'abord envoyés et qui retournent des points, et un message à mots-clés extent: qui est ensuite envoyé et qui retourne un rectangle. Finalement le message unaire center est envoyé au rectangle et le point central est retourné.

Évaluer ce message sans parenthèses déclencherait une erreur car l'objet 100 ne comprend pas le message center.

Exemple 4.3 – Exemple avec des parenthèses.

(65@325 extent: 134@100) center		
(1)	65@325	"binaire"
	→ aPoint	
(2)	134@100	"binaire"
	→ anotherPoint	
(3)	aPoint extent: anotherPoint	"à mots-clés"
	→ aRectangle	
(4)	aRectangle center	"unaire"
	→ 132@375	

De gauche à droite

Maintenant nous savons comment les messages de différentes natures ou priorités sont traités. Il reste une question à traiter : comment les messages de même priorité sont envoyés ? Ils sont envoyés de gauche à droite. Notez que vous avez déjà vu ce comportement dans l'exemple 4.3 dans lequel les deux messages de création de points (@) sont envoyés en premier.

Règle trois. Lorsque les messages sont de même nature, l'ordre d'évaluation est de gauche à droite.

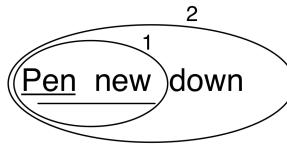


FIGURE 4.6 – Décomposition de Pen new down.

Exemple. Dans l’expression Pen new down, tous les messages sont des messages unaires, donc celui qui est le plus à gauche Pen new est envoyé en premier. Il retourne un nouveau crayon auquel le deuxième message down (pour poser la pointe du crayon et dessiner) est envoyé comme il est montré dans la figure 4.6.

Incohérences arithmétiques

Les règles de composition des messages sont simples mais peuvent engendrer des incohérences dans l’évaluation des expressions arithmétiques qui sont exprimées sous forme de messages binaires (nous parlons aussi d’irrationnalité arithmétique). Voici des situations habituelles où des parenthèses supplémentaires sont nécessaires.

$3 + 4 * 5 \longrightarrow 35$ "(pas 23) les messages binaires sont envoyés de gauche à droite"

$3 + (4 * 5) \longrightarrow 23$

$1 + 1/3 \longrightarrow (2/3)$ "et pas 4/3"

$1 + (1/3) \longrightarrow (4/3)$

$1/3 + 2/3 \longrightarrow (7/9)$ "et pas 1"

$(1/3) + (2/3) \longrightarrow 1$

Exemple. Dans l’expression $20 + 2 * 5$, il y a seulement les messages binaires + et *. En Smalltalk, il n’y a pas de priorité spécifique pour les opérations + et *. Ce ne sont que des messages binaires, ainsi * n’a pas priorité sur +. Ici le message le plus à gauche + est envoyé en premier (1) et ensuite * est envoyé au résultat comme nous le voyons dans l’exemple 4.4.

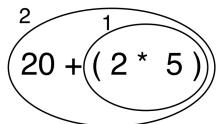
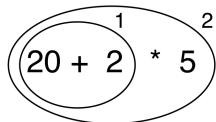
Exemple 4.4 – Décomposer $20 + 2 * 5$

"Comme il n'y a pas de priorité entre les messages binaires, le message le plus à gauche, + est évalué en premier même si d'après les règles de l'arithmétique le * devrait d'abord être envoyé."

$20 + 2 * 5$

(1) $20 + 2 \longrightarrow 22$

(2) $22 * 5 \longrightarrow 110$



Comme il est montré dans l'exemple 4.4 le résultat de cette expression n'est pas 30 mais 110. Ce résultat est peut-être inattendu mais résulte directement des règles utilisées pour envoyer des messages. Ceci est le prix à payer pour la simplicité du modèle de Smalltalk. Afin d'avoir un résultat correct, nous devons utiliser des parenthèses. Lorsque les messages sont entourés par des parenthèses, ils sont évalués en premier. Ainsi l'expression $20 + (2 * 5)$ retourne le résultat comme nous le voyons dans l'exemple 4.5.

Exemple 4.5 – Décomposition de $20 + (2 * 5)$

*"Les messages entourés de parenthèses sont évalués en premier ainsi * est envoyé avant + afin de produire le comportement souhaité."*

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

En Smalltalk, les opérateurs arithmétiques comme + et * n'ont pas des priorités différentes. + et * ne sont que des messages binaires ; donc * n'a pas priorité sur +. Utiliser des parenthèses pour obtenir le résultat désiré.

Notez que la première règle, disant que les messages unaires sont envoyés avant les messages binaires ou à mots-clés, ne nous force pas à mettre explicitement des parenthèses autour d'eux. La table 4.8 montre des expressions écrites en respectant les règles et les expressions équivalentes si les règles n'existaient pas. Les deux versions engendrent le même effet et retournent les mêmes valeurs.

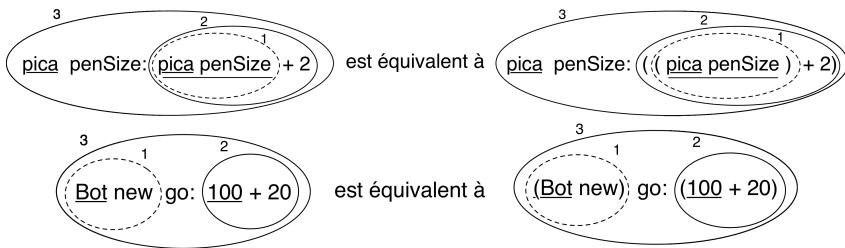


FIGURE 4.7 – Messages équivalents en utilisant des parenthèses.

Priorité implicite	Équivalent explicite parenthésé
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 – Des expressions et leurs versions équivalentes complètement parenthésées.

4.4 Quelques astuces pour identifier les messages à mots-clés

Souvent les débutants ont des problèmes pour comprendre quand ils doivent ajouter des parenthèses. Voyons comment les messages à mots-clés sont reconnus par le compilateur.

Des parenthèses ou pas ?

Les caractères [,], and (,) délimitent des zones distinctes. Dans ces zones, un message à mots-clés est la plus longue séquence de mots terminés par (:) qui n'est pas coupé par les caractères (.), ou (;). Lorsque les caractères [,], et (,) entourent des mots avec des deux points, ces mots participent au message à mots-clés *local* à la zone définie.

Dans cet exemple, il y a deux mots-clés distincts : rotatedBy:magnify:smoothing: et at:put:.

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

put: 3

Les caractères [,], et (,) délimitent des zones distinctes. Dans ces zones, un message à mots-clés est la plus longue séquence de mots qui se termine par (:) qui n'est pas coupé par les caractères (.), ou ;. Lorsque les caractères [,], et (,) entourent des mots avec des deux points, ces mots participent au message à mots-clés local à cette zone.

ASTUCE Si vous avez des problèmes avec ces règles de priorité, vous pouvez commencer simplement en entourant avec des parenthèses chaque fois que vous voulez distinguer deux messages avec la même priorité.

L'expression qui suit ne nécessite pas de parenthèses car l'expression `x isNil` est unaire donc envoyée avant le message à mots-clés `ifTrue :`

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

L'expression qui suit nécessite des parenthèses car les messages `includes:` et `ifTrue:` sont chacun des messages à mots-clés.

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

Sans les parenthèses le message inconnu `includes:ifTrue:` serait envoyé à la collection !

Quand utiliser les [] ou les () ?

Vous pouvez avoir des difficultés à comprendre quand utiliser des crochets plutôt que des parenthèses. Le principe de base est que vous devez utiliser des [] lorsque vous ne savez pas combien de fois une expression peut être évaluée (peut-être même jamais). [expression] va créer une fermeture lexicale ou bloc (c-à-d. un objet) à partir de `expression`, qui peut être évaluée autant de fois qu'il le faut (voire jamais) en fonction du contexte.

Ainsi les clauses conditionnelles de `ifTrue:` ou `ifTrue:ifFalse:` nécessitent des blocs. Suivant le même principe, à la fois le receveur et l'argument du message `whileTrue:` nécessitent l'utilisation des crochets car nous ne savons pas combien de fois le receveur ou l'argument seront exécutés.

Les parenthèses quant à elles n'affectent que l'ordre d'envoi des messages. Aucun objet n'est créé, ainsi dans (`expression`), `expression` sera toujours

évalué exactement une fois (en supposant que le code englobant l'expression soit évalué une fois).

[x isReady] whileTrue: [y doSomething]	"à la fois le receveur et l'argument doivent être des blocs"
4 timesRepeat: [Beeper beep]	"l'argument est évalué plus d'une fois, donc doit être un bloc"
(x isReady) ifTrue: [y doSomething]	"le receveur est évalué qu'une fois, donc n'est pas un bloc"

4.5 Séquences d'expression

Les expressions (c-à-d. envois de message, affectations...) séparées par des points sont évaluées en séquence. Notez qu'il n'y a pas de point entre la définition d'un variable et l'expression qui suit. La valeur d'une séquence est la valeur de la dernière expression. Les valeurs retournées par toutes les expressions exceptée la dernière sont ignorées. Notez que le point est un séparateur et non un terminateur d'expression. Le point final est donc optionnel.

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

4.6 Cascades de messages

Smalltalk offre la possibilité d'envoyer plusieurs messages au même receveur en utilisant le point-virgule (:). Dans le jargon Smalltalk, nous parlons de *cascade*.

Expression Msg1 ; Msg2

Transcript show: 'Pharo est'.
Transcript show: 'extra'.
Transcript cr.

est équivalent à :

Transcript
show: 'Pharo est';
show: 'extra';
cr

Notez que l'objet qui reçoit la cascade de messages peut également être le résultat d'un envoi de message. En fait, le receveur de la cascade est le receveur du premier message de la cascade. Dans l'exemple qui suit, le premier message en cascade est setX:setY puisqu'il est suivi du point-virgule.

Le receveur du message cascadé `setX:setY:` est le nouveau point résultant de l'évaluation de `Point new`, et *non pas Point*. Le message qui suit `isZero` (pour tester s'il s'agit de zéro) est envoyé au même receveur.

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

4.7 Résumé du chapitre

- Un message est toujours envoyé à un objet nommé le *receveur* qui peut être le résultat d'autres envois de messages.
- Les messages unaires sont des messages qui ne nécessitent pas d'arguments.
Ils sont de la forme **receveur sélecteur**.
- Les messages binaires sont des messages qui concernent deux objets, le receveur et un autre objet et dont le sélecteur est composé de un ou deux caractères de la liste suivante : `+, -, *, /, |, &, =, >, <, ^, et @`.
Ils sont de la forme : **receveur sélecteurMotUn : argumentUn motDeux : argumentDeux**.
- **Règle un.** Les messages unaires sont d'abord envoyés, puis les messages binaires et finalement les messages à mots-clés.
- **Règle deux.** Les messages entre parenthèses sont envoyés avant tous les autres.
- **Règle trois.** Lorsque les messages sont de même nature, l'ordre d'évaluation est de gauche à droite.
- En Smalltalk, les opérateurs arithmétiques traditionnels comme `+` ou `*` ont la même priorité. `+` et `*` ne sont que des messages binaires ; donc `*` n'a aucune priorité sur `+`. Vous devez utiliser les parenthèses pour obtenir un résultat différent.

Deuxième partie

Développer avec Pharo

Chapitre 5

Le modèle objet de Smalltalk

Le modèle de programmation de Smalltalk est simple et homogène : tout est objet et les objets communiquent les uns avec les autres uniquement via l'envoi de messages. Cependant, ces caractéristiques de simplicité et d'homogénéité peuvent être source de quelques difficultés pour le programmeur habitué à d'autres langages. Dans ce chapitre nous présenterons les concepts de base du modèle objet de Smalltalk ; en particulier nous discuterons des conséquences de représenter les classes comme des objets.

5.1 Les règles du modèle

Le modèle objet de Smalltalk repose sur un ensemble de règles simples qui sont appliquées de manière *uniforme*. Les règles s'énoncent comme suit :

Règle 1. Tout est objet.

Règle 2. Tout objet est instance de classe.

Règle 3. Toute classe a une super-classe.

Règle 4. Tout se passe par envoi de messages.

Règle 5. La recherche des méthodes suit la chaîne de l'héritage.

Prenons le temps d'étudier ces règles en détail.

5.2 Tout est objet

Le mantra “tout est objet” est hautement contagieux. Après seulement peu de temps passé avec Smalltalk, vous serez progressivement surpris par la façon dont cette règle simplifie tout ce que vous faites. Par exemple, les entiers sont véritablement des objets (de classe Integer). Dès lors vous pouvez leur envoyer des messages, comme vous le feriez avec n’importe quel autre objet.

3 + 4	→	7 "envoie '+ 4' à 3, donnant 7"
20 factorial	→	2432902008176640000 "envoie factorial, donnant un grand nombre"

La représentation de 20 factorial est certainement différente de la représentation de 7, mais aucune partie du code—pas même l’implémentation de factorial¹—n’a besoin de le savoir puisque ce sont des objets tous deux.

La conséquence fondamentale de cette règle pourrait s’énoncer ainsi :

Les classes sont aussi des objets.

Plus encore, les classes ne sont pas des objets de seconde zone : elles sont véritablement des objets de premier plan auxquels vous pouvez envoyer des messages, que vous pouvez inspecter, etc. Ainsi Pharo est vraiment un système réflexif offrant une grande expressivité aux développeurs.

En regardant plus avant dans l’implémentation de Smalltalk, nous trouvons trois sortes différentes d’objets. Il y a (1) les objets ordinaires avec des variables d’instance passées par référence ; il y a (2) *les petits entiers*² qui sont passés par valeur, et enfin, il y a (3) les objets indexés comme les Array (tableaux) qui ont une portion contigüe de mémoire. La beauté de Smalltalk réside dans le fait que vous n’avez aucunement à vous soucier des différences entre ces trois types d’objet.

5.3 Tout objet est instance de classe

Tout objet a une classe ; pour vous en assurer, vous pouvez envoyer à un objet le message class (classe en anglais).

1 class	→	SmallInteger
20 factorial class	→	LargePositiveInteger
'hello' class	→	ByteString

1. En anglais, factorielle.
2. En anglais, *small integers*.

#(1 2 3) class	→	Array
(4@5) class	→	Point
Object new class	→	Object

Une classe définit la *structure* pour ses instances via les variables d'instance (instance variables en anglais) et leur *comportement* (*behavior* en anglais) via les méthodes. Chaque méthode a un nom. C'est le *sélecteur*. Il est unique pour chaque classe.

Puisque *les classes sont des objets* et que *tout objet est une instance d'une classe*, nous en concluons que les classes doivent aussi être des instances de classes. Les classes dont les instances sont des classes sont nommées des *méta-classes*. à chaque fois que vous créez une classe, le système crée pour vous une métaclass automatiquement. La métaclass définit la structure et le comportement de la classe qui est son instance. 99% du temps vous n'aurez pas à penser aux métaclasses et vous pourrez joyeusement les ignorer. (Nous porterons notre attention aux métaclasses dans le chapitre 13.)

Les variables d'instance

Les variables d'instance en Smalltalk sont privées vis-à-vis de l'*instance* elle-même. Ceci diffère de langages comme Java et C++ qui permettent l'accès aux variables d'instance (aussi connues sous le nom d'"attributs" ou "variables membre") depuis n'importe quelle autre instance de la même classe. Nous disons que l'*espace d'encapsulation*³ des objets en Java et en C++ est la classe, là où, en Smalltalk, c'est l'instance.

En Smalltalk, deux instances d'une même classe ne peuvent pas accéder aux variables d'instance l'une de l'autre à moins que la classe ne définisse des "méthodes d'accès" (en anglais, accessor methods). Aucun élément de la syntaxe ne permet l'accès direct à la variable d'instances de n'importe quel autre objet. (En fait, un mécanisme appelé réflexivité offre une véritable possibilité d'interroger un autre objet sur la valeur de ses variables d'instance ; la métaprogrammation permet d'écrire des outils tel que l'inspecteur d'objets (nous utiliserons aussi le terme Inspector). La seule vocation de ce dernier est de regarder le contenu des autres objets.)

Les variables d'instance peuvent être accédées par nom dans toutes les méthodes d'instance de la classe qui les définit ainsi que dans les méthodes définies dans les sous-classes de cette classe. Cela signifie que les variables d'instance en Smalltalk sont semblables aux variables protégées (protected) en C++ et en Java. Cependant, nous préférerons dire qu'elles sont privées parce qu'il n'est pas d'usage en Smalltalk d'accéder à une variable d'instance directement depuis une sous-classe.

3. En anglais, encapsulation boundary.

Exemple

La méthode `Point»dist:` (méthode 5.1) calcule la distance entre le receveur et un autre point. Les variables d'instance `x` et `y` du receveur sont accédées directement par le corps de la méthode. Cependant, les variables d'instance de l'autre point doivent être accédées en lui envoyant les messages `x` et `y`.

Méthode 5.1 – *la distance entre deux points. le nom arbitraire aPoint est utilisé dans le sens de a point qui, en anglais, signifie un point*

`Point»dist: aPoint`

"Retourne la distance entre aPoint et le receveur."

```
| dx dy |
dx := aPoint x - x.
dy := aPoint y - y.
↑ ((dx * dx) + (dy * dy)) sqrt
```

1@1 dist: 4@5 → 5.0

La raison-clé de préférer l'encapsulation basée sur l'instance à l'encapsulation basée sur la classe tient au fait qu'elle permet à différentes implementations d'une même abstraction de coexister. Par exemple, la méthode `point»dist:` n'a besoin ni de surveiller, ni même de savoir si l'argument `aPoint` est une instance de la même classe que le receveur. L'argument objet pourrait être représenté par des coordonnées polaires, voire comme un enregistrement dans une base de données ou sur une autre machine d'un réseau distribué ; tant qu'il peut répondre aux messages `x` et `y`, le code de la méthode 5.1 fonctionnera toujours.

Les méthodes

Toutes les méthodes sont publiques⁴. Les méthodes sont regroupées en protocoles qui indiquent leur objectif. Certains noms de protocoles courants ont été attribués par convention, par exemple, *accessing* pour les méthodes d'accès, et *initialization* pour construire un état initial stable pour l'objet. Le protocole *private* est parfois utilisé pour réunir les méthodes qui ne devraient pas être visibles depuis l'extérieur. Rien ne vous empêche cependant d'envoyer un message qui est implémenté par une telle méthode "privée".

Les méthodes peuvent accéder à toutes les variables d'instance de l'objet. Certains programmeurs en Smalltalk préfèrent accéder aux variables d'instance uniquement au travers des méthodes d'accès. Cette pratique a un certain avantage, mais elle tend à rendre l'interface de vos classes chaotique, ou pire, à exposer des états privés à tous les regards.

4. En fait, presque toutes. En Pharo, des méthodes dont les sélecteurs commencent par la chaîne de caractères `pvt` sont privées : un message `pvt` ne peut être envoyé qu'à `self` *uniquement*. N'importe comment, les méthodes `pvt` sont très peu utilisées.

Le côté instance et le côté classe

Puisque les classes sont des objets, elles peuvent avoir leurs propres variables d'instance ainsi que leurs propres méthodes. Nous les appelons *variables d'instance de classe* (en anglais *class instance variables*) et *méthodes de classe*, mais elles ne sont véritablement pas différentes des variables et méthodes d'instances ordinaires : les variables d'instance de classe ne sont seulement que des variables d'instance définies par une métaclassse. Quant aux méthodes de classe, elles correspondent juste aux méthodes définies par une métaclassse.

Une classe et sa métaclassse sont deux classes distinctes, et ce, même si cette première est une instance de l'autre. Pour vous, tout ceci sera somme toute largement trivial : vous n'aurez qu'à vous concentrer sur la définition du comportement de vos objets et des classes qui les créent.

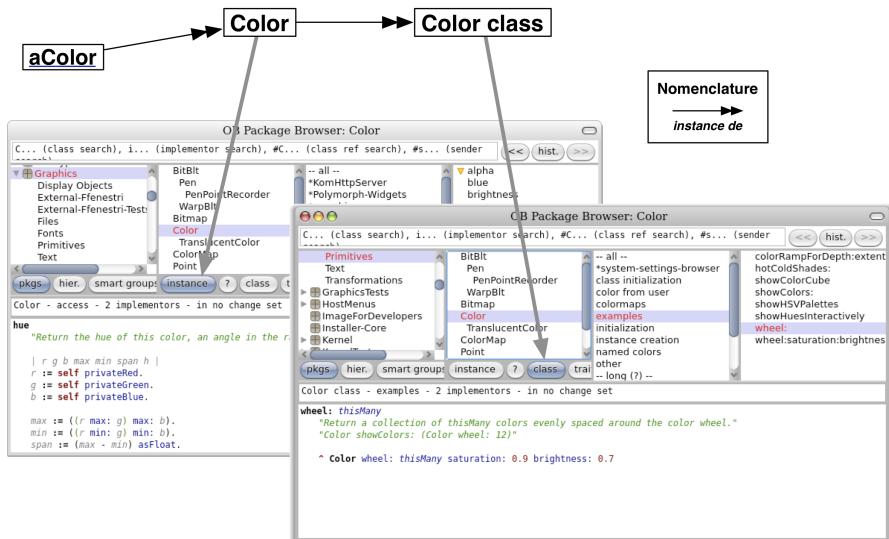


FIGURE 5.1 – Naviguer dans une classe et sa métaclassse.

De ce fait, le navigateur de classes nommé Browser vous aide à parcourir à la fois classes et métaclasses comme si elles n'étaient qu'une seule entité avec deux "côtés" : le "côté instance" et le "côté classe", comme le montre la la figure 5.1. En cliquant sur le bouton **instance**, vous voyez la présentation de la classe Color, *c-à-d.* vous pouvez naviguer dans les méthodes qui sont exécutées quand les messages sont envoyés à une instance de Color, comme la couleur blue (correspondant au bleu). En appuyant sur le bouton **class** (pour classe), vous naviguez dans la classe Color class, autrement dit vous voyez les méthodes qui seront exécutées en envoyant les messages directement à la

classe Color elle-même. Par exemple, Color blue envoie le message blue (pour bleu) à la classe Color. Vous trouverez donc la méthode blue définie côté classe de la classe Color et non du côté instance.

aColor := Color blue.	"Méthode de classe blue"
aColor	→ Color blue
aColor red	→ 0.0 "Méthode d'accès red (rouge) côté instance"
aColor blue	→ 1.0 "Méthode d'accès blue (bleu) côté instance"

Vous définissez une classe en remplissant le patron (ou *template* en anglais) proposé dans le côté instance. Quand vous acceptez ce patron, le système crée non seulement la classe que vous définissez mais aussi la métaclass correspondante. Vous pouvez naviguer dans la métaclass en cliquant sur le bouton **class**. Du patron employé pour la création de la métaclass, seule la liste des noms des variables d'instance vous est proposée pour une édition directe.

Une fois que vous avez créé une classe, cliquer sur le bouton **instance** vous permet d'éditer et de parcourir les méthodes qui seront possédées par les instances de cette classe (et de ses sous-classes). Par exemple, nous pouvons voir dans la figure 5.1 que la méthode hue est définie pour les instances de la classe Color. A contrario, le bouton **class** vous laisse parcourir et éditer la métaclass (dans ce cas Color class).

Les méthodes de classe

Les méthodes de classe peuvent être relativement utiles ; naviguez dans Color class pour voir quelques bons exemples. Vous verrez qu'il y a deux sortes de méthodes définies dans une classe : celles qui créent les instances de la classe, comme Color class»blue et celles qui ont une action *utilitaire*, comme Color class»showColorCube. Ceci est courant, bien que vous trouvez occasionnellement des méthodes de classe utilisées d'une autre manière.

Il est commun de placer des méthodes utilitaires dans le côté classe parce qu'elles peuvent être exécutées sans avoir à créer un objet additionnel dans un premier temps. En fait, beaucoup d'entre elles contiennent un commentaire pour les rendre plus compréhensibles pour l'utilisateur qui les exécute.

 Naviguez dans la méthode Color class»showColorCube, double-cliquez à l'intérieur des guillements englobant le commentaire "Color showColorCube" et tape au clavier CMD-d.

Vous verrez l'effet de l'exécution de cette méthode. (Sélectionnez World > restore display (r) pour annuler les effets.)

Pour les familiers de Java et C++, les méthodes de classe peuvent être assimilées aux méthodes statiques. Néanmoins, l'homogénéité de Smalltalk

induit une différence : les méthodes statiques de Java sont des fonctions résolues de manière statique alors que les méthodes de classe de Smalltalk sont des méthodes à transfert dynamique⁵ Ainsi, l'héritage, la surcharge et l'utilisation de *super* fonctionnent avec les méthodes de classe dans Smalltalk, ce qui n'est pas le cas avec les méthodes statiques en Java.

Les variables d'instance de classe

Dans le cadre des variables d'instance ordinaires, toutes les instances d'une classe partagent le même ensemble de noms de variable et les instances de ses sous-classes héritent de ces noms ; cependant, chaque instance possède son propre jeu de valeurs. C'est exactement la même histoire avec les variables d'instance de classe : chaque classe a ses propres variables d'instance de classe privées. Une sous-classe héritera de ces variables d'instance de classe, *mais elle aura ses propres copies privées de ces variables*. Aussi vrai que les objets ne partagent pas les variables d'instance, les classes et leurs sous-classes ne partagent pas les variables d'instance de classe.

Vous pouvez utiliser une variable d'instance de classe *count*⁶ afin de suivre le nombre d'instances que vous créez pour une classe donnée. Cependant, les sous-classes ont leur propre variable *count*, les instances des sous-classes seront comptées séparément.

Exemple : les variables d'instance de classe ne sont pas partagées avec les sous-classes. Soient les classes Dog et Hyena⁷ telles que Hyena hérite de la variable d'instance de classe *count* de la classe Dog.

Classe 5.2 – Créer Dog et Hyena

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

Dog class
  instanceVariableNames: 'count'
```

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

5. En anglais, dynamically-dispatched methods.

6. En français, compteur.

7. En français, chien et hyène.

Supposons que nous ayons des méthodes de classe de Dog pour initialiser sa variable count à 0 et pour incrémenter cette dernière quand de nouvelles instances sont créées :

Méthode 5.3 – Comptabiliser les nouvelles instances de Dog via count

```
Dog class»initialize
```

```
super initialize.
```

```
count := 0.
```

```
Dog class»new
```

```
count := count +1.
```

```
↑ super new
```

```
Dog class»count
```

```
↑ count
```

Maintenant, à chaque fois que nous créons un nouveau Dog, son compteur count est incrémenté. Il en est de même pour toute nouvelle instance de Hyena, mais elles sont comptées séparément :

```
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
```

Remarquons aussi que les variables d'instance de classe sont privées à la classe tout comme les variables d'instance sont privées à l'instance. Comme les classes et leurs instances sont des objets différents, il en résulte que :

Une classe n'a pas accès aux variables d'instance de ses propres instances.

Une instance d'une classe n'a pas accès aux variables d'instance de classe de sa classe.

C'est pour cette raison que les méthodes d'initialisation d'instance doivent toujours être définies dans le côté instance — le côté classe n'ayant pas accès aux variables d'instance, il ne pourrait y avoir initialisation ! Tout ce que peut

faire la classe, c'est d'envoyer des messages d'initialisation à des instances nouvellement créées ; ces messages pouvant bien sûr utiliser les méthodes d'accès.

De même, les instances ne peuvent accéder aux variables d'instance de classe que de manière indirecte en envoyant les messages d'accès à leur classe.

Java n'a rien d'équivalent aux variables d'instance de classe. Les variables statiques en Java et en C++ ont plutôt des similitudes avec les variables de classe de Smalltalk dont nous parlerons dans la section 5.7 : toutes les sous-classes et leurs instances partagent la même variable statique.

Exemple : Définir un Singleton. Le patron de conception⁸ nommé Singleton⁹ offre un exemple-type de l'usage de variables d'instance de classe et de méthodes de classe. Imaginez que nous souhaitions d'une part, créer une classe WebServer et d'autre part, s'assurer qu'il n'a qu'une et une seule instance en faisant appel au patron Singleton.

En cliquant sur le bouton `instance` dans le navigateur de classe, nous définissons la classe WebServer comme suit (classe 5.4).

Classe 5.4 – Un classe Singleton

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

Ensuite, en cliquant sur le bouton `class`, nous pouvons ajouter une variable d'instance `uniqueInstance` au côté classe.

Classe 5.5 – Le côté classe de la classe Singleton

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

Par conséquence, la classe WebServer a désormais un autre variable d'instance, en plus des variables héritées telles que `superclass` et `methodDict`.

Nous pouvons maintenant définir une méthode de classe que nous appellerons `uniqueInstance` comme dans la méthode 5.6. Pour commencer, cette méthode vérifie si `uniqueInstance` a été initialisée ou non : dans ce dernier cas, la méthode crée une instance et l'assigne à la variable d'instance de classe `uniqueInstance`. *In fine*, la valeur de `uniqueInstance` est retournée. Puisque

8. En anglais, nous parlons de *Design Patterns*.

9. Sherman R. Alpert, Kyle Brown et Bobby Woolf, *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998, ISBN 0-201-18462-1.

`uniqueInstance` est une variable d'instance de classe, cette méthode peut directement y accéder.

Méthode 5.6 – `uniqueInstance` (côté classe)

```
WebServer class»uniqueInstance
uniqueInstance ifNil: [uniqueInstance := self new].
^ uniqueInstance
```

La première fois que l'expression `WebServer uniqueInstance` est exécutée, une instance de la classe `WebServer` sera créée et affectée à la variable `uniqueInstance`. La seconde fois, l'instance précédemment créée sera retournée au lieu d'y avoir une nouvelle création.

Remarquons que la clause conditionnelle à l'intérieur du code de création de la méthode 5.6 est écrite `self new` et non `WebServer new`. Quelle en est la différence ? Comme la méthode `uniqueInstance` est définie dans `WebServer class`, vous pouvez penser qu'elles sont identiques. En fait, tant que personne ne crée une sous-classe de `WebServer`, elles sont pareilles. Mais en supposant que `ReliableWebServer` est une sous-classe de `WebServer` et qu'elle hérite de la méthode `uniqueInstance`, nous devrions nous attendre à ce que `ReliableWebServer uniqueInstance` réponde un `ReliableWebServer`. L'utilisation de `self` assure que cela arrivera car il sera lié à la classe correspondante. Du reste, notez que `WebServer` et `ReliableWebServer` ont chacune leur propre variable d'instance de classe nommée `uniqueInstance`.

Ces deux variables ont, bien entendu, différentes valeurs.

5.4 Toute classe a une super-classe

Chaque classe en Smalltalk hérite de son comportement et de la description de sa structure d'une unique *super-classe*. Ceci est équivalent à dire que Smalltalk a un héritage simple.

<code>SmallInteger superclass</code>	→	<code>Integer</code>
<code>Integer superclass</code>	→	<code>Number</code>
<code>Number superclass</code>	→	<code>Magnitude</code>
<code>Magnitude superclass</code>	→	<code>Object</code>
<code>Object superclass</code>	→	<code>ProtoObject</code>
<code>ProtoObject superclass</code>	→	<code>nil</code>

Traditionnellement, la racine de la hiérarchie d'héritage en Smalltalk est la classe `Object` ("Objet" en anglais ; puisque tout est objet). En Pharo, la racine est en fait une classe nommée `ProtoObject`, mais normalement, vous n'aurez aucune attention à accorder à cette classe. `ProtoObject` encapsule le jeu de messages restreint que tout objet *doit* avoir. N'importe comment, la plupart des

classes héritent de Object qui, pour sa part, définit beaucoup de messages supplémentaires que presque tous les objets devraient comprendre et auxquels ils devraient pouvoir répondre. à moins que vous ayez une autre raison de faire autrement, vous devriez normalement générer des classes d'application par l'héritage de la classe Object ou d'une de ses sous-classes lors de la création de classe.

 *Une nouvelle classe est normalement créée par l'envoi du message subclass: instanceVariableNames: ... à une classe existante. Il y a d'autres méthodes pour créer des classes. Veuillez jeter un coup d'œil au protocole Kernel-Classes ▷ Class ▷ subclass creation pour voir desquelles il s'agit.*

Bien que Pharo ne dispose pas d'héritage multiple, il dispose d'un mécanisme appelé *traits*¹⁰ pour partager le comportement entre des classes distincts. Les *traits* sont des collections de méthodes qui peuvent être réutilisées par plusieurs classes sans lien d'héritage. Employer les *traits* vous permet de partager du code entre les différentes classes sans reproduire ce code.

Les méthodes abstraites et les classes abstraites

Une classe abstraite est une classe qui n'existe que pour être héritée, au lieu d'être instanciée. Une classe abstraite est habituellement incomplète, dans le sens qu'elle ne définit pas toutes les méthodes qu'elle utilise. Les méthodes "manquantes"— celle que les autres méthodes envoyent, mais qui ne sont pas définies elles-mêmes — sont dites méthodes abstraites.

Smalltalk n'a pas de syntaxe dédiée pour dire qu'une méthode ou qu'une classe est abstraite. Par convention, le corps d'une méthode abstraite contient l'expression `self subclassResponsibility`¹¹. Ceci est connu sous le nom de "marker method" ou marqueur de méthode ; il indique que les sous-classes ont la responsabilité de définir une version concrète de la méthode. Les méthodes `self subclassResponsibility` devraient toujours être surchargées, et ainsi, ne devraient jamais être exécutées. Si vous oubliez d'en surcharger une et que celle-ci est exécutée, une exception sera levée.

Une classe est considérée comme abstraite si une de ses méthodes est abstraite. Rien ne vous empêche de créer une instance d'une classe abstraite ; tout fonctionnera jusqu'à ce qu'une méthode abstraite soit invoquée.

Exemple : la classe **Magnitude**.

Magnitude est une classe abstraite qui nous aide à définir des objets pouvant être comparables les uns avec les autres. Les sous-classes de Magnitude

10. Dans le sens de trait de caractères, nous faisons allusion ainsi à la génétique du comportement d'une méthode.

11. Dans le sens, laissée à la responsabilité de la sous-classe.

devraient implémenter les méthodes `<`, `=` et `hash`¹². Grâce à ces messages, `Magnitude` définit d'autres méthodes telles que `>`, `>=`, `<=`, `max:`, `min:`, `between:and:` et d'autres encore pour comparer des objets. Ces méthodes sont héritées par les sous-classes. La méthode `<` est abstraite et est définie comme dans la méthode 5.7.

Méthode 5.7 – `Magnitude»<`. Le commentaire dit : “répond si le receveur est inférieur à l’argument”

`Magnitude»< aMagnitude`

“Answer whether the receiver is less than the argument.”

↑self subclassResponsibility

A contrario, la méthode `>=` est concrète ; elle est définie en fonction de `<` :

Méthode 5.8 – `Magnitude»>=`. Le commentaire dit : “répond si le receveur est plus grand ou égal à l’argument”

`>= aMagnitude`

“Answer whether the receiver is greater than or equal to the argument.”

↑(self < aMagnitude) not

Il en va de même des autres méthodes de comparaison.

`Character` est une sous-classe de `Magnitude` ; elle surcharge la méthode `subclassResponsibility` de `<` avec sa propre version de `<` (voir méthode 5.9). `Character` définit aussi les méthodes `=` et `hash` ; elles héritent les méthodes `>=`, `<=`, `~=` et autres de la classe `Magnitude`.

Méthode 5.9 – `Character»<`. Le commentaire dit : “répond vrai si la valeur du receveur est inférieure à la valeur du l’argument”

`Character»< aCharacter`

“Answer true if the receiver's value < aCharacter's value.”

↑self asciiValue < aCharacter asciiValue

Traits

Un *trait* est une collection de méthodes qui peut être incluse dans le comportement d'une classe sans le besoin d'un héritage. Les classes disposent non seulement d'une seule super-classe mais aussi de la facilité offerte par le partage de méthodes utiles avec d'autres méthodes sans lien de parenté vis-à-vis de l'héritage.

Définir un nouveau *trait* se fait en remplaçant simplement le patron pour la création de la sous-classe par un message à la classe `Trait`.

12. Relatif au code de hachage.

Classe 5.10 – Définir un nouveau trait

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

Nous définissons ici le *trait* TAuthor dans la catégorie *PBE-LightsOut*. Ce *trait* n'utilise¹³ aucun autre *trait* existant. En général, nous pouvons spécifier l'*expression de composition d'un trait* par d'autres *traits* en utilisant le mot-clé uses:. Dans notre cas, nous écrivons un tableau vide ({}).

Les *traits* peuvent contenir des méthodes, mais aucune variable d'instance. Supposons que nous voulons ajouter une méthode author (auteur en anglais) à différentes classes sans lien hiérarchique ; nous le ferions ainsi :

Méthode 5.11 – Définir la méthode author

```
TAuthor>author
"Returns author initials"
↑ 'on'  "oscar nierstrasz"
```

Maintenant nous pouvons employer ce *trait* dans une classe ayant déjà sa propre super-classe, disons, la classe LOGame que nous avons définie dans le chapitre 2. Nous n'avons qu'à modifier le patron de création de la classe LOGame pour y inclure cette fois l'argument-clé uses: suivi du *trait* à utiliser : TAuthor.

Classe 5.12 – Utiliser un trait

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

Si nous instancions maintenant LOGame, l'instance répondra comme prévu au message author.

```
LOGame new author → 'on'
```

Les expressions de composition de *trait* peuvent combiner plusieurs *traits* via l'opérateur +. En cas de conflit (*c-à-d.* quand plusieurs *traits* définissent des méthodes avec le même nom), ces conflits peuvent être résolus en retirant explicitement ces méthodes (avec -) ou en redéfinissant ces méthodes dans la classe ou le *trait* que vous êtes en train de définir. Il est possible aussi de créer un alias des méthodes (avec @) leur fournissant ainsi un nouveau nom.

13. Terme anglais : uses : il signifie “utilise”.

Les *traits* sont employés dans le noyau du système¹⁴. Un bon exemple est la classe Behavior.

Classe 5.13 – Behavior définit par les traits

```
Object subclass: #Behavior
  uses: TPureBehavior @ {#basicAddTraitSelector:withMethod:->
    #addTraitSelector:withMethod:}
  instanceVariableNames: 'superclass methodDict format'
  classVariableNames: 'ObsoleteSubclasses'
  poolDictionaries: ''
  category: 'Kernel-Classes'
```

Ici, nous voyons que la méthode basicAddTraitSelector:withMethod: définie dans le trait TPureBehavior a été renommée en addTraitSelector:withMethod:. Les *traits* sont à présent supportés par les navigateurs de classe (ou *browsers*).

5.5 Tout se passe par envoi de messages

Cette règle résume l'essence même de la programmation en Smalltalk.

Dans la programmation procédurale, lorsqu'une procédure est appelée, l'appelant (*caller*, en anglais) fait le choix du morceau de code à exécuter ; il choisit la procédure ou la fonction à exécuter *statiquement*, par nom.

En programmation orientée objet, nous ne faisons pas d'"appel de méthodes". Nous faisons un "envoi de messages." Le choix de terminologie est important. Chaque objet a ses propres responsabilités. Nous ne pouvons dire à un objet ce qu'il faut faire en lui imposant une procédure. Au lieu de cela, nous devons lui demander poliment de faire quelque chose en lui envoyant un message. Le message n'est pas un morceau de code : ce n'est rien d'autre qu'un nom (sélecteur) et une liste d'arguments. Le receveur décide alors de comment y répondre en sélectionnant en retour sa propre méthode correspondant à ce qui a été demandé. Puisque des objets distincts peuvent avoir différentes méthodes pour répondre à un même message, le choix de la méthode doit se faire *dynamiquement* à la réception du message.

3 + 4	→	7	"envoie le message + d'argument 4 à l'entier 3"
(1@2) + 4	→	5@6	"envoie le message + d'argument 4 au point (1@2)"

En conséquence, nous pouvons envoyer le même message à différents objets, chacun pouvant avoir sa propre méthode en réponse au message. Nous ne disons pas à SmallInteger 3 ou au Point 1@2 comment répondre au message + 4. Chacun a sa propre méthode pour répondre à cet envoi de message, et répond ainsi selon le cas.

14. En anglais, System kernel.

L'une des conséquences du modèle d'envoi de messages de Smalltalk est qu'il encourage un style de programmation dans lequel les objets tendent à avoir des méthodes très compactes en déléguant des tâches aux autres objets, plutôt que d'implémenter de gigantesques méthodes procédurales engendrant trop de responsabilité. Joseph Pelrine dit succinctement le principe suivant :

Ne fais rien que tu ne peux déléguer à quelqu'un d'autre[†].

[†]. Don't do anything that you can push off onto someone else.

Beaucoup de langages orientés objets disposent à la fois d'opérations statiques et dynamiques pour les objets ; en Smalltalk il n'y a qu'envois de messages dynamiques. Au lieu de fournir des opérations statiques sur les classes, nous leur envoyons simplement des messages, puisque les classes sont aussi des objets.

Pratiquement tout en Smalltalk se passe par envoi de messages. à certains stades, le pragmatisme doit prendre le relais :

- Les *déclarations de variable* ne reposent pas sur l'envoi de messages. En fait, les déclarations de variable ne sont même pas exécutables. Déclarer une variable produit simplement l'allocation d'un espace pour la référence de l'objet.
- Les *affectations* (ou assignations) ne reposent pas sur l'envoi de messages. L'affectation d'une variable produit une liaison de nom de variable dans le cadre de sa définition.
- Les *retours* (ou renvois) ne reposent pas sur l'envoi de message. Un retour ne produit que le retour à l'envoyeur du résultat calculé.
- Les *primitives* ne reposent pas sur l'envoi de message. Elles sont codées au niveau de la machine virtuelle.

à quelques autres exceptions près, presque tout le reste se déroule véritablement par l'envoi de messages. En particulier, la seule façon de mettre à jour une variable d'instance d'un autre objet est de lui envoyer un message réclamant le changement de son propre attribut (ou champ) car ces derniers ne sont pas des "attributs publics" en Smalltalk. Bien entendu, offrir des méthodes d'accès dites accesseurs (*getter*, en anglais, retournant l'état de la variable) et mutateurs (*setter* en anglais, changeant la variable) pour chaque variable d'instance d'un objet n'est pas une bonne méthodologie orientée objet. Joseph Pelrine annonce aussi à juste titre :

Ne laissez jamais personne d'autre jouer avec vos données[†].

[†]. Don't let anyone else play with your data.

5.6 La recherche de méthode suit la chaîne d'héritage

Qu'arrive-t-il exactement quand un objet reçoit un message ?

Le processus est relativement simple : la classe du receveur cherche la méthode à utiliser pour opérer le message. Si cette classe n'a pas de méthode, elle demande à sa super-classe et remonte ainsi de suite la chaîne d'héritage. Quand la méthode est enfin trouvée, les arguments sont affectés aux paramètres de la méthode et la machine virtuelle l'exécute.

C'est, en essence, aussi simple que cela. Mais il reste quelques questions auxquelles nous devons prendre soin de répondre :

- *Que se passe-t-il lorsque une méthode ne renvoie pas explicitement une valeur ?*
- *Que se passe-t-il quand une classe réimplémente une méthode d'une super-classe ?*
- *Quelle différence y a-t-il entre les envois faits à self et ceux faits à super ?*
- *Que se passe-t-il lorsqu'aucune méthode est trouvée ?*

Les règles pour la recherche par référencement (en anglais *lookup*) présentées ici sont conceptuelles : des réalisations au sein de la machine virtuelle ruseront pour optimiser la vitesse de recherche des méthodes. C'est leur travail mais tout est fait pour que vous ne remarquiez jamais qu'elles font quelque chose de différent des règles énoncées.

Tout d'abord, penchons-nous sur la stratégie de base de la recherche. Ensuite nous répondrons aux questions.

La recherche de méthode

Supposons la création d'une instance de `EllipseMorph`.

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

Si nous envoyons à cet objet le message `defaultColor`, nous obtenons le résultat `Color yellow`¹⁵ :

```
anEllipse defaultColor → Color yellow
```

La classe `EllipseMorph` implémente `defaultColor`, donc la méthode adéquate est trouvée immédiatement.

Méthode 5.14 – Une méthode implementée localement. Le commentaire dit : “retourne la couleur par défaut ; le style de remplissage pour le receveur”

15. Yellow est la couleur jaune.

```

EllipseMorph»defaultColor
  "answer the default color/fill style for the receiver"
  ↑ Color yellow

```

A contrario, si nous envoyons le message `openInWorld` à `anEllipse`, la méthode n'est pas trouvée immédiatement parce que la classe `EllipseMorph` n'implémente pas `openInWorld`. La recherche continue plus avant dans la super-classe `BorderedMorph`, puis ainsi de suite, jusqu'à ce qu'une méthode `openInWorld` soit trouvée dans la classe `Morph` (voir la figure 5.2).

Méthode 5.15 – Une méthode héritée. Le commentaire dit : “Ajoute ce morph dans le monde (world).”

```

Morph»openInWorld
  "Add this morph to the world."
self openInWorld: self currentWorld

```

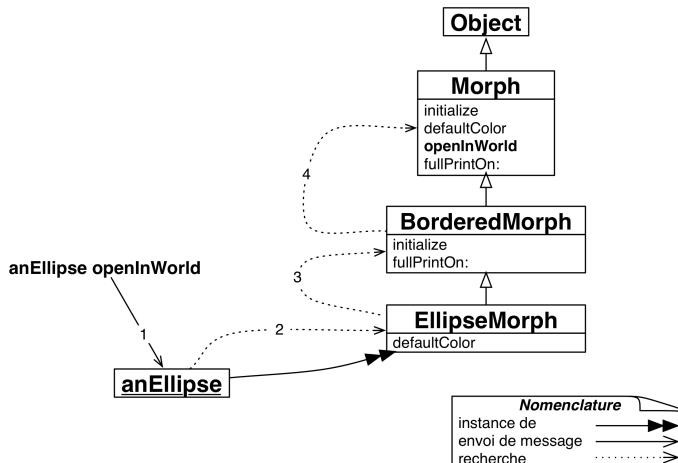


FIGURE 5.2 – Recherche par référencement d'une méthode suivant la hiérarchie d'héritage.

Renvoyer self

Remarquez que `EllipseMorph»defaultColor` (méthode 5.14) renvoie explicitement `Color yellow` alors que `Morph»openInWorld` (méthode 5.15) semble ne rien retourner.

En réalité une méthode renvoie *toujours* une valeur — qui est, bien entendu, un objet. La réponse peut être explicitement définie par l'utilisation

du symbole \uparrow dans la méthode. Si lors de l'exécution, on atteint la fin de la méthode sans avoir rencontré de \uparrow , la méthode retournera toujours une valeur : l'objet receveur lui-même. On dit habituellement que la méthode "renvoie self", parce qu'en Smalltalk la pseudo-variable `self` représente le receveur du message. En Java, on utilise le mot-clé `this`.

Ceci induit le constat suivant : la méthode 5.15 est équivalent à la méthode 5.16 :

Méthode 5.16 – *Renvoi explicite de self. Le dernier commentaire dit : "Ne faites pas cela à moins d'en être sûr"*

```
Morph»openInWorld
  "Add this morph to the world."
```

```
self openInWorld: self currentWorld.
↑ self   "Don't do this unless you mean it"
```

Pourquoi écrire \uparrow `self` explicitement n'est pas une bonne chose à faire ? Parce que, quand vous renvoyez explicitement quelque chose, vous communiquez que vous retournez quelque chose d'importance à l'expéditeur du message. Dès lors vous spécifiez que vous attendez que l'expéditeur fasse quelque chose de la valeur retournée. Puisque ce n'est pas le cas ici, il est préférable de ne pas renvoyer explicitement `self`.

C'est une convention en Smalltalk, ainsi résumé par Kent Beck se référant à la valeur de retour importante "Interesting return value"¹⁶ :

Renvoyez une valeur seulement quand votre objet expéditeur en a l'usage †.

†. Return a value only when you intend for the sender to use the value.

Surcharge et extension.

Si nous revenons à la hiérarchie de classe `EllipseMorph` dans la figure 5.2, nous voyons que les classes `Morph` et `EllipseMorph` implémentent toutes deux `defaultColor`. En fait, si nous ouvrons un nouvel élément graphique `Morph` (`Morph new openInWorld`), nous constatons que nous obtenons un morph bleu, là où l'ellipse (`EllipseMorph`) est jaune (yellow) par défaut.

Nous disons que `EllipseMorph` surcharge la méthode `defaultColor` qui hérite de `Morph`. La méthode héritée n'existe plus du point de vue `anEllipse`.

16. Kent Beck, *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

Parfois nous ne voulons pas surcharger les méthodes héritées, mais plutôt les étendre avec de nouvelles fonctionnalités ; autrement dit, nous souhaiterions pouvoir invoquer la méthode surchargée complétée par la nouvelle fonctionnalité que nous aurons définie dans la sous-classe. En Smalltalk, comme dans beaucoup de langages orientés objet reposant sur l'héritage simple, nous pouvons le faire à l'aide d'un envoi de message à super.

La méthode initialize est l'exemple le plus important de l'application de ce mécanisme. Quand une nouvelle instance d'une classe est initialisée, il est vital d'initialiser toutes les variables d'instance héritées. Cependant, les méthodes initialize de chacune des super-classes de la chaîne d'héritage fournissent déjà la connaissance nécessaire. La sous-classe n'a pas à s'occuper d'initialiser les variables d'instance héritées !

C'est une bonne pratique d'envoyer super initialize avant tout autre considération lorsque nous créons une méthode d'initialisation :

Méthode 5.17 – *Super initialize*. Le commentaire dit : “initialise l'état du receveur”

```
BorderedMorph>initialize
"initialize the state of the receiver"
super initialize.
self borderInitialize
```

Une méthode initialize devrait toujours commencer par la ligne super initialize.

Self et super

Nous avons besoin des envois sur super pour réutiliser le comportement hérité qui pourrait sinon être surchargé. Cependant, la technique habituelle de composition de méthodes, héritées ou non, est basée sur l'envoi à self.

Comment l'envoi à self diffère de celle avec super ? Comme self, super représente le receveur du message. La seule différence est dans la méthode de recherche. Au lieu de faire partir la recherche depuis la classe du receveur, celle-ci démarre dans la super-classe de la méthode dans laquelle l'envoi à super se produit.

Remarquez que super n'est pas la super-classe ! C'est une erreur courante et normale que de le penser. C'est aussi une erreur de penser que la recherche commence dans la super-classe du receveur. Nous allons voir précisément comment cela marche avec l'exemple suivant.

Considérons le message constructorString, que nous pouvons envoyer à n'importe quel morph :

```
anEllipse constructorString → '((EllipseMorph newBounds: (0@0 corner: 50@40)
color: Color yellow) setBorderWidth: 1 borderColor: Color black)'
```

La valeur de retour est une chaîne de caractères qui peut être évaluée pour recréer un morph.

Comment ce résultat est-il exactement obtenu grâce à l'association de self et de super ? Pour commencer, `anEllipse constructorString` trouvera la méthode `constructorString` dans la classe `Morph`, comme vu dans la figure 5.3.

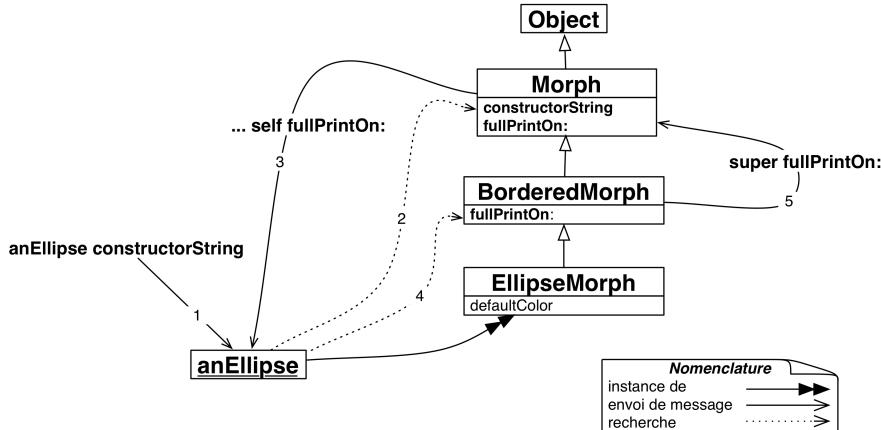


FIGURE 5.3 – Les envois à self et super.

Méthode 5.18 – Un envoi à self

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

La méthode `Morph»constructorString` envoie `printConstructorOn:indent:` à `self`. Ce message est recherché dans la hiérarchie en commençant dans la classe `EllipseMorph` et finalement trouvé dans `Morph`. Cette méthode en retour envoie à `self` le message `printConstructorOn :indent :nodeDict :`, qui, à son tour, envoie `fullPrintOn: à self`. Encore une fois, `fullPrintOn:` est recherché depuis la classe `EllipseMorph` et `fullPrintOn:` est trouvé dans `BorderedMorph` (revoir la figure 5.3)

Un envoi à self déclenche le départ de la recherche dynamique de méthode dans la classe du receveur.

Méthode 5.19 – Combiner l'usage de super et self

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

Maintenant, `BorderedMorph»fullPrintOn:` utilise l'envoi à `super` pour étendre le comportement `fullPrintOn:` hérité de sa super-classe. Parce qu'il s'agit d'un envoi à `super`, la recherche démarre alors depuis la super-classe de la classe dans laquelle se produit l'envoi à `super`, autrement dit, dans `Morph`. Nous trouvons ainsi immédiatement `Morph»fullPrintOn:` que nous évaluons.

Notez que la recherche sur `super` n'a pas commencé dans la super-classe du receveur. Ainsi il en aurait résulté un départ de la recherche depuis `BorderedMorph`, créant alors une boucle infinie !

Un envoi à `super` déclenche un départ de recherche *statische* de méthode dans la super-classe de la classe dont la méthode envoie le message à `super`.

Si vous regardez attentivement l'envoi à `super` et la figure 5.3, vous réaliserez que les liens à `super` sont statiques : tout ce qui importe est la classe dans laquelle le texte de l'envoi à `super` est trouvé. A contrario, le sens de `self` est dynamique : `self` représente toujours le receveur du message courant exécuté. Ce qui signifie que *tout* message envoyé à `self` est recherché en partant de la classe du receveur.

MessageNotUnderstood

Que se passe-t-il si la méthode que nous cherchons n'est pas trouvée ?

Supposons que nous envoyons le message `foo` à une ellipse `anEllipse`. Tout d'abord, la recherche normale de cette méthode aurait à parcourir toute la chaîne d'héritage jusqu'à la classe `Object` (ou plutôt `ProtoObject`). Comme cette méthode n'est pas trouvée, la machine virtuelle veillera à ce que l'object envoie `self doesNotUnderstand: #foo`. (voir la figure 5.4.)

Ceci est un envoi dynamique de message tout à fait normal. Ainsi la recherche recommence depuis la classe `EllipseMorph`, mais cette fois-ci en cherchant la méthode `doesNotUnderstand:`¹⁷. Il apparaît que `Object` implémente `doesNotUnderstand:..` Cette méthode créera un nouvel objet `MessageNotUnderstood` (en français : Message incompréhensible) capable de démarrer `Debugger`, le débogueur, dans le contexte actuel de l'exécution.

17. Le nom du message peut se traduire par : ne comprend pas.

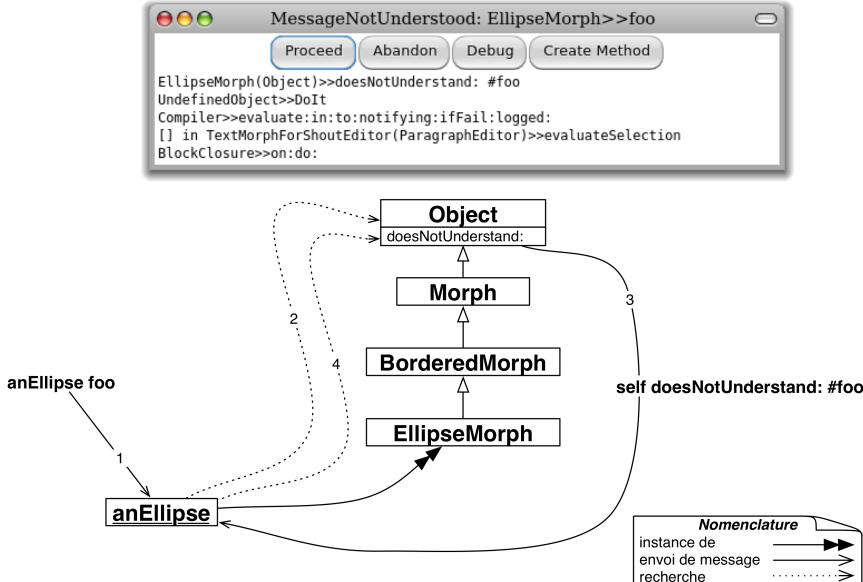


FIGURE 5.4 – Le message foo n'est pas compris (not understood).

Pourquoi prenons-nous ce chemin sinueux pour gérer une erreur si évidente ? Parce qu'en faisant ainsi, le développeur dispose de tous les outils pour agir alternativement grâce à l'interception de ces erreurs. N'importe qui peut surcharger la méthode `doesNotUnderstand:` dans une sous-classe de `Object` en étendant ses possibilités en offrant une façon différente de capturer l'erreur.

En fait, nous simplifions la vie en implémentant une délégation automatique de messages d'un objet à un autre. Un objet `Delegator` peut simplement déléguer tous les messages qu'il ne comprend pas à un autre objet dont la responsabilité est de les gérer ou de lever une erreur lui-même !

5.7 Les variables partagées

Maintenant, intéressons-nous à un aspect de Smalltalk que nous n'avons pas couvert par nos cinq règles : les variables partagées.

Smalltalk offre trois sortes de variables partagées : (1) les variables *globales* ; (2) les *variables de classe* partagées entre les instances et les classes, et (3) les variables partagées parmi un groupe de classes ou *variables de pool*. Les noms de toutes ces variables partagées commencent par une lettre capi-

tale (majuscule), pour nous informer qu'elles sont partagées entre plusieurs objets.

Les variables globales

En Pharo, toutes les variables globales sont stockées dans un espace de nommage appelé Smalltalk qui est une instance de la classe SystemDictionary. Les variables globales sont accessibles de partout. Toute classe est nommée par une variable globale ; en plus, quelques variables globales sont utilisées pour nommer des objets spéciaux ou couramment utilisés.

La variable Transcript nomme une instance de TranscriptStream, un flux de données ou *stream* qui écrit dans une fenêtre à ascenseur (dite aussi *scrollable*). Le code suivant affiche des informations dans le Transcript en passant une ligne.

```
Transcript show: 'Pharo est extra' ; cr
```

Avant de lancer la commande `do it`, ouvrez un Transcript en sélectionnant `World > Tools ... > Transcript`.

ASTUCE Écrire dans le Transcript est lent, surtout quand la fenêtre Transcript est ouverte. Ainsi, si vous constatez un manque de réactivité de votre système alors que vous êtes en train d'écrire dans le Transcript, pensez à le minimiser (bouton collapse this window).

D'autres variables globales utiles

- Smalltalk est une instance de SystemDictionary (Dictionnaire Système) définissant toutes les variables globales — dont l'objet Smalltalk lui-même. Les clés de ce dictionnaire sont des symboles nommant les objets globaux dans le code Smalltalk. Ainsi par exemple,

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

Puisque Smalltalk est aussi une variable globale lui-même,

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

et

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

- Sensor est une instance of EventSensor. Il représente les entrées interactives ou interfaces de saisie (en anglais, *input*) dans Pharo. Par exemple, Sensor keyboard retourne le caractère suivant saisi au clavier, et Sensor leftShiftDown répond true (vrai en booléen) si la touche *shift*

gauche est maintenue enfoncée, alors que Sensor mousePoint renvoie un Point indiquant la position actuelle de la souris.

- World (Monde en anglais) est une instance de PasteUpMorph représentant l'écran. World bounds retourne un rectangle définissant l'espace tout entier de l'écran ; tous les morphs (objet Morph) sur l'écran sont des sous-morphs ou *submorphs* de World.
- ActiveHand est l'instance courante de HandMorph, la représentation graphique du curseur. Les sous-morphs de ActiveHand tiennent tout ce qui est glissé par la souris.
- Undeclared¹⁸ est un autre dictionnaire — il contient toutes les variables non déclarées. Si vous écrivez une méthode qui référence une variable non déclarée, le navigateur de classe (Browser) vous l'annoncera normalement pour que vous la déclariez, par exemple, en tant que variable globale ou variable d'instance de la classe. Cependant, si par la suite, vous effacez la déclaration, le code référencera une variable non déclarée. Inspecter Undeclared peut parfois expliquer des comportements bizarres !
- SystemOrganization est une instance de SystemOrganizer : il enregistre l'organisation des classes en paquets. Plus précisément, il catégorise les noms des classes, ainsi

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

Une pratique courante est de limiter fortement l'usage des variables globales ; il est toujours préférable d'utiliser des variables d'instance de classe ou des variables de classes et de fournir des méthodes de classe pour y accéder. En effet, si aujourd'hui Pharo devait être reprogrammé à partir de zéro¹⁹, la plupart des variables globales qui ne sont pas des classes seraient remplacées par des Singletons.

La technique habituellement employée pour définir une variable globale est simplement de faire un `do it` sur une affectation d'un identifiant non déclaré commençant par une majuscule. Dès lors, l'analyseur syntaxique ou *parser* vous la déclarera en tant que variable globale. Si vous voulez en définir une de manière programmatique, exécutez Smalltalk at: `#AGlobalName put: nil`. Pour l'effacer, exécutez Smalltalk `removeKey: #AGlobalName`.

Les variables de classe

Nous avons besoin parfois de partager des données entre les instances d'une classe et la classe elle-même. C'est possible grâce aux *variables de classe*. Le terme variable de classe indique que le cycle de vie de la variable est le même que celui de la classe. Cependant, le terme ne véhicule pas l'idée

18. Non déclaré, en français.

19. Le terme anglais est : *from scratch*, signifiant depuis le début.

que ces variables sont partagées aussi bien parmi toutes les instances d'une classe que dans la classe elle-même comme nous pouvons le voir sur la figure 5.5. En fait, *variables partagées* (ou *shared variables*, en anglais) aurait été un meilleur nom car ce dernier exprime plus clairement leur rôle tout en pointant le danger de les utiliser, en particulier si elles sont sujettes aux modifications.

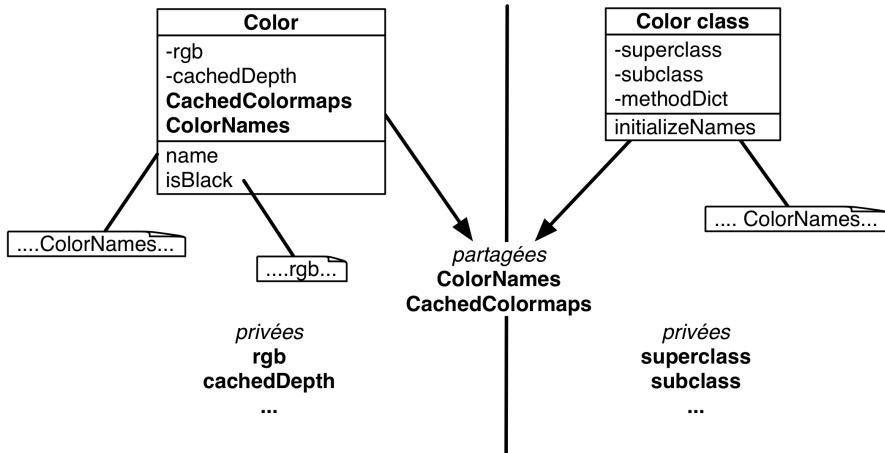


FIGURE 5.5 – Des méthodes d’instance et de classe accédant à différentes variables.

Sur la figure 5.5 nous voyons que `rgb` et `cachedDepth` sont des variables d’instance de `Color` uniquement accessibles par les instances de `Color`. Nous remarquons aussi que `superclass`, `subclass`, `methodDict`... etc, sont des variables d’instance de classe, *c-à-d.* des variables d’instance accessibles seulement par `Color class`.

Mais nous pouvons noter quelque chose de nouveau : `ColorNames` et `CachedColormaps` sont des *variables de classe* définies pour `Color`. La capitalisation du nom de ces variables nous donne un indice sur le fait qu’elles sont partagées. En fait, non seulement toutes les instances de `Color` peuvent accéder à ces variables partagées, mais aussi la classe `Color` elle-même, ainsi que *toutes ses sous-classes*. Les méthodes d’instance et de classe peuvent accéder toutes les deux à ces variables partagées.

Une variable de classe est déclarée dans le patron de définition de la classe. Par exemple, la classe `Color` définit un grand nombre de variables de classe pour accélérer la création des couleurs ; sa définition est visible ci-dessous (classe 5.20).

Classe 5.20 – Color et ces variables de classe

```
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'
```

La variable de classe `ColorNames` est un tableau contenant le nom des couleurs fréquemment utilisées. Ce tableau est partagé par toutes les instances de `Color` et de sa sous-classe `TranslucentColor`. Elles sont accessibles via les méthodes d'instance et de classe.

`ColorNames` est initialisée une fois dans `Color class»initializeNames`, mais elle est en libre accès depuis les instances de `Color`. La méthode `Color»name` utilise la variable pour trouver le nom de la couleur. Il semble en effet inopportun d'ajouter une variable d'instance `name` à chaque couleur car la plupart des couleurs n'ont pas de noms.

L'initialisation de classe

La présence de variables de classe soulève une question : comment les initialiser ?

Une solution est l'initialisation dite paresseuse (ou *lazy initialization* en anglais). Cela est possible avec l'introduction d'un message d'accès qui initialise la variable, durant l'exécution, si celle-ci n'a pas été encore initialisée. Ceci nous oblige à utiliser la méthode d'accès tout le temps et à ne jamais faire appel à la variable de classe directement. De plus, notons que le coût de l'envoi d'un accesseur et le test d'initialisation sont à prendre en compte. Ceci va à l'encontre de notre motivation à utiliser une variable de classe, parce qu'en réalité elle n'est plus partagée.

Méthode 5.21 – Color class»colorNames

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

Une autre solution consiste à surcharger la méthode `initialize`.

Méthode 5.22 – Color class»initialize

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

Si vous adoptez cette solution, vous devez vous rappeler qu'il faut invoquer la méthode initialize après que vous l'ayez définie, *par ex.*, en utilisant Color initialize. Bien que les méthodes côté classe soient exécutées automatiquement lorsque le code est chargé en mémoire, elles ne sont *pas* exécutées durant leur saisie et leur compilation dans le navigateur Browser ou en phase d'édition et de recompilation.

Les variables de pool

Les variables de *pool*²⁰ sont des variables qui sont partagées entre plusieurs classes qui ne sont pas liées par une arborescence d'héritage. à la base, les variables de pool sont stockées dans des dictionnaires de pool ; maintenant elles devraient être définies comme variables de classe dans des classes dédiées (sous-classes de SharedPool). Notre conseil : évitez-les. Vous n'en aurez besoin qu'en des circonstances exceptionnelles et spécifiques. Ici, notre but est de vous expliquer suffisamment les variables de pool pour comprendre leur fonction quand vous les rencontrez durant la lecture de code.

Une classe qui accède à une variable de pool doit mentionner le *pool* dans sa définition de classe. Par exemple, la classe Text indique qu'elle emploie le dictionnaire de pool TextConstants qui contient toutes les constantes textuelles telles que CR and LF. Ce dictionnaire a une clé #CR à laquelle est affectée la valeur Character cr, *c-à-d.* le caractère retour-chariot ou carriage return.

Classe 5.23 – Dictionnaire de pool dans la classe Text

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

Ceci permet aux méthodes de la classe Text d'accéder aux clés du dictionnaire *directement* dans le corps de la méthode, *c-à-d.* en utilisant la syntaxe de variable plutôt qu'une recherche explicite dans le dictionnaire. Par exemple, nous pouvons écrire la méthode suivante.

Méthode 5.24 – Text»testCR

20. Pool signifie piscine en anglais, ces variables sont dans un même bain !

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

Encore une fois, nous recommandons d'éviter d'utiliser les variables et les dictionnaires de pool.

5.8 Résumé du chapitre

Le modèle objet de Pharo est à la fois simple et uniforme. Tout est objet et quasiment tout se passe via l'envoi de messages.

- Tout est objet. Les entités primitives telles que les entiers sont des objets, tout comme il est vrai que les classes soient des objets de premier ordre.
- Tout objet est instance d'une classe. Les classes définissent la structure de leurs instances via des variables d'instance *privées* et leur comportement via des méthodes *publiques*. Chaque classe est l'unique instance de sa métaclass. Les variables de classe sont des variables privées partagées par la classe et toutes les instances de la classe. Les classes ne peuvent pas accéder directement aux variables d'instance de leurs instances et les instances ne peuvent pas accéder aux variables de classe de leur classe. Les méthodes d'accès (accesseurs et mutateurs) doivent être définies au besoin.
- Toute classe a une super-classe. La racine de la hiérarchie basée sur l'héritage simple est ProtoObject. Cependant les classes que vous définissez devrait normalement hériter de la classe Object ou de ses sous-classes. Il n'y a pas d'élément sémantique pour la définition de classes abstraites. Une classe abstraite est simplement une classe avec au moins une méthode abstraite — une dont l'implémentation contient l'expression self subclassResponsibility. Bien que Pharo ne dispose que du principe d'héritage simple, il est facile de partager les implémentations de méthodes en regroupant ces dernières en *traits*.
- Tout se passe par envoi de messages.
Nous ne faisons pas des "appels de méthodes", nous faisons des "envois de messages". Le receveur choisit alors sa propre méthode pour répondre au message.
- La recherche de méthodes suit la chaîne d'héritage ; Les envois à self sont dynamiques et la recherche de méthode démarre dans le receveur de la classe, alors que celles à super sont statiques et la recherche commence dans la super-classe de la classe dans laquelle l'envoi à super est écrit.
- Il y a trois sortes de variables partagées. Les variables globales sont accessibles partout dans le système. Les variables de classe sont partagées entre une classe, ses sous-classes et ses instances. Les variables

de pool sont partagées dans un ensemble de classes particulier. Vous devez éviter l'emploi de variables partagées autant que possible.

Chapitre 6

L'environnement de programmation de Pharo

L'objectif de ce chapitre est de vous montrer comment développer des programmes dans l'environnement de programmation de Pharo. Vous avez déjà vu comment définir des méthodes et des classes en utilisant le navigateur de classes ; ce chapitre vous présentera plus de caractéristiques du Browser ainsi que d'autres navigateurs.

Bien entendu, vous pouvez occasionnellement rencontrer des situations dans lesquelles votre programme ne marche pas comme voulu. Pharo a un excellent débogueur, mais comme la plupart des outils puissants, il peut s'avérer déroutant au début. Nous vous en parlerons au travers de sessions de débogages et vous montrerons certaines de ses possibilités.

Lorsque vous programmez, vous le faites dans un monde d'objets vivants et non dans un monde de programmes textuels statiques ; c'est une des particularités uniques de Smalltalk. Elle permet d'obtenir une réponse très rapide de vos programmes et vous rend plus productif. Il y a deux outils vous permettant l'observation et aussi la modification de ces objets vivants : l'*Inspector* (ou inspecteur) et l'*Explorer* (ou explorateur).

La programmation dans un monde d'objets vivants plutôt qu'avec des fichiers et un éditeur de texte vous oblige à agir explicitement pour exporter votre programme depuis l'image Smalltalk.

La technique traditionnelle, aussi supportée par tous les dialectes Smalltalk consiste à créer un fichier d'exportation *fileout* ou une archive d'échange dit *change set*. Il s'agit principalement de fichiers textes encodés pouvant être importés dans un autre système. Une technique plus récente de Pharo est le chargement de code dans un dépôt de versions sur un serveur. Elle est plus efficace surtout en travail coopératif et est rendu possible via un

outil nommé Monticello.

Finalement, en travaillant, vous pouvez trouver un *bug* (dit aussi bogue) dans Pharo ; nous vous expliquerons aussi comment reporter les bugs et comment soumettre les corrections de bugs ou *bug fixes*.

6.1 Une vue générale

Smalltalk et les interfaces graphiques modernes ont été développées ensemble. Bien avant la première sortie publique de Smalltalk en 1983, Smalltalk avait un environnement de développement graphique écrit lui-même en Smalltalk et tout le développement est intégré à cet environnement. Commençons par jeter un coup d'œil sur les principaux outils de Pharo.

- Le **Browser** ou *navigateur de classes* est l'outil de développement central. Vous l'utiliserez pour créer, définir et organiser vos classes et vos méthodes. Avec lui, vous pourrez aussi naviguer dans toutes les classes de la bibliothèque de classes : contrairement aux autres environnements où le code source est réparti dans des fichiers séparés, en Smalltalk toutes les classes et méthodes sont contenues dans l'image.
- L'outil **Message Names** sert à voir toutes les méthodes ayant un sélecteur (noms de messages sans argument) spécifique ou dont le sélecteur contient une certaine sous-chaine de caractères.
- Le **Method Finder** vous permet aussi de trouver des méthodes, soit selon leur *comportement*, soit en fonction de leur nom.
- Le **Monticello Browser** est le point de départ pour le chargement ou la sauvegarde de code via des paquetages Monticello dit aussi *packages*.
- Le **Process Browser** offre une vue sur l'ensemble des processus (threads) exécutés dans Smalltalk.
- Le **Test Runner** permet de lancer et de déboguer les tests unitaires SUnit. Il est décrit dans le chapitre 7.
- Le **Transcript** est une fenêtre sur le flux de données sortant de Transcript. Il est utile pour écrire des fichiers-journaux ou *log* et a déjà été étudié dans la section 1.5.
- Le **Workspace** ou *espace de travail* est une fenêtre dans laquelle vous pouvez entrer des commandes. Il peut être utilisé dans plusieurs buts mais il l'est plus généralement pour taper des expressions Smalltalk et les exécuter avec **do it**¹. L'utilisation de Workspace a déjà été vu dans la section 1.5.

L'outil **Debugger** a un rôle évident, mais vous découvrirez qu'il a une place plus centrale comparé aux débogueurs d'autres langages de programmation car en Smalltalk vous pouvez *programmer* dans l'outil Debugger. Il n'est pas lancé depuis un menu ; il apparaît normalement en situation d'ér-

1. En anglais, *do it* correspond à l'exclamation “fais-le!”.

reur, en tapant CMD-. pour interrompre un processus lancé ou encore en insérant une expression `self halt` dans le code.

6.2 Le Browser

De nombreux navigateurs de classes ou *browsers* ont été développés pour Smalltalk durant des années. Pharo simplifie l'histoire en proposant un unique navigateur disposant de multiples vues, le Browser. La figure 6.1 présente le Browser tel qu'il apparaît lorsque vous l'ouvrez pour la première fois².

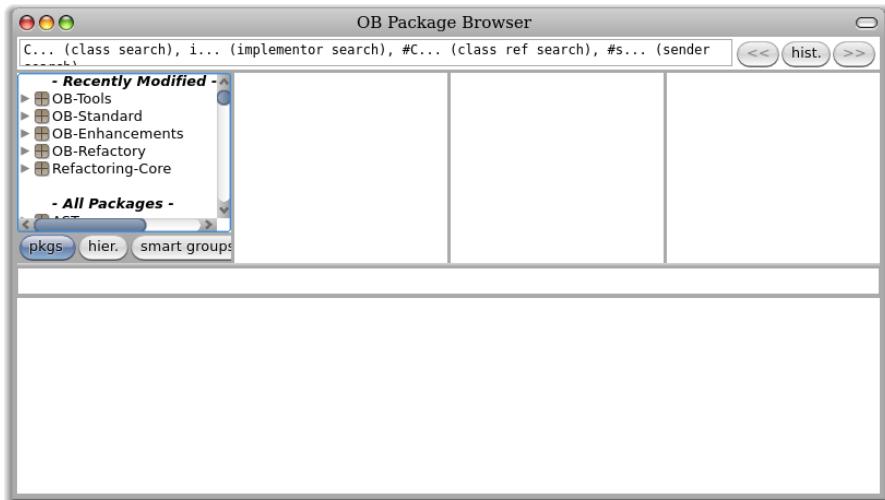


FIGURE 6.1 – Le navigateur de classes.

Les quatre petits panneaux en haut du Browser représentent la vue hiérarchique des méthodes dans le système de la même manière que le *File Viewer* de NeXTstep et le *Finder* de Mac OS X fournissent une vue en colonnes des fichiers du disque.

Le premier panneau de gauche liste les *paquetages* de classes ou, en anglais, *packages*; sélectionnez-en une (disons *Kernel*) et alors le panneau immédiatement à droite affichera toutes les classes incluses dans ce paquetage.

De même, si vous sélectionnez une des classes de ce second panneau, disons, *Model* (voir la figure 6.2), le troisième panneau vous affichera tous

2. Rappelez-vous que si votre navigateur ne ressemble pas à celui présenté sur la figure 6.1, vous pourriez avoir besoin de changer le navigateur par défaut (voir la FAQ ??, p. ??).

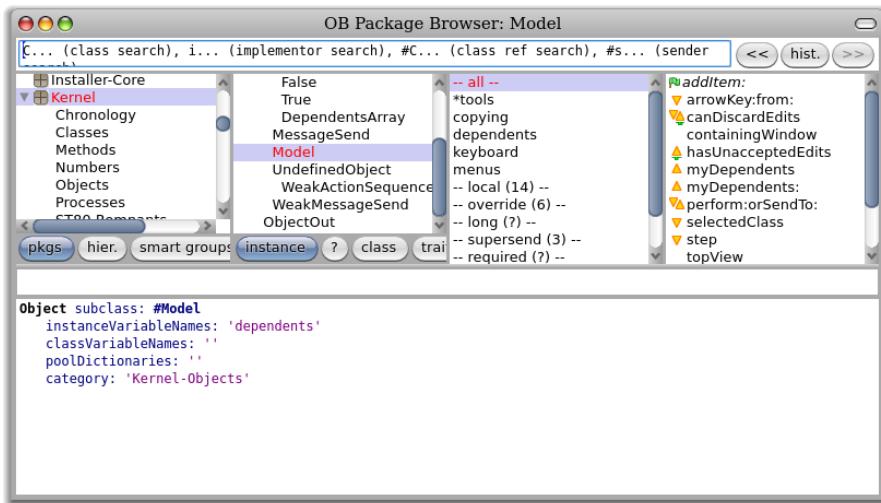


FIGURE 6.2 – Le Browser avec la classe Model sélectionnée.

les protocoles définis pour cette classe ainsi qu'un protocole virtuel `--all-` (désignant l'ensemble des catégories). Ce dernier est sélectionné par défaut. Les protocoles sont une façon de catégoriser les méthodes ; ils rendent la recherche des méthodes plus facile et détaillent le comportement d'une classe en la découplant en petites divisions cohérentes. Le quatrième panneau montre les noms de toutes les méthodes définies dans le protocole sélectionné. Si vous sélectionnez enfin un nom de méthode, le code source de la méthode correspondante apparaît dans le grand panneau inférieur du navigateur. Là, vous pouvez voir, éditer et sauvegarder la version éditée. Si vous sélectionnez la classe `Model`, le protocole `dependents` et la méthode `myDependents`, le navigateur devrait ressembler à la figure 6.3.

Contrairement aux répertoires du *Finder* de Mac OS X, les quatre panneaux supérieurs ne sont aucunement égaux. Là où les classes et les méthodes font partie du langage Smalltalk, les paquetages et les protocoles ne sont que des commodités introduites par le navigateur pour limiter la quantité d'information que chaque panneau pourrait présenter. Par exemple, s'il n'y avait pas de protocoles, le navigateur devrait afficher la liste de toutes les méthodes dans la classe choisie ; pour la plupart des classes, cette liste sera trop importante pour être parcourue aisément.

De ce fait, la façon dont vous créez un nouveau paquetage ou un nouveau protocole est différente de la manière avec laquelle vous créez une nouvelle classe ou une nouvelle méthode. Pour créer un nouveau paquetage, sélectionnez `new package` dans le menu contextuel accessible en cliquant avec le bouton d'action dans le panneau des paquetages ; pour créer un nouveau

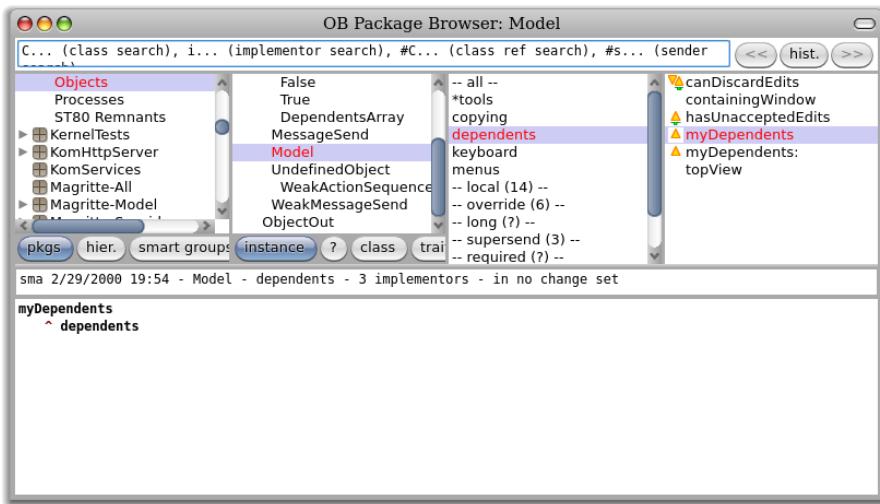


FIGURE 6.3 – Le Browser affichant la méthode myDependents de la classe Model.

protocole, sélectionnez new protocol via le menu accessible en cliquant avec le bouton d'action dans le panneau des protocoles. Entrez le nom de la nouvelle entité (paquetage ou protocole) dans la zone de saisie, et voilà ! Un paquetage ou un protocole, ça n'est qu'un nom et son contenu.



FIGURE 6.4 – Le Browser montrant le patron de création de classe.

à l'opposé, créer une classe ou une méthode nouvelle nécessite l'écriture

de code Smalltalk. Si vous cliquez sur le paquetage actuellement sélectionné dans le panneau de gauche, le panneau inférieur affichera un patron de création de classe (voir la figure 6.4). Vous créez une nouvelle classe en éditant ce patron ou *template*. Pour ce faire, remplacez Object par le nom de la classe existante que vous voulez dériver, puis remplacez NameOfSubclass par le nom que vous avez choisi pour votre nouvelle classe (sous-classe de la première) et enfin, remplissez la liste des noms de variables d'instance si vous en connaissez. La catégorie pour la nouvelle classe est par défaut la catégorie du paquetage actuellement sélectionné³, mais vous pouvez à loisir la changer si vous voulez. Si vous avez déjà la classe à dériver sélectionnée dans le Browser, vous pouvez obtenir le même patron avec une initialisation quelque peu différente en cliquant avec le bouton d'action dans le panneau des classes et en sélectionnant class templates ... ▷ subclass template. Vous pouvez aussi éditer simplement la définition de la classe existante en changeant le nom de la classe en quelque chose d'autre. Dans tous les cas, à chaque fois que vous acceptez la nouvelle définition, la nouvelle classe (celle dont le nom est précédé par #) est créée (ainsi que sa métaclass associée). Créer une classe crée aussi une variable globale référençant la classe. En fait, l'existence de celle-ci vous permet de vous référer à toutes les classes existantes en utilisant leur nom. définir une classe

Voyez-vous pourquoi le nom d'une nouvelle classe doit apparaître comme un Symbol (c-à-d. préfixé avec #) dans le patron de création de classe, mais qu'après la création de classe, le code peut s'y référer en utilisant son nom comme identifiant (c-à-d. sans le #) ?

Le processus de création d'une nouvelle méthode est similaire. Premièrement sélectionnez la classe dans laquelle vous voulez que la méthode apparaisse, puis sélectionnez un protocole. Le navigateur affichera un patron de création de méthode que vous pouvez remplir et éditer, comme indiqué par la figure 6.5. définir une méthode

Naviguer dans l'espace de code

Le navigateur de classes fournit plusieurs outils pour l'exploration et l'analyse de code. Ces outils sont accessibles en cliquant avec le bouton d'action dans divers menus contextuels ou (pour les outils les plus communs) via des raccourcis-clavier.

Ouvrir une nouvelle fenêtre de Browser

Vous aurez besoin parfois d'ouvrir de multiples navigateurs de classes. Lorsque vous écrivez du code, vous aurez presque toujours besoin d'au

3. Rappelez-vous que paquetages et catégories ne sont pas exactement la même chose. Nous verrons la relation qui existe entre eux dans la section 6.3.



FIGURE 6.5 – Le Browser montrant le patron de création de méthode.

moins deux fenêtres : une pour la méthode que vous éditez et une pour naviguer dans le reste du système pour y voir ce dont vous aurez besoin pour la méthode éditée dans la première. Vous pouvez ouvrir un Browser sur une classe en sélectionnant son nom et en utilisant le raccourci-clavier CMD-b raccourci-clavier.

 *Essayez ceci : dans un espace de travail ou Workspace, saisissez le nom d'une classe (par exemple, Morph), sélectionnez-le et pressez CMD-b. Cette astuce est souvent utile ; elle marche depuis n'importe quelle fenêtre de texte.*

Senders et implementors d'un message

Cliquer avec le bouton d'action sur `browse ... > senders (n)` dans le panneau des méthodes vous renvoie une liste de toutes les méthodes pouvant utiliser la méthode sélectionnée. En prenant le Browser ouvert sur Morph, cliquez sur la méthode `drawOn:` dans le panneau des méthodes ; le corps de `drawOn:` s'affiche dans la partie inférieure du navigateur. Si vous sélectionnez `senders (n)` (voir la figure 6.6), un menu apparaîtra avec `drawOn:` comme premier élément de la pile, suivi de tous les messages que `drawOn:` envoie (voir la figure 6.7). Sélectionner un message dans ce menu ouvrira un navigateur avec la liste de toutes les méthodes dans l'image qui envoie le message choisi (voir la figure 6.8).

Le "n" dans `senders (n)` vous informe que le raccourci-clavier pour trouver les `senders` (c-à-d. les méthodes émettrices) d'un message est CMD-n.

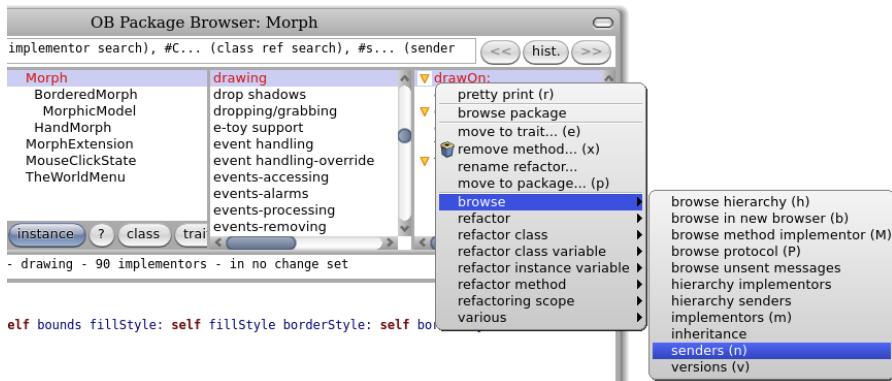


FIGURE 6.6 – L'élément de menu senders (n).

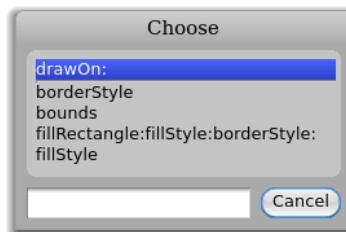


FIGURE 6.7 – Choisir un message dans la liste pour avoir ses senders.

Cette commande fonctionne depuis *n'importe quelle* fenêtre de texte.

➊ Sélectionnez le texte “drawOn :” dans le panneau de code et pressez CMD-n pour faire apparaître immédiatement les senders de drawOn:.

Si vous regardez bien les senders de drawOn: dans AtomMorph>drawOn:, vous verrez qu'il s'agit d'un envoi à super. Nous savons ainsi que la méthode qui sera exécutée est dans la superclasse de AtomMorph. Quelle est cette classe ? Cliquez avec le bouton d'action sur browse > hierarchy implementors et vous verrez qu'il s'agit de EllipseMorph.

Maintenant observons le sixième sender de la liste, Canvas>draw, comme le montre la figure 6.8. Vous pouvez voir que cette méthode envoie drawOn: à n'importe quel objet passé en argument ; ce peut être une instance de n'importe quelle classe. L'analyse du flux de données peut nous aider à mettre la main sur la classe du receveur de certains messages, mais de manière générale, il n'a pas de moyen simple pour que le navigateur sache quelle méthode sera exécutée à l'envoi d'un message.

C'est pourquoi, le navigateur de “senders” (c-à-d. le Browser des mé-

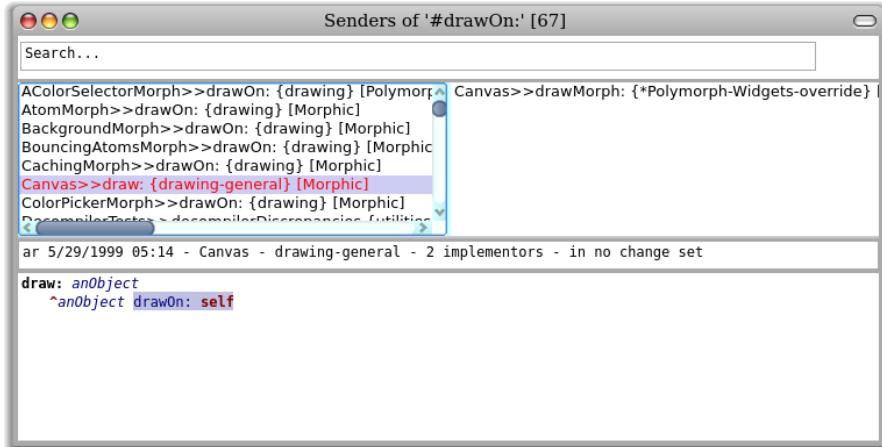


FIGURE 6.8 – Le navigateur Senders Browser montrant que la méthode `Canvas>>draw` envoie le message `drawOn:` à son argument.

thodes émettrices) nous montre exactement ce que son nom suggère : tous les *senders* d'un message ayant *pour nom* le sélecteur choisi. Ce navigateur devient grandement indispensable quand vous avez besoin de comprendre le rôle d'une méthode : il vous permet de naviguer rapidement à travers les exemples d'usage. Puisque toutes les méthodes avec un même sélecteur devraient être utilisées de la même manière, toutes les utilisations d'un message donné devraient être semblables.

L'Implementors Browser fonctionne de même mais, au lieu de lister les senders d'un message, il affiche toutes les classes contenantes ou *implementors*, c-à-d. les classes qui implémentent une méthode avec le même sélecteur que celui sélectionné. Sélectionnez, par exemple, `drawOn:` dans le panneau des méthodes et sélectionnez `browse > implementors (m)` (ou sélectionnez le texte "`drawOn:`" dans la zone inférieure du code et pressez `CMD-m`). Vous devriez voir une fenêtre listant des méthodes montrant ainsi la liste déroulante des 59 classes qui implémentent une méthode `drawOn:`. Ceci ne devrait pas être si surprenant qu'il y ait tant de classes implémentant cette méthode : `drawOn:` est le message qui est compris par chaque objet capable de se représenter graphiquement à l'écran.

Les versions d'une méthode

Quand vous sauvegardez une nouvelle version d'une méthode, l'ancienne version n'est pas perdue. Pharo garde toutes les versions passées et vous permet de comparer les différentes versions entre elles et de revenir (en

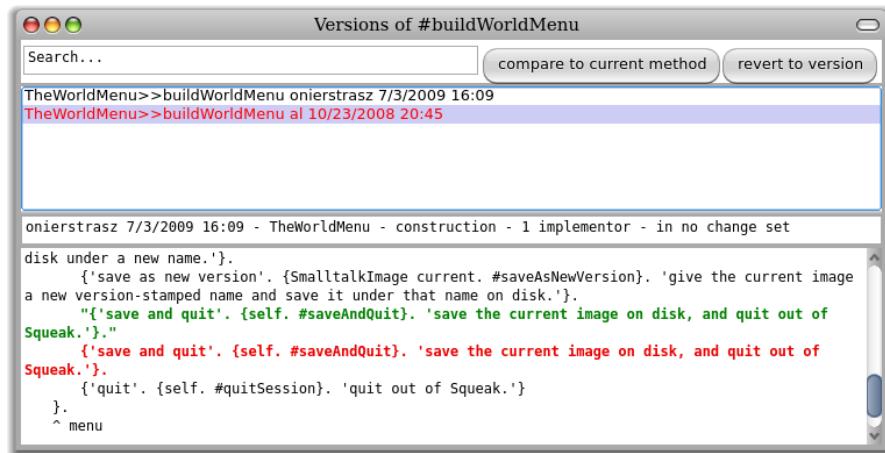


FIGURE 6.9 – Le Versions Browser montre deux versions de la méthode `TheWorldMenu>>buildWorldMenu`:

anglais, “revert”) à une ancienne version.

L’élément de menu `browse >versions (v)` donne accès aux modifications successives effectuées sur la méthode sélectionnée. Dans la figure 6.9 nous pouvons voir deux versions de la méthode `buildWorldMenu`:

Le panneau supérieur affiche une ligne pour chaque version de la méthode incluant les initiales du programmeur qui l’a écrite, la date et l’heure de sauvegarde, les noms de la classe et de la méthode et le protocole dans lequel elle est définie. La version courante (active) est au sommet de la liste ; quelle que soit la version sélectionnée affichée dans le panneau inférieur. Les boutons offrent aussi l’affichage des différences entre la méthode sélectionnée et la version courante et la possibilité de revenir à la version choisie.

Le Versions Browser existe pour que vous ne vous inquiétez jamais de la préservation de code que vous pensiez ne plus avoir besoin : effacez-le simplement. Si vous vous rendez compte que vous en avez vraiment besoin, vous pouvez toujours revenir à l’ancienne version ou copier le morceau de code utile de la version antérieure pour le coller dans une autre méthode.

Ayez pour habitude d’utiliser les versions ; “commenter” le code qui n’est plus utile n’est pas une bonne pratique car ça rend le code courant plus difficile à lire. Les Smalltalkiens⁴ accordent une extrême importance à la lisibilité du code.

ASTUCE Qu’en est-il du cas où vous décidez de revenir à une méthode

4. En anglais, nous les appelons *Smalltalkers*.

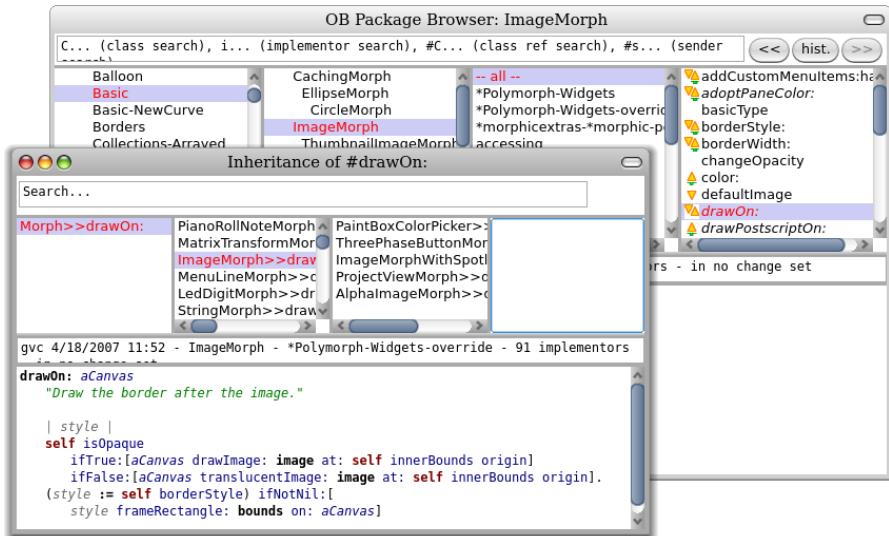


FIGURE 6.10 – `ImageMorph>>drawOn:` et les méthodes qu'il surcharge. Les méthodes apparentées ou *siblings* des méthodes sélectionnées sont visibles dans les listes déroulantes .

que vous avez entièrement effacée ? Vous pouvez trouver l'effacement dans un change set dans lequel vous pouvez demander à visiter les versions en cliquant avec le bouton d'action. Le navigateur de change set est décrit dans la section 6.8

Les surcharges de méthodes

Le Inheritance Browser est un navigateur spécialisé affichant toutes les méthodes surchargées par la méthode affichée. Pour le voir à l'action, sélectionnez la méthode `ImageMorph>>drawOn:` dans le Browser. Remarquez les icônes triangulaires juxtant le nom des méthodes (voir la figure 6.11). Le triangle pointant vers le haut vous indique que `ImageMorph>>drawOn:` surcharge une méthode héritée (c-à-d. `Morph>>drawOn:`) et triangle pointant vers le bas vous indique que cette méthode est surchargée dans ses sous-classes (vous pouvez aussi cliquer sur les icônes pour naviguer vers ces méthodes). Sélectionnez maintenant `browse > inheritance`. L'Inheritance Browser vous montre la hiérarchie de méthodes surchargées (voir la

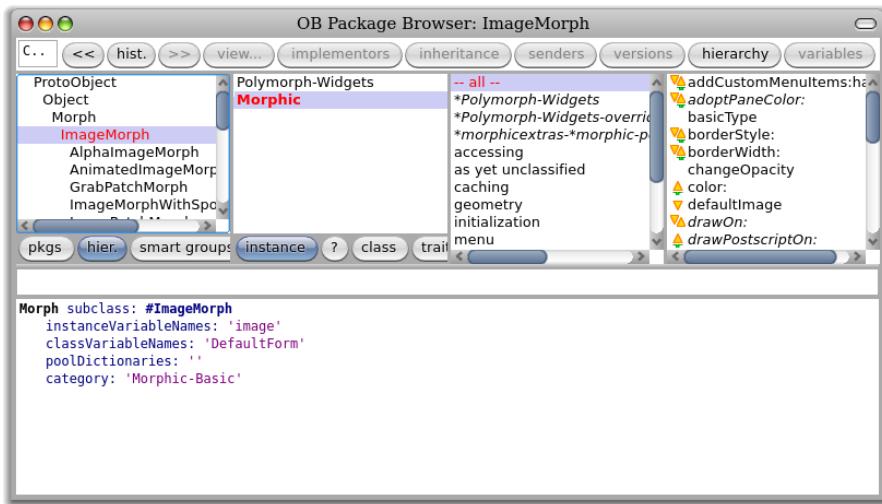


FIGURE 6.11 – Une vue hiérarchique de ImageMorph.

La vue hiérarchique

Par défaut, le navigateur présente une liste de paquetages dans son panneau supérieur gauche. Cependant, il est possible de changer le contenu de ce panneau pour avoir une vue hiérarchique des classes. Pour cela, sélectionner tout simplement une classe de votre choix, disons `ImageMorph` et cliquer sur le bouton `[hier.]`. Vous verrez alors dans le panneau le plus à gauche une hiérarchie de classes affichant toutes les super-classes et sous-classes de la classe sélectionnée. Le second panneau liste les paquetages implémentant les méthodes de la classe sélectionnée. Sur la figure 6.11, la vue hiérarchique dans le navigateur montre que `ImageMorph` est la super-classe directe de `Morph`.

Trouver les références aux variables

En cliquant avec le bouton d'action sur une classe dans le panneau de classes du navigateur, puis en sélectionnant `browse > chase variables`⁵, vous pouvez trouver où une certaine variable — d'instance ou de classe — est utilisée. Vous naviguez au travers des méthodes d'accès de toutes les variables d'instance ou de classe via ce *navigateur de poursuite* et, en retour, visitez les méthodes qui envoyent ces accesseurs, et ainsi de suite (voir `chasingBrowser`).

5. Chase signifie “poursuivre” en anglais.

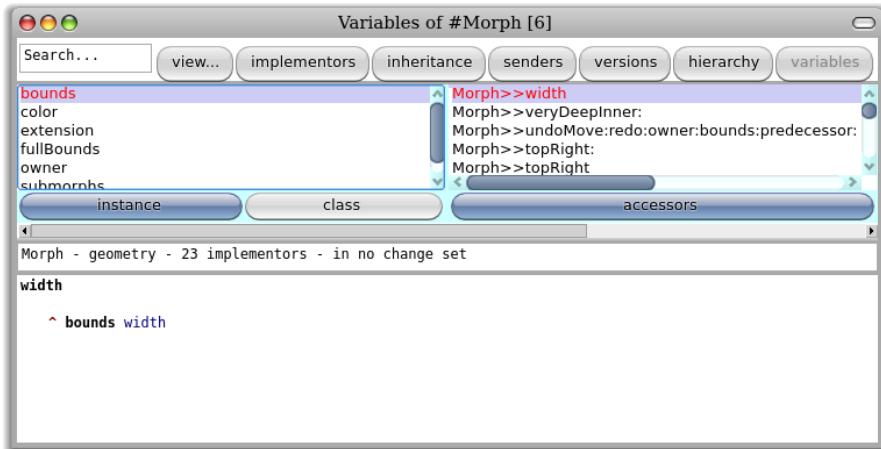


FIGURE 6.12 – Un navigateur de poursuite ouvert sur Morph.

Le panneau de code

L’option de menu `various ▷ view ...` disponible en cliquant avec le bouton d’action dans le panneau des méthodes affiche le menu “comment faut-il l’afficher” qui vous permet de choisir comment le navigateur va afficher la méthode sélectionnée dans le panneau inférieur *c-à-d.* le panneau de code (ou *panneau source*). Parmi les propositions, nous avons l’affichage du code source, du code source en mode `prettyPrint` (affichage élégant), du code compilé ou `byteCode`, ou encore du code source décompilé depuis le `bytecode` via `decompile`.

Remarquez que le choix de `prettyPrint` dans ce menu n’est *absolument pas* le même que le travail en mode `pretty-print` d’une méthode avant sa sauvegarde⁶. Le menu contrôle seulement l’affichage du navigateur et n’a aucun effet sur le code enregistré dans le système. Vous pouvez le vérifier en ouvrant deux navigateurs et en sélectionnant `prettyPrint` pour l’un et `source` pour l’autre. Pointer les deux navigateurs sur la même méthode et en choisissant `byteCode` dans l’un et `decompile` dans l’autre est vraiment une bonne manière d’en apprendre plus sur le jeu d’instructions codées (dit aussi *byte-codées*) de la machine virtuelle Pharo.

6. `pretty print (r)` est la première option de menu dans le panneau des méthodes ou celui à mi-hauteur dans le menu du panneau de code.

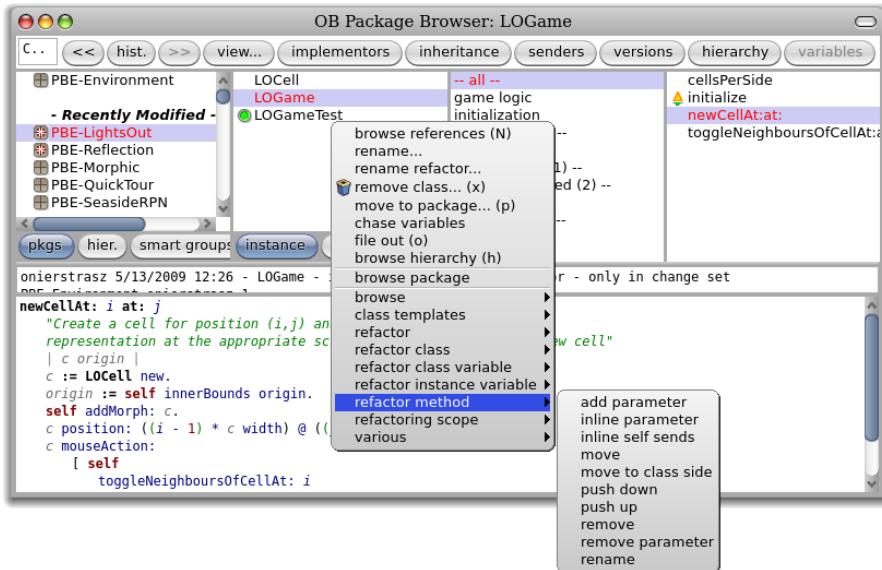


FIGURE 6.13 – Refactoring operations.

La refactorisation

Les menus contextuels proposent un grand nombre de refactorisations (ou *refactoring*) classiques. Cliquez avec le bouton d'action sur l'un des quatre panneaux supérieurs pour voir les opérations de refactoring actuellement disponibles (voir la figure 6.13). à l'origine, cette fonction était disponible uniquement par un navigateur spécifique nommé Refactoring Browser, mais elle peut désormais être accessible depuis n'importe quel navigateur.

Les menus du navigateur

De nombreuses fonctions complémentaires sont disponibles en cliquant avec le bouton d'action dans les panneaux du Browser. Même si les options de menus portent le même nom, leur *signification* dépend du contexte. Par exemple, le panneau des paquetages, le panneau des classes, le panneau des protocoles et enfin, celui des méthodes ont tous *file out* dans leurs menus respectifs. Cependant, chaque *file out* fait une chose différente : dans le panneau des paquetages, il enregistre entièrement dans un fichier le paquetage sélectionné ; dans le celui des classes, des protocoles ou des méthodes, il exporte respectivement la classe entière, le protocole entier ou la méthode affichée. Bien qu'apparemment évident, ce peut être une source de confusion pour

les débutants.

L'option de menu probablement la plus utile est `find class... (f)` dans le panneau des paquetages. Elle permet de trouver une classe. ~~Bien que les catégories soient utiles pour arranger le code que nous sommes en train de développer, la plupart d'entre nous ne connaissent pas la catégorisation de tout le système, et c'est beaucoup plus rapide en tapant CMD-f suivi par les premiers caractères du nom d'une classe que de deviner dans quel paquetage elle peut bien être. recent classes... (r) vous aide aussi à retrouver rapidement une classe parmi celles que vous avez visitées récemment, même si vous avez oublié son nom.~~

Vous pouvez aussi rechercher une classe ou méthode en particulier en saisissant son nom dans la boîte de requête située dans la partie supérieure gauche de votre navigateur. Quand vous tapez sur la touche "entrée", une requête sera envoyée au système et son résultat sera affiché. Notez qu'en préfixant votre requête avec `#`, vous pouvez chercher toutes les références à une classe ou tous les *senders* d'un message. Cependant, si vous recherchez une méthode dans une classe sélectionnée, il est souvent plus efficace de naviguer dans le protocole `--all--` (qui d'ailleurs est le choix par défaut), placer la souris dans le panneau des méthodes et taper la première lettre du nom de la méthode que vous recherchez. Ceci va faire glisser l'ascenseur du panneau jusqu'à ce que la méthode souhaitée soit visible.

❶ Essayez les deux techniques de navigation pour `OrderedCollection» removeAt:`

Il y a beaucoup d'autres options dans les menus. Passer quelques minutes à tester les possibilités du navigateur est véritablement payant.

❷ Comparez le résultat de `Browse Protocol`, `Browse Hierarchy`, et `Show Hierarchy` dans le menu contextuel du panneau de classes.

Naviguer par programme

La classe `SystemNavigation` offre de nombreuses méthodes utiles pour naviguer dans le système. Beaucoup de fonctionnalités offertes par le navigateur classique sont programmées par `SystemNavigation`.

❸ Ouvrez un espace de travail `Workspace` et exécutez le code suivant pour naviguer dans la liste des senders du message `drawOn:` en utilisant `do it :`

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

Pour restreindre le champ de la recherche à une classe spécifique :

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

Les outils de développement sont complètement accessibles depuis un programme car *ceux-ci sont aussi des objets*. Vous pouvez dès lors développer vos propres outils ou adapter ceux qui existent déjà selon vos besoins.

L'équivalent programmatique de l'option de menu `implementors` est :

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

Pour en apprendre plus sur ce qui est disponible, explorez la classe `SystemNavigation` avec le navigateur.

Des exemples supplémentaires peuvent être trouvés dans le chapitre A.

6.3 Monticello

Nous vous avons donné un aperçu de Monticello, l'outil de gestion de paquetages de Pharo dans la section 2.9. Cependant Monticello a beaucoup plus de fonctions que celles dont nous allons discuter ici. Comme Monticello gère des paquetages dits *packages*, nous allons expliquer ce qu'est un paquetage avant d'aborder Monticello proprement dit.

Les paquetages : une catégorisation déclarative du code de Pharo

Dans la section 2.3, nous avons pointé le fait que les paquetages sont plus ou moins équivalents aux catégories. Nous allons désormais voir la relation qui existe entre eux. Le système du paquetage est une façon simple et légère d'organiser le code source de Smalltalk ; il exploite une simple convention de nommage pour les catégories et les protocoles.

Prenons l'exemple suivant en guise d'explication. Supposons que nous soyons en train de développer une librairie pour nous faciliter l'utilisation d'une base de données relationnelles depuis Pharo. Vous avez décidé d'appeler votre librairie (ou *framework*) `PharoLink` et vous avez créé une série de catégories contenant toutes les classes que vous avez écrites, *par ex.*, la catégorie '`PharoLink-Connections`' contient les classes `OracleConnection` `MySQLConnection` `PostgresConnection` et la catégorie '`PharoLink-Model`' contient les classes `DBTable` `DBRow` `DBQuery` et ainsi de suite. Cependant, tout le code ne résidera pas dans ces classes. Par exemple, vous pouvez aussi avoir une série de méthodes pour convertir des objets dans un format sympathique pour notre format SQL⁷ :

```
Object»asSQL
String»asSQL
```

7. Nous dirions que ce format est SQL-friendly.

Date»asSQL

Ces méthodes appartiennent au même paquetage que les classes dans les catégories PharoLink–Connections et PharoLink–Model. Mais la classe Object n'appartient clairement pas à notre paquetage ! Vous avez donc besoin de trouver un moyen pour associer certaines *méthodes* à un paquetage même si le reste de la classe est dans un autre.

Pour ce faire, nous plaçons ces méthodes (de Object, String, Date etc) dans un protocole nommé **PharoLink* (remarquez l'astérisque en début de nom). L'association des catégories en *PharoLink*... et des protocoles **PharoLink* forme un paquetage nommé PharoLink. Précisement, les règles de formation d'un paquetage s'énoncent comme suit.

Un paquetage appelé Foo contient :

1. toutes les *définitions de classe* des classes présentes dans la catégorie *Foo* ou toutes catégories avec un nom commençant par *Foo-* ;
2. toutes les *méthodes* dans *n'importe quelle classe* dont le protocole se nomme **Foo* ou **foo*⁸, et ;
3. toutes les *méthodes* dans les classes présentes dans *Foo* ou toutes catégories avec un nom commençant par *Foo-*, exception faite des méthodes dont le nom des protocoles débute par ***.

Une conséquence de ces règles est que chaque définition de classe et chaque méthode appartiennent exactement à un paquetage. L'*exception* de la dernière règle est justifiée parce que ces méthodes doivent appartenir à d'autres paquetages. La raison pour laquelle la casse est ignorée dans la règle 2 est que, conventionnellement les noms de protocoles sont typiquement (mais pas nécessairement) en minuscules (et peuvent inclure des espaces) ; alors que les noms de catégories utilisent un format d'écriture en forme chameau comme par exemple AlanKay, LargePositiveInteger ou CamelCase (d'ailleurs CamelCase est le nom anglais de ce type de format de noms).

La classe PackageInfo implémente ces règles et vous pouvez mieux les appréhender en expérimentant cette classe.

❶ Évaluez l'expression suivante dans un espace de travail :

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

Il est possible maintenant de faire une introspection de ce paquetage. Par exemple, imprimer via print it le code mc classes dans l'espace de travail nous retourne la longue liste des classes qui font le paquetage Monticello. L'expression mc coreMethods nous renvoie une liste de MethodReferences ou réfé-

8. Durant la comparaison de ces noms, la casse des lettres est parfaitement ignorée.

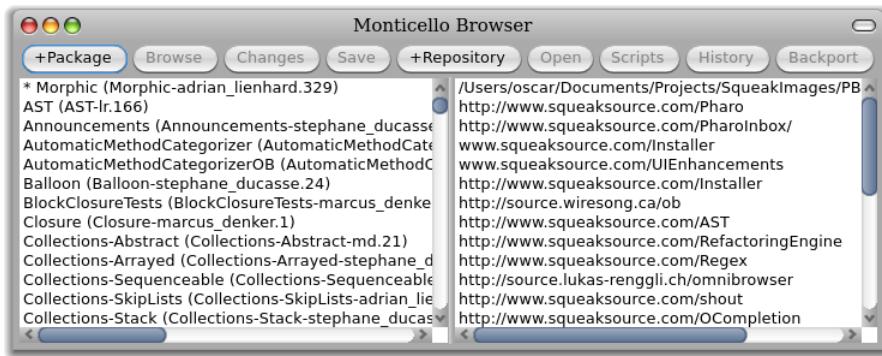


FIGURE 6.14 – Le navigateur Monticello.

rences de méthodes pour toutes les méthodes de ces classes. La requête `mc extensionMethods` est peut-être une des plus intéressantes : elle retourne la liste de toutes les méthodes contenues dans le paquetage Monticello qui ne sont pas dans une classe de Monticello.

Les paquetages sont des ajouts à Pharo relativement récents mais, puisque les conventions de nommage de paquetage sont basées sur celles déjà existantes, il est possible d'utiliser PackageInfo pour analyser du code plus ancien qui n'a pas été explicitement adapté pour pouvoir y répondre.

 *Imprimer le code (PackageInfo named: 'Collections') externalSubclasses ; cette expression répond une liste de toutes les sous-classes de Collection qui ne sont pas dans le paquetage Collections.*

Les fondamentaux de Monticello

Monticello est nommé ainsi d'après la villégiature de Thomas Jefferson, troisième président des États-Unis d'Amérique et auteur de la statue pour les libertés religieuses (Religious Freedom) en Virginie. Le nom signifie "petite montagne" en italien, en ainsi, il est toujours prononcé avec un "c" italien, *c-à-d.* avec le son *tch* comme dans "quetsche" : Monn-ti-tchel-lo⁹.

Quand vous ouvrez le navigateur Monticello, vous voyez deux panneaux de listes et une ligne de boutons, comme sur la figure 6.14.

Le panneau de gauche liste tous les paquetages qui ont été chargés dans l'image actuelle ; la version courante du paquetage est présentée entre parenthèses à la suite de son nom.

9. Note du traducteur : c'est aussi une commune de Haute-Corse.

Celle de droite liste tous les dépôts (ou *repository*) de code source que Monticello connaît généralement pour les avoir utilisés pour charger le code. Si vous sélectionnez un paquetage dans le panneau de gauche, celui de droite est filtré pour ne montrer que les dépôts qui contiennent des versions du paquetage choisi.

Un des dépôts est un répertoire nommé *package-cache* qui est un sous-répertoire du répertoire courant où vous avez votre image. Quand vous chargez du code depuis un dépôt distant (ou *remote repository*) ou quand vous écrivez du code, une copie est effectuée aussi dans ce répertoire de cache. Il peut être utile si le réseau n'est pas disponible et que vous ayez besoin d'accéder à un paquetage. De plus, si vous avez directement reçu un fichier Monticello (.mcz), par exemple, en pièce jointe dans un courriel, la façon la plus convenable d'y accéder depuis Pharo est de le placer dans le répertoire *package-cache*.

Pour ajouter un nouveau dépôt à la liste, cliquez sur le bouton **[+Repository]** et choisissez le type de dépôt dans le menu flottant. Disons que nous voulons ajouter un dépôt HTTP.

 Ouvrez Monticello, cliquez sur **[+Repository]** et choisissez **HTTP**. Éditez la zone de texte à lire :

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

Ensuite cliquez sur **[Open]** pour ouvrir un navigateur de dépôts ou Repository Browser. Vous devriez voir quelque chose comme la figure 6.15. Sur la gauche, nous voyons une liste de tous les paquetages présents dans le dépôt ; si vous en sélectionnez un, le panneau de droite affichera toutes les versions du paquetage choisi dans ce dépôt.

Si vous choisissez une des versions, vous pourrez naviguer dans son contenu (sans le charger dans votre image) via le bouton **[Browse]**, le charger par le bouton **[Load]** ou encore inspecter les modifications via **[Changes]** qui seront faites à votre image en chargeant la version sélectionnée. Vous pouvez aussi créer une copie grâce au bouton **[Copy]** d'une version d'un paquetage que vous pourriez ensuite écrire dans un autre dépôt.

Comme vous pouvez le voir, les noms des versions contiennent le nom du paquetage, les initiales de l'auteur de la version et un numéro de version. Le nom d'une version est aussi le nom du fichier dans le dépôt. Ne changez jamais ces noms ; le déroulement correct des opérations effectuées dans Monticello dépend d'eux ! Les fichiers de version de Monticello sont simplement des archives compressées et, si vous êtes curieux vous pouvez les décompresser avec un outil de décompression ou *dézippeur*, mais la meilleure

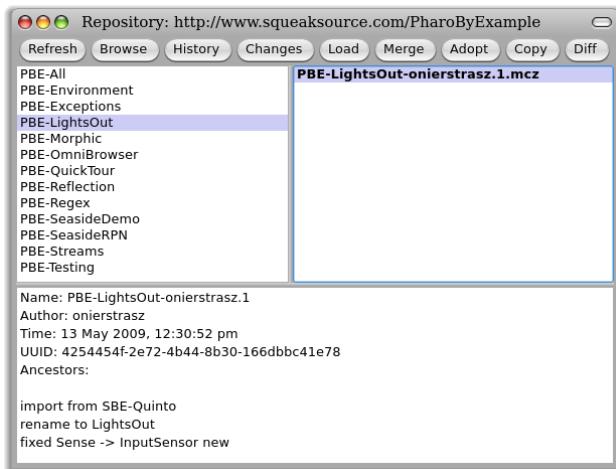


FIGURE 6.15 – Un navigateur de dépôts ou Repository Browser.

façon d’explorer leur contenu consiste à faire appel à Monticello lui-même.

Pour créer un paquetage avec Monticello, vous n’avez que deux choses à faire : écrire du code et le mentionner à Monticello.

 Créez une paquetage appelé PBE-Monticello, et mettez-y une paire de classes, comme vu sur la figure 6.16. Créez une méthode dans une classe existante, par exemple Object, et mettez-la dans le même paquetage que vos classes en utilisant les règles de la page 129 — voir la figure 6.17.

Pour mentionner à Monticello l’existence de votre paquetage, cliquez sur le bouton **+Package** et tapez le nom du paquetage, dans notre cas “PBE”. Monticello ajoutera PBE à sa liste de paquetages ; l’entrée du paquetage sera marquée avec une astérisque pour montrer que la version présente dans votre image n’a pas été encore écrite dans le dépôt. Remarquez que vous devriez avoir maintenant deux paquetages ; un nommé PBE et un autre nommé PBE–Monticello. C’est normal puisque PBE contiendra PBE–Monticello ainsi que tout autre paquetage dont le nom commence par PBE–.

Initialement, le seul dépôt associé à ce paquetage sera votre package cache comme nous pouvons le voir sur la figure 6.18. C’est parfait : vous pouvez toujours sauvegarder le code en l’écrivant dans ce répertoire local de cache. Maintenant, cliquez sur **Save** et vous serez invité à fournir des informations ou *log message* pour la version de ce paquetage, comme le montre la figure 6.19 ; quand vous acceptez le message entré, Monticello sauvegardera votre paquetage et l’astérisque décorant le nom du paquetage du panneau de gauche de Monticello disparaîtra avec le changement du numéro de

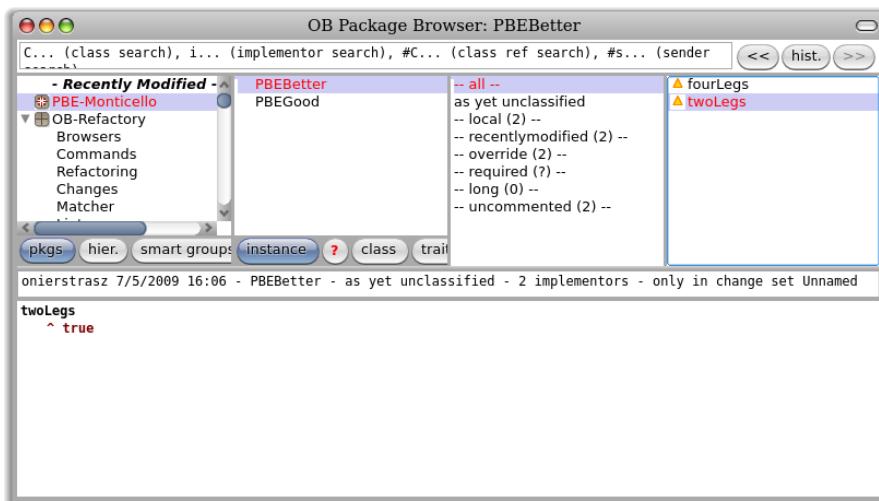


FIGURE 6.16 – Deux classes dans le paquetage (ou package) “PBE”.

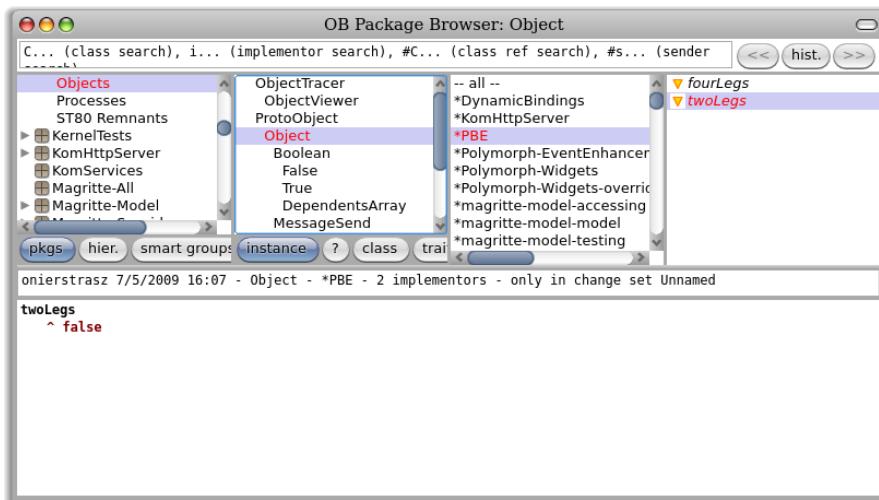


FIGURE 6.17 – Une extension de méthode qui sera aussi incluse dans le paquetage (ou package) “PBE”.



FIGURE 6.18 – Le paquetage PBE pas encore sauvegardé dans Monticello.



FIGURE 6.19 – Fournir un *log* message pour une version d'un paquetage.

version.

Si vous faites ensuite une modification dans votre paquetage,— disons en ajoutant une méthode à une des classes— l’astérisque réapparaîtra pour signaler que vous avez des changements non-sauvegardés. Si vous ouvrez un Repository Brower sur le package cache, vous pouvez choisir une version sauvee et utiliser le bouton **Changes** ou d’autres boutons. Vous pouvez aussi bien sûr sauvegarder la nouvelle version dans ce dépôt; une fois que vous rafraîchissez la vue du dépôt via le bouton **Refresh**, vous devriez voir la même chose que sur la figure 6.20.

Pour sauvegarder notre nouveau paquetage dans un autre dépôt (autre que package-cache), vous avez besoin de vous assurer tout d'abord que Monticello connaît ce dépôt en l'ajoutant si nécessaire. Alors vous pouvez utiliser le bouton **Copy** dans le Repository Browser de package-cache et choisir le dépôt vers lequel le paquetage doit être copié. Vous pouvez aussi associer le dépôt désiré avec le paquetage en sélectionnant **add to package ...** dans le menu contextuel du répertoire accessible en cliquant avec le bouton d'action, comme nous pouvons le voir dans la figure 6.21. Une fois que le paquetage

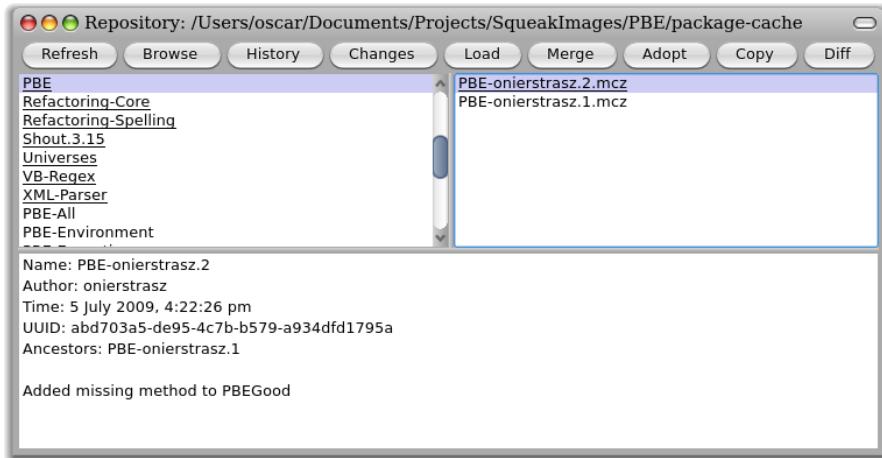


FIGURE 6.20 – Deux versions de notre paquetage sont maintenant le dépôt package cache.

est lié à un dépôt, vous pouvez sauvegarder toute nouvelle version en sélectionnant le dépôt et le paquetage dans le Monticello Brower puis en cliquant sur le bouton **Save**. Bien entendu, vous devez avoir une permission d'écrire dans un dépôt. Le dépôt PharoByExample sur *SqueakSource* est lisible pour tout le monde mais n'est pas ouvert en écriture à tout le monde ; ainsi, si vous essayez d'y sauvegarder quelque chose, vous aurez un message d'erreur. Cependant, vous pouvez créer votre propre dépôt sur *SqueakSource* en utilisant l'interface web de <http://www.squeaksource.com> et en l'utilisant pour sauvegarder votre travail. Ceci est particulièrement utile pour partager votre code avec vos amis ou si vous utilisez plusieurs ordinateurs.

Si vous essayez de sauvegarder dans un répertoire dans lequel vous n'avez pas les droits en écriture, une version sera de toute façon écrite dans le package-cache. Donc vous pourrez corriger en éditant les informations du dépôt (en cliquant avec le bouton d'action dans Monticello Brower) ou en choisissant un dépôt différent puis, en le copiant depuis le navigateur ouvert sur package-cache avec le bouton **Copy**.

6.4 L'inspecteur et l'explorateur

Une des caractéristiques de Smalltalk qui le rend différent de nombreux environnements de programmation est qu'il vous offre une fenêtre sur une monde d'objets vivants et non pas sur un monde de codes statiques. Chacun de ces objets peut être examiné par le programmeur et même changé — bien

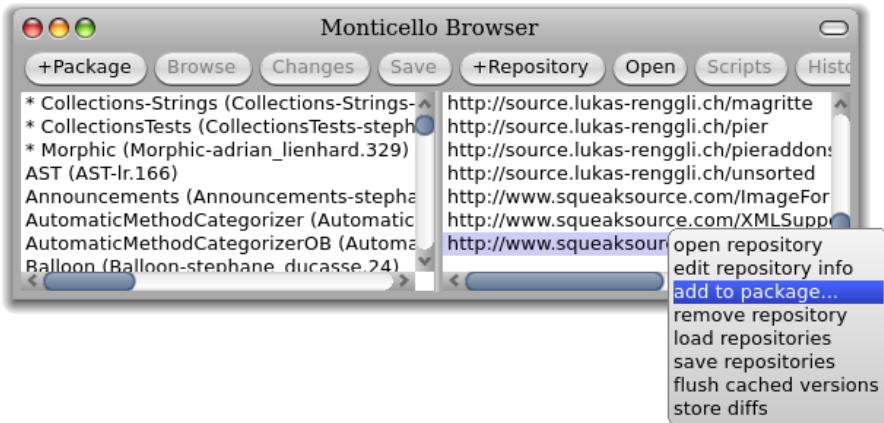


FIGURE 6.21 – Ajouter un dépôt à l'ensemble des dépôts liés au paquetage.

qu'un certain soin doit être apporté lorsqu'il s'agit de modifier des objets bas niveau qui soutiennent le système. De toute façon, expérimentez à votre guise, mais sauvegardez votre image avant !

Inspector

Pour illustrer ce que vous pouvez faire avec l'inspecteur ou Inspector, tapez `TimeStamp now` dans un espace de travail puis cliquez avec le bouton d'action et choisissez `inspect it`.

(Il n'est pas nécessaire de sélectionner le texte avant d'utiliser le menu ; si aucun texte n'est sélectionné, les opérations du menu fonctionnent sur la ligne entière. Vous pouvez aussi entrer `CMD-i` pour `inspect it`.)

Une fenêtre comme celle de la figure 6.22 apparaîtra. Cet inspecteur peut être vu comme une fenêtre sur les états internes d'un objet particulier — dans ce cas, l'instance particulière de `TimeStamp` qui a été créée en évaluant l'expression `TimeStamp now`. La barre de titre de la fenêtre affiche la représentation imprimée de l'objet en cours d'inspection. Si vous sélectionnez `self` dans le panneau supérieur de gauche, le panneau de droite affichera la description de l'objet en chaîne de caractères ou `printstring` de l'objet.

Le panneau de gauche montre une vue arborescente de l'objet avec `self` pour racine. Les variables d'instance peuvent être explorées en cliquant sur les triangles à côté de leurs noms.

Le panneau horizontal inférieur de l'Inspector est un petit espace de travail ou Workspace. C'est utile car dans cette fenêtre, la pseudo-variable `self`

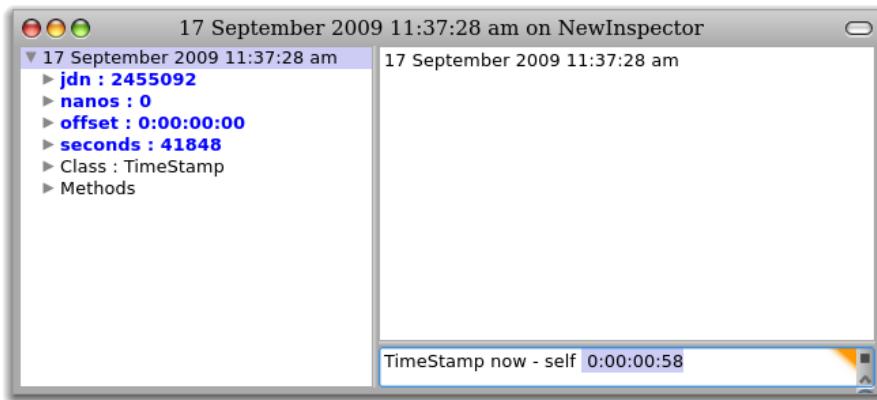


FIGURE 6.22 – Inspecter TimeStamp now.

correspond à l'objet que vous avez sélectionné dans le panneau de gauche. Ainsi, si vous inspectez via `inspect it` l'expression :

```
self -> TimeStamp today
```

dans ce panneau-espace de travail, le résultat sera un objet Duration qui représente l'intervalle temporel entre la date d'aujourd'hui (en anglais, today, le nom du message envoyé) à minuit et le moment où vous avez évalué TimeStamp now et ainsi créé l'objet TimeStamp que vous inspectez. Vous pouvez aussi essayer d'évaluer `TimeStamp now - self`; ce qui vous donnera le temps que vous avez mis à lire la section de ce livre !

En plus de `self`, toutes les variables d'instance de l'objet sont visibles dans le panneau-espace de travail; dès lors vous pouvez les utiliser dans des expressions ou même les affecter. Par exemple, si vous sélectionnez l'objet racine et que vous évaluez `jdn := jdn - 1` dans ce panneau, vous verrez que la valeur de la variable d'instance `jdn` changera réellement et que la valeur de `TimeStamp now - self` sera augmentée d'un jour.

Il y a des variantes spécifiques de l'inspecteur pour les dictionnaires sous-classes de Dictionaries, pour les collections ordonnées sous-classes de OrderedCollections, pour les CompiledMethods (objet des méthodes compilées) et pour quelques autres classes facilitant ainsi l'examen du contenu de ces objets spéciaux.

Object Explorer

L'*Object Explorer* ou explorateur d'objets est sur le plan conceptuel semblable à l'inspecteur mais présente ses informations de manière différente.

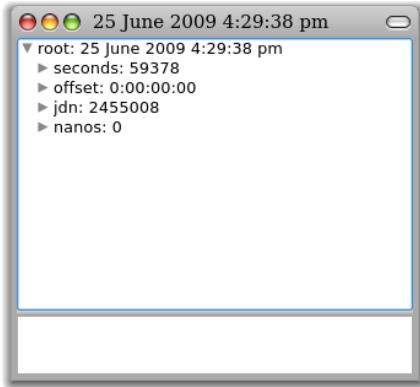


FIGURE 6.23 – ExplorerTimeStamp now.

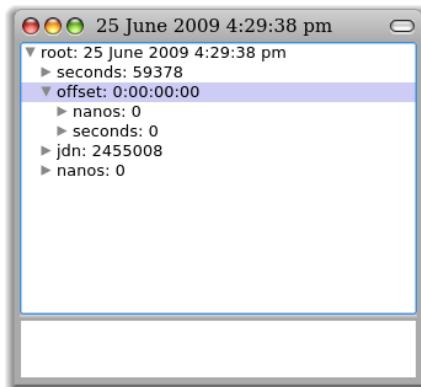


FIGURE 6.24 – Explorer les variables d’instance.

Pour voir la différence, nous allons *explorer* le même objet que nous venons juste d’inspecter.

❶ Sélectionnez `self` dans le panneau gauche de notre inspecteur et choisissez `explore (l)` dans le menu contextuel obtenu en cliquant avec le bouton d’action.

La fenêtre Explorer apparaît alors comme sur la figure 6.23. Si vous cliquez sur le petit triangle à gauche de `root` (racine, en anglais), la vue changera comme dans la figure 6.24 qui nous montre les variables d’instance de l’objet que nous explorons. Cliquez sur le triangle proche d’`offset` et vous verrez ses variables d’instance. L’explorateur est véritablement un outil puissant lorsque vous avez besoin d’explorer une structure hiérarchique complexe—d’où son nom.

Le panneau Workspace de l’Object Explorer fonctionne de façon légèrement différente de celui de l’Inspector. `self` n’est pas lié à l’objet racine `root` mais plutôt à l’objet actuellement sélectionné ; les variables d’instance de l’objet sélectionné sont aussi à portée¹⁰.

Pour comprendre l’importance de l’explorateur, employons-le pour explorer une structure profonde imbriquant beaucoup d’objets.

❷ Évaluez `Object explore` dans un espace de travail.

C’est l’objet qui représente la classe `Object` dans Pharo. Notez que vous pouvez naviguer directement dans les objets représentants le dictionnaire

10. En anglais, vous entendrez souvent le terme “scope” pour désigner la portée des variables d’instance.

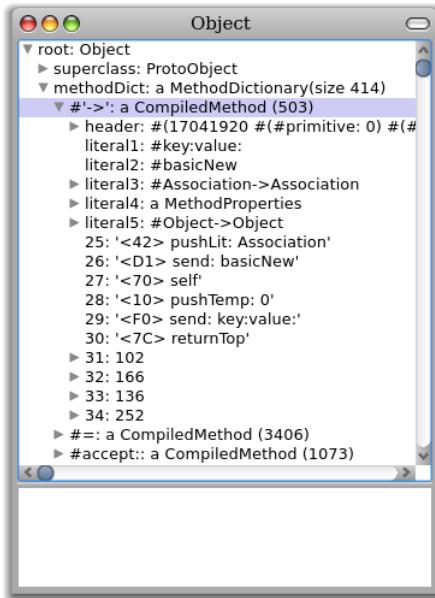


FIGURE 6.25 – Explorer un Object.

de méthodes et même explorer les méthodes compilées de cette classe (voir la figure 6.25).

6.5 Le débogueur

Le débogueur Debugger est sans conteste l'outil le plus puissant dans la suite d'outils de Pharo. Il est non seulement employé pour déboguer c'est-à-dire pour corriger les erreurs mais aussi pour écrire du code nouveau. Pour démontrer la richesse du Debugger, commençons par créer un *bug* !

Via le navigateur, ajouter la méthode suivante dans la classe String :

Méthode 6.1 – Une méthode boguée

suffix

"disons que je suis un nom de fichier et que je fournis mon suffixe, la partie suivant le dernier point"

| dot dotPosition |

dot := FileDirectory dot.

dotPosition := (self size to: 1 by: -1) detect: [:i | (self at: i) = dot].

↑ self copyFrom: dotPosition to: self size

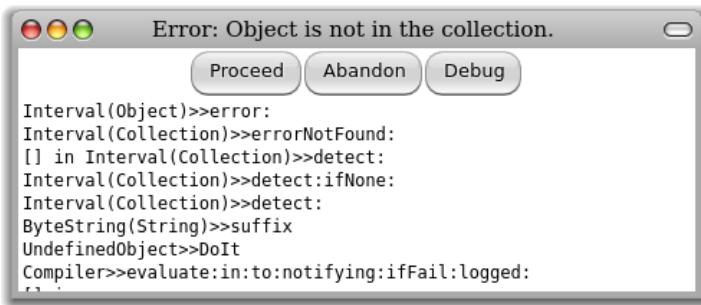


FIGURE 6.26 – Un PreDebugWindow nous alarme de la présence d'un bug.

Bien sûr, nous sommes certain qu'une méthode si triviale fonctionnera. Ainsi plutôt que d'écrire un test *SUnit* (que nous verrons dans le chapitre 7), nous entrons simplement 'readme.txt' suffix dans un Workspace et nous en imprimons l'exécution via `print it (p)`. Quelle surprise ! Au lieu d'obtenir la réponse attendu 'txt', une notification PreDebugWindow s'ouvre comme sur la figure 6.26.

Le PreDebugWindow nous indique dans sa barre de titre qu'une erreur s'est produite et nous affiche une trace de la pile d'exécution ou *stack trace* des messages qui ont conduit à l'erreur. En démarrant depuis la base de la trace (en haut de la liste), `UndefinedObject»DoIt` représente le code qui vient d'être compilé et lancé quand nous avons sélectionné 'readme.txt' suffix dans notre espace de travail et que nous avons demandé à Pharo de l'imprimer sur cet espace par `print it`. Ce code envoia, bien sûr, le message `suffix` à l'objet `ByteString ('readme.txt')`. S'en suit l'exécution de la méthode `suffix` héritée de la classe `String` ; toutes ces informations sont encodées dans la ligne suivante de la trace, `ByteString(String)>>suffix`. En visitant la pile, nous pouvons voir que `suffix` envoie à son tour `detect:...` et `detect:ifNone` émet `errorNotFound`.

Pour trouver *pourquoi* le point (dot) n'a pas été trouvé, nous avons besoin du débogueur lui-même ; dès lors, il suffit de cliquer sur le bouton `Debug`.

Le débogueur est visible sur la figure 6.27 ; il semble intimidant au début, mais il est assez facile à utiliser. La barre de titre et le panneau supérieur sont très similaires à ceux que nous avons vu dans le notificateur PreDebugWindow. Cependant, le Debugger combine la trace de la pile avec un navigateur de méthode, ainsi quand vous sélectionnez une ligne dans le *stack trace*, la méthode correspondante s'affiche dans le panneau inférieur. Vous devez absolument comprendre que l'exécution qui a causée l'erreur est toujours dans l'image mais dans un état suspendu. Chaque ligne de la trace représente une tranche de la pile d'exécution qui contient toutes les informations nécessaires

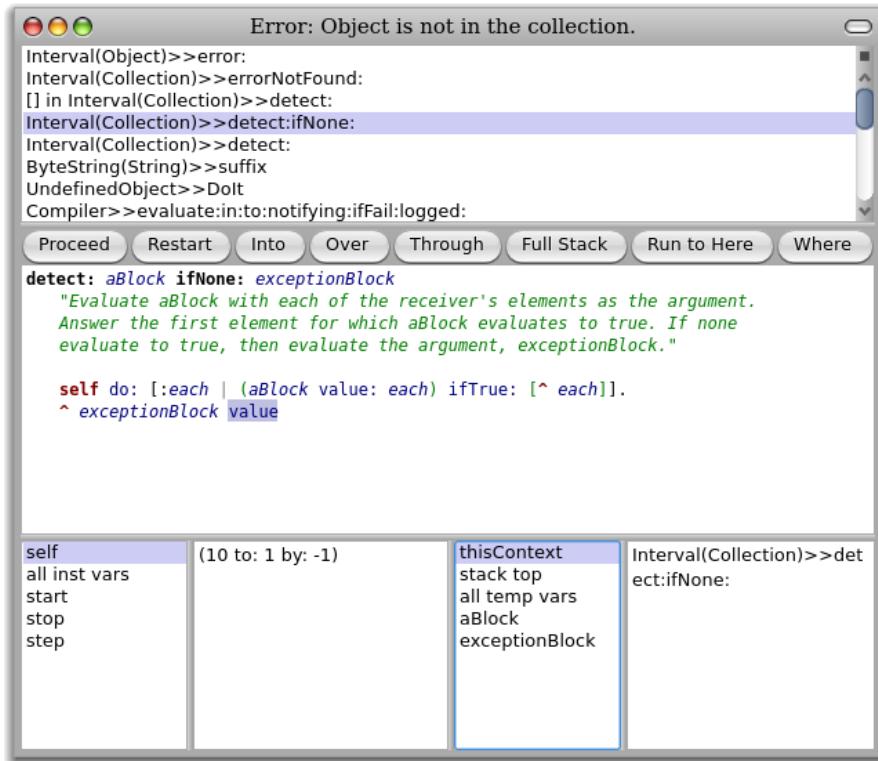


FIGURE 6.27 – Le débogueur.

pour poursuivre l'exécution. Ceci comprend tous les objets impliqués dans le calcul, avec leurs variables d'instance et toutes les variables temporaires des méthodes exécutées.

Dans la figure 6.27 nous avons sélectionné la méthode `detect:ifNone:` dans le panneau supérieur. Le corps de la méthode est affiché dans le panneau central ; la sélection bleue entourant le message `value` nous montre que la méthode actuelle a envoyé le message `value` et attend une réponse.

Les quatre panneaux inférieurs du débogueur sont véritablement deux mini-inspecteurs (sans panneaux-espace de travail). L'inspecteur de gauche affiche l'objet actuel, c'est-à-dire l'objet nommé `self` dans le panneau central. En sélectionnant différentes lignes de la pile, l'identité de `self` peut changer ainsi que le contenu de l'inspecteur du `self`. Si vous cliquez sur `self` dans le panneau inférieur gauche, vous verrez que `self` est un intervalle (`10 to: 1 by -1`), ce à quoi nous devions nous attendre. Les panneaux Workspace ne sont pas nécessaires dans les mini-inspecteurs de Debugger car toutes les variables

sont aussi à portée dans le panneau de méthode ; vous pouvez entrer et évaluer à loisir n'importe quelle expression. Vous pouvez toujours annuler vos changements en utilisant `cancel (l)` dans le menu ou en tapant `CMD-l`.

L'inspecteur de droite affiche les variables temporaires du contexte courant. Dans la figure 6.27, `value` a été envoyé au paramètre `exceptionBlock`.

Comme nous pouvons le voir sur la méthode plus bas sur la pile, exceptionBlock est [self errorNotFound: ...]. Il n'y a donc rien de surprenant à voir le message d'erreur correspondant.

Du coup, si vous voulez ouvrir un inspecteur complet sur une des variables affichées dans les mini-inspecteurs, vous n'avez qu'à double-cliquer sur le nom de la variable ou alors sélectionner le nom de la variable et cliquez avec le bouton d'action pour choisir `inspect (i)` ou `explore (l)` : utile si vous voulez suivre le changement d'une variable lorsque vous exécutez un autre code.

En revenant sur le panneau de méthode, nous voyons que nous nous attendions à trouver `dot` dans la chaîne de caractère 'readme.txt' à l'antépénultième (soit l'avant-avant-dernière) ligne de la méthode et que l'exécution n'aurait jamais du atteindre la dernière ligne. Pharo ne nous permet pas de lancer une exécution en arrière mais il permet de relancer une méthode, ce qui marche parfaitement dans notre code qui ne mute pas les objets mais qui en crée de nouveaux.

❶ Cliquez sur le bouton `Restart` et vous verrez que le locus de l'exécution retournera dans l'état premier de la méthode courante. La sélection bleue englobe maintenant le message suivant à envoyer : `do:` (voir la figure 6.28).

Les boutons `Into` et `Over` offrent deux façons différentes de parcourir l'exécution pas-à-pas. Si vous cliquez sur le bouton `Over`, Pharo exécutera sauf erreur l'envoi du message actuel (dans notre cas `do:`) d'un pas (en anglais, `step`). Ainsi `Over` nous amènera sur le prochain message à envoyer dans la méthode courante. Ici nous passons à `value` — c'est exactement l'endroit d'où nous avons démarré et ça ne nous aide pas beaucoup. En fait, nous avons besoin de trouver pourquoi `do:` ne trouve pas le caractère que nous cherchons.

❷ Après avoir cliqué sur le bouton `Over`, cliquez sur le bouton `Restart` pour obtenir la situation vue dans la figure 6.28.

❸ Cliquez sur le bouton `Into` ; Pharo ira dans la méthode correspondante au message surligné par la sélection bleue ; dans ce cas, `Collection»do:`.

Cependant, ceci ne nous aide pas plus : nous pouvons être confiant dans le fait que la méthode `Collection»do:` n'est pas erronée. Le bug est plutôt dans ce que nous demandons à Pharo de faire. `Through` est le bouton approprié à

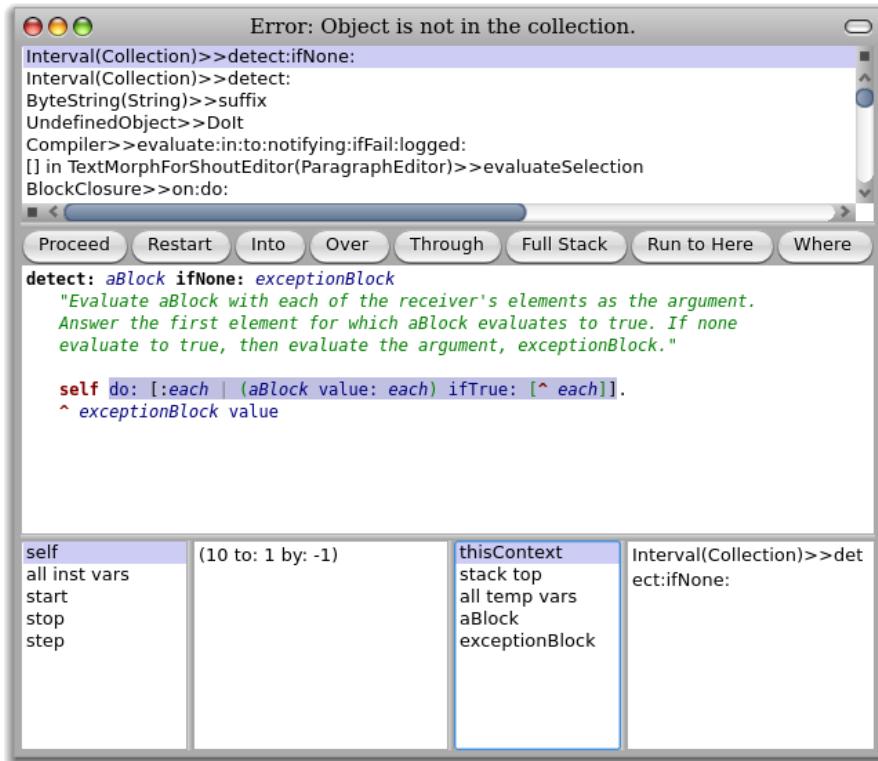


FIGURE 6.28 – Debugger après avoir relancé la méthode detect: ifNone:.

ce cas : nous voulons ignorer les détails de do: lui-même et se focaliser sur l'exécution du bloc argument.

Sélectionnez encore la méthode detect:ifNone: et cliquez sur le bouton **Restart** pour revenir à l'état de la figure 6.28. Cliquez maintenant sur le bouton **Through** plusieurs fois. Sélectionnez each dans le mini-inspecteur de contexte (en bas à droite). Vous remarquez que each décompte depuis 10 au fur et à mesure de l'exécution de la méthode do:.

Quand each est 7, nous nous attendons à ce que le bloc ifTrue: soit exécuté, mais ce n'est pas le cas. Pour voir ce qui ne marche pas, allez dans l'exécution de value: par le bouton **Into** comme illustré par la figure 6.29.

Après avoir cliqué sur le bouton **Into**, nous nous trouvons dans la position illustrée par la figure 6.30. Tout d'abord, il semble que nous soyons revenus à la méthode suffix mais c'est parce que nous exécutons désormais le bloc que suffix fourni en argument à detect:. Si vous sélectionnez dot dans l'inspec-

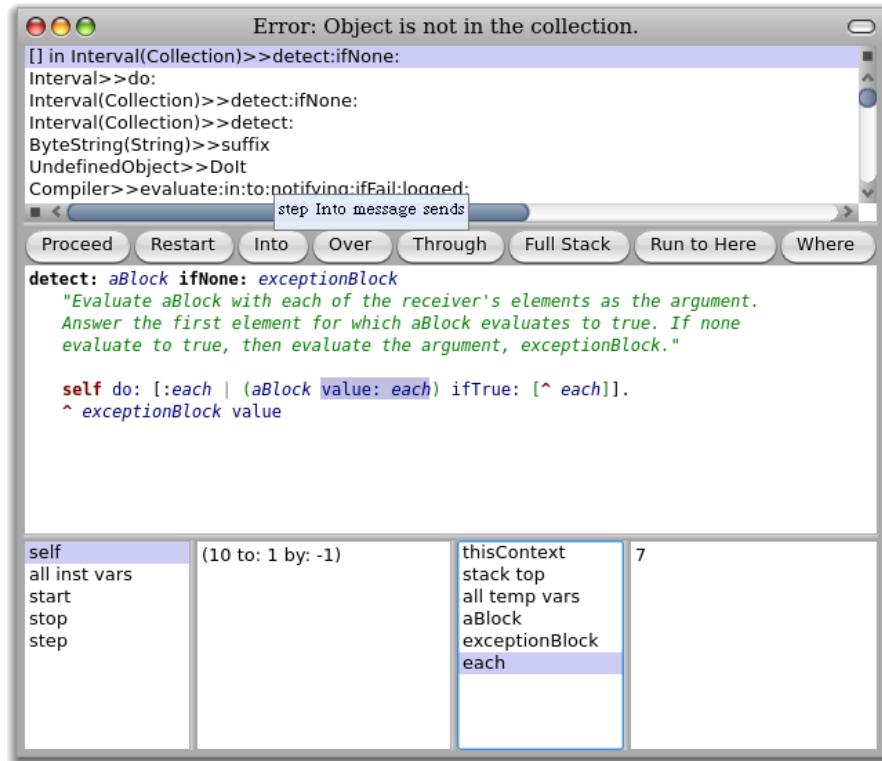


FIGURE 6.29 – Debugger après un *pas* dans la méthode `do:` plusieurs fois grâce au bouton `Through`.

teur contextuel, vous verrez que sa valeur est '!'. Vous constatez maintenant qu'ils ne sont pas égaux : le septième caractère de 'readme.txt' est pourtant un objet Character (donc un caractère), alors que dot est un String (c-à-d. une chaîne de caractères).

Maintenant nous pouvons mettre le doigt sur le bug, la correction¹¹ est évidente : nous devons convertir dot en un caractère avant de recommencer la recherche.

Changez le code directement dans le débogueur de façon à ce que l'affection est la forme `dot := FileDirectory dot first` et acceptez la modification.

Puisque nous sommes en train d'exécuter le code dans un bloc à l'intérieur d'un `detect :`, plusieurs trames de la pile devront être abandonnées de manière à valider le changement. Pharo nous demande si c'est ce que nous

11. En anglais, nous parlons de *bug fix*.

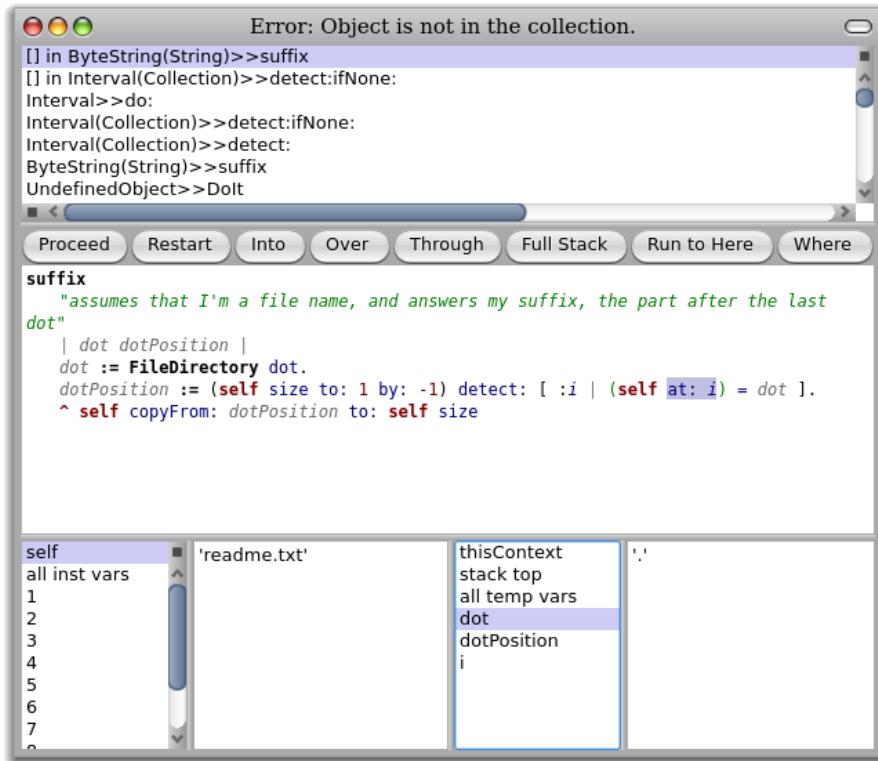


FIGURE 6.30 – Debugger montrant pourquoi 'readme.txt' at: 7 n'est pas égal à dot.

voulons (voir la figure 6.31) et, à condition de cliquer sur `yes`, Pharo sauvegardera (et compilera) la nouvelle méthode.

L'évaluation de l'expression `'readme.txt' suffix` sera complète et affichera la réponse `'.txt'`.

Est-ce pour autant une réponse correcte ? Malheureusement nous ne pouvons répondre avec certitude. Le suffixe devrait-il être `.txt` ou `txt` ? Le commentaire dans la méthode `suffix` n'est pas très précis. La façon d'éviter ce type de problème est d'écrire un test SUnit pour définir la réponse.

Méthode 6.2 – Un simple test pour la méthode `suffix`

```
testSuffixFound
    self assert: 'readme.txt' suffix = 'txt'
```

L'effort requis pour ce faire est à peine plus important que celui qui consiste à lancer le même test dans un espace de travail ; l'avantage de SUnit

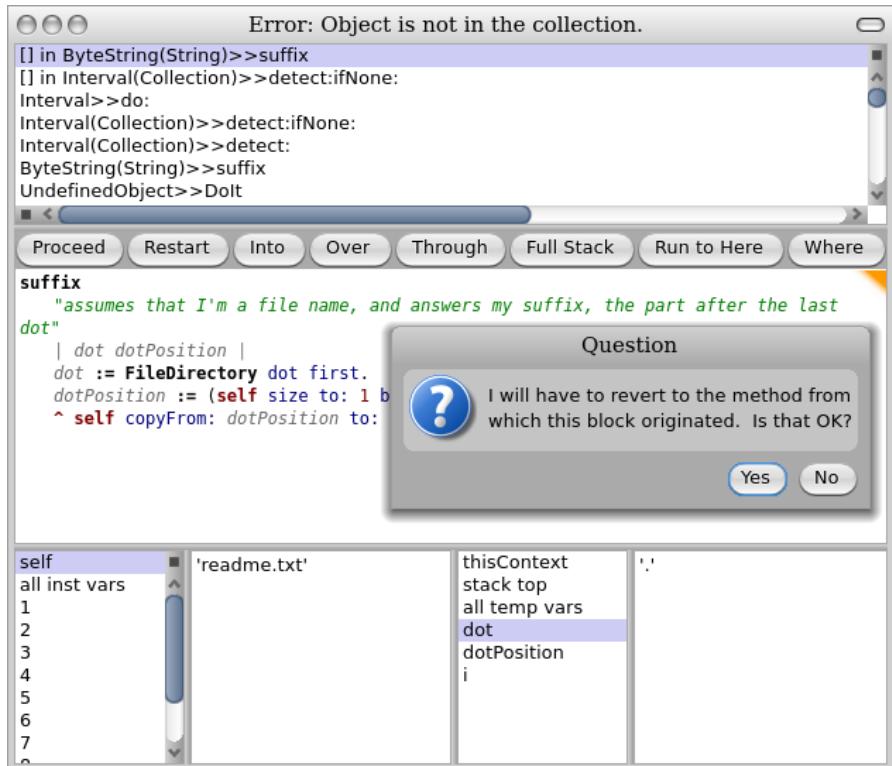


FIGURE 6.31 – Changer la méthode `suffix` dans Debugger : demander la confirmation de la sortie du bloc interne. La boîte d'alerte nous dit : “Je devrais revenir à la méthode d'où ce bloc est originaire. Est-ce bon ?”.

est de sauvegarder ce test sous la forme d'une documentation exécutable et de faciliter l'accessibilité des usagers de la méthode. En plus, si vous ajoutez la méthode 6.2 à la classe `StringTest` et que vous lancez ce test avec SUnit, vous pouvez très facilement revenir pour déboguer l'actuelle erreur. SUnit ouvre Debugger sur l'assertion fautive mais là vous avez simplement besoin de descendre d'une ligne dans la liste-pile, redémarrez le test avec le bouton `Restart` et allez dans la méthode `suffix` par le bouton `Into`. Vous pouvez alors corriger l'erreur, comme nous l'avons fait dans la figure 6.32. Il s'agit maintenant de cliquer sur le bouton `Run Failures` dans le SUnit Test Runner et de se voir confirmer que le test passe (en anglais, *pass*) normalement. Rapide, non ?

Voici un meilleur test :

Méthode 6.3 – Un meilleur test pour la méthode `suffix`

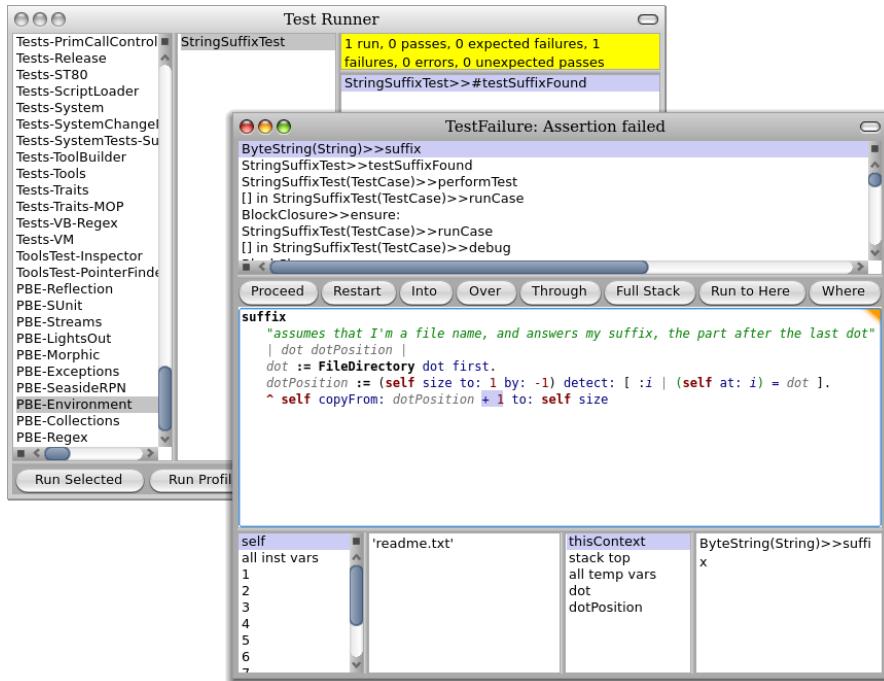


FIGURE 6.32 – Changer la méthode suffix dans Debugger : corriger l’erreur du plus-d’un-point après l’assertion fautive SUnit.

```
testSuffixFound
self assert: 'readme.txt' suffix = 'txt'.
self assert: 'read.me.txt' suffix = 'txt'
```

Pourquoi ce test est-il meilleur ? Simplement parce que nous informons le lecteur de ce que la méthode devrait faire s'il y a plus d'un point dans la chaîne de caractères, instance de String.

Il y a d'autres moyens d'obtenir une fenêtre de débogueur en plus de ceux qui consistent à capturer une erreur effective ou à faire une assertion fautive (ou *assertion failures*). Si vous exécutez le code qui conduit à une boucle infinie, vous pouvez l'interrompre et ouvrir un débogueur durant le calcul en tapant CMD-.¹² Vous pouvez aussi éditer simplement le code suspect en insérant l'expression self halt. Ainsi, par exemple, nous pourrions éditer la méthode suffix comme suit :

12. Sachez que vous pouvez ouvrir un débogueur d'urgence n'importe quand en tapant CMD-SHIFT.

Méthode 6.4 – Insérer une pause par halt dans la méthode suffix.

suffix

"disons que je suis un nom de fichier et que je fournis mon suffixe, la partie suivant le dernier point"

```
| 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
```

Quand nous lançons cette méthode, l'exécution de self halt ouvre un n^{otificateur} ou *pre-debugger* d'où nous pouvons continuer en cliquant sur **proceed** ou déboguer et explorer l'état des variables, parcourir pas-à-pas la pile d'exécution et éditer le code.

C'est tout pour le débogueur mais nous n'en avons pas fini avec la méthode suffix. Le bug initial aurait dû vous faire réaliser que s'il n'y a pas de point dans la chaîne cible la méthode suffix lèvera une erreur. Ce n'est pas le comportement que nous voulons. Ajoutons ainsi un second test pour signaler ce qu'il pourrait arriver dans ce cas.

Méthode 6.5 – Un second test pour la méthode suffix : la cible n'a pas de suffixe

```
testSuffixNotFound
self assert: 'readme' suffix = "
```

 Ajoutez la méthode 6.5 à la suite de tests dans la classe StringTest et observez l'erreur levée par le test. Entrez dans Debugger en sélectionnant le test erroné dans SUnit puis éditez le code de façon à passer normalement le test (donc sans erreur). La méthode la plus facile et la plus claire consiste à remplacer le message detect: par detect: ifNone:¹³ où le second argument un bloc qui retourne tout simplement une chaîne.

Nous en apprendrons plus sur SUnit dans le chapitre 7.

6.6 Le navigateur de processus

Smalltalk est un système multitâche : plusieurs processus légers (aussi connu sous le nom de *threads*) tournent simultanément dans votre image. Dans l'avenir la machine virtuelle de Pharo bénéficiera davantage des multiprocesseurs lorsqu'ils seront disponibles, mais le partage d'accès est actuellement programmé sur le principe de tranches temporelles (ou *time-slice*).

13. En anglais, *if none* signifie "s'il n'y a rien".

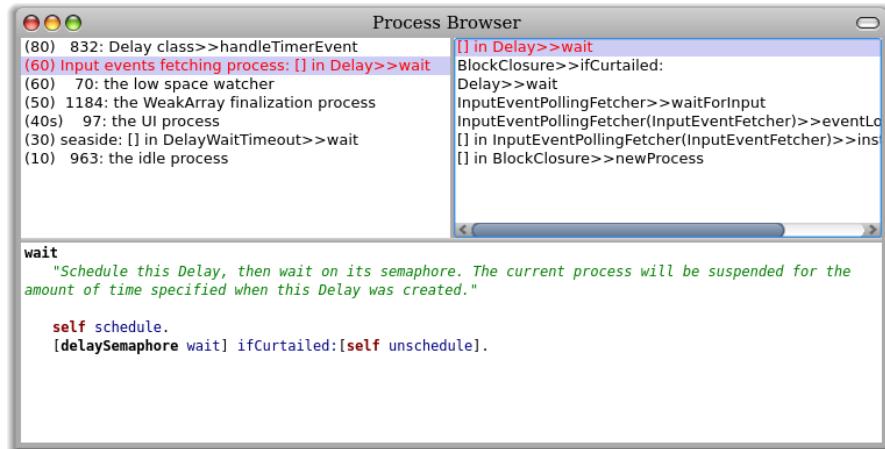


FIGURE 6.33 – Le Process Browser.

Le Process Browser ou navigateur de processus est un cousin de Debugger qui vous permet d’observer les divers processus tournant dans le système Pharo. La figure 6.33 nous en présente une capture d’écran. Le panneau supérieur gauche liste tous les processus présents dans Pharo, dans l’ordre de leur priorité depuis le *timer interrupt watcher* (système de surveillance d’interruption d’horloge) de priorité 80 au *idle process* ou processus inactif du système de priorité 10. Bien sûr, sur un système mono-processeur, le seul processus pouvant être lancé en phase de visualisation est le *UI*¹⁴ process ou processus graphique ; tous les autres processus seront en attente d’un quelconque événement. Par défaut, l’affichage des processus est statique ; il peut être mis à jour en cliquant avec le bouton d’action et en sélectionnant turn on auto-update (a).

Si vous sélectionnez un processus dans le panneau supérieur gauche, le panneau de droite affichera son *stack trace* tout comme le fait le débogueur. Si vous en sélectionnez un, la méthode correspondante est affichée dans le panneau inférieur. Le Process Browser n’est pas équipé de mini-inspecteurs pour *self* et *thisContext* mais cliquer avec le bouton d’action sur les tranches de la pile offre une fonctionnalité équivalente.

6.7 Trouver les méthodes

Il y a deux outils dans Pharo pour vous aider à trouver des messages. Ils diffèrent en termes d’interface et de fonctionnalité.

14. UI désigne *User Interface* ; en français, interface utilisateur.

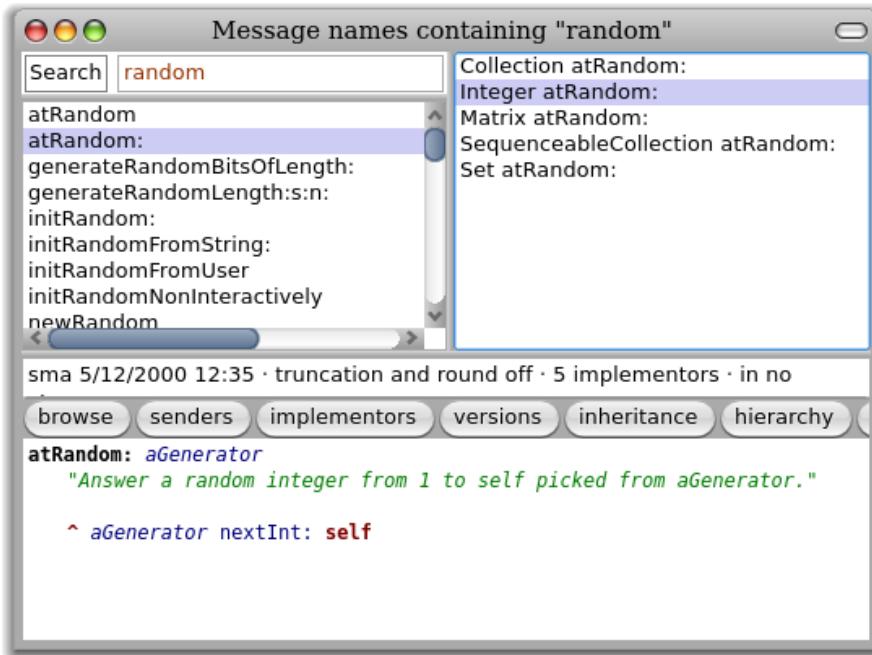


FIGURE 6.34 – Le Message Names Browser montrant toutes les méthodes contenant le sous-élément de chaîne random dans leur sélecteur.

Le *Method Finder* (ou chercheur de méthodes) a été longuement décrit dans la section 1.9 ; vous pouvez l'utiliser pour trouver des méthodes par leur nom ou leur fonction. Cependant, pour observer le corps d'une méthode, le Method Finder ouvre un nouveau navigateur. Cela peut vite devenir pénible.

La fonctionnalité de recherche du Message Names Browser ou navigateur de *noms de messages* est plus limitante : vous entrez un morceau d'un sélecteur de message dans la boîte de recherche et le navigateur liste toutes les méthodes contenant ce fragment dans leurs noms, comme nous pouvons le voir dans la figure 6.34. Cependant, c'est un navigateur complet : si vous sélectionnez un des noms dans le panneau de gauche, toutes les méthodes ayant ce nom seront listées dans celui de droite et vous pourrez alors naviguer dans le panneau inférieur. Le Message Names Browser a une barre de bouton, comme le Browser, pouvant être utilisée pour ouvrir d'autres navigateurs sur la méthode choisie ou sur sa classe.

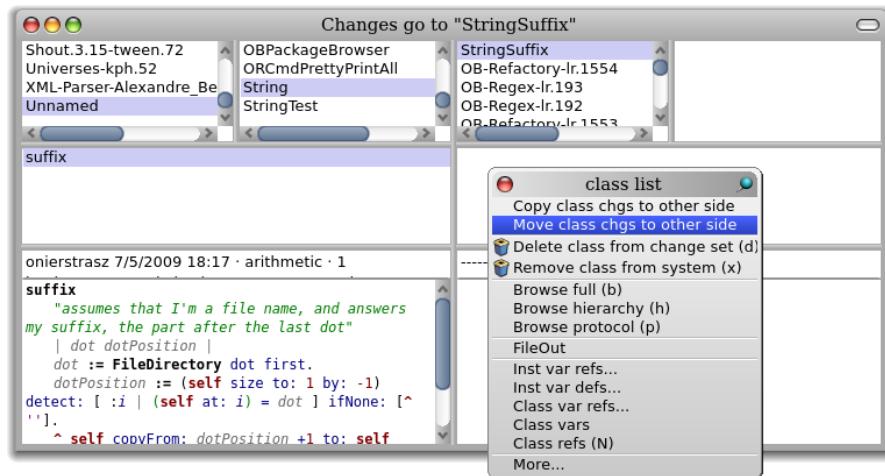


FIGURE 6.35 – Le Change Sorter.

6.8 Change set et son gestionnaire Change Sorter

Chaque fois que vous travaillez dans Pharo, tous les changements que vous effectuez sur les méthodes et les classes sont enregistrés dans un *change set* (traduisible par “ensemble des modifications”). Ceci inclus la création de nouvelles classes, le renommage de classes, le changement de catégories, l’ajout de méthodes dans une classe existante—en bref, tout ce qui a un impact sur le système. Cependant, les exécutions arbitraires avec *do it* ne sont pas incluses ; par exemple, si vous créez une nouvelle variable globale par affectation dans un espace de travail, la création de variable ne sera pas dans un *change set*.

A tout moment, beaucoup de *change sets* existent, mais un seul d’entre eux—*ChangeSet current*—collecte les changements qui sont en cours dans l’image actuelle. Vous pouvez voir quel *change set* est le *change set* actuel et vous pouvez examiner tous les *change sets* en utilisant le Change Sorter (ou trieur de *change set*) disponible dans le menu principal dans *World > Tools ... > Change Sorter*.

La figure 6.35 nous montre ce navigateur. La barre de titre affiche le *change set* actuel et ce *change set* est sélectionné quand le navigateur s’ouvre.

Les autres *change sets* peuvent être choisis dans le panneau supérieur de gauche ; le menu contextuel accessible via le bouton jaune vous permet de faire de n’importe quel *change set* votre *change set* actuel ou de créer un nouveau *change set*. Le panneau supérieur de droite liste toutes les classes

(accompagnées de leurs catégories) affectées par le *change set* sélectionné. Sélectionner une des classes affiche les noms de ses méthodes qui sont aussi dans le *change set* (*pas* toutes les méthodes de la classe) dans le panneau central et sélectionner un de ces noms de méthodes affiche sa définition dans le panneau inférieur. Remarquez que le navigateur ne montre *pas* si la création de la classe elle-même fait partie du *change set* bien que cette information soit stockée dans la structure de l'objet qui est utilisé pour représenter le *change set*.

Le Change Sorter vous permet d'effacer des classes et des méthodes du *change set* en cliquant avec le bouton d'action sur les éléments correspondants.

Le Change Sorter vous permet de voir simultanément deux *change sets* : un *change set* à gauche et un autre à droite. Cette disposition offre les principales fonctions du Change Sorter telles que la possibilité de déplacer ou copier les changements d'un *change set* à un autre, comme nous pouvons le voir sur la figure 6.35, dans le menu contextuel accessible en cliquant avec le bouton d'action. Nous pouvons aussi copier des méthodes d'un pan à un autre.

Vous pouvez vous demander pourquoi vous devez accorder de l'importance à la composition d'un *change set* : la réponse est que les *change sets* fournissent un mécanisme simple pour exporter du code depuis Pharo vers le système de fichiers d'où il peut être importé dans une autre image Pharo ou vers un autre Smalltalk que Pharo. L'exportation de *change set* est connu sous le nom "filing-out" et peut être réalisé en utilisant le menu contextuel obtenu en cliquant avec le bouton d'action sur n'importe quel *change set*, classe ou méthode dans n'importe quel navigateur. Des exportations (ou fileouts) répétées créent une nouvelle version du fichier mais les *change sets* ne sont pas un outil de versionnage (gestion de versions) comme peut l'être Monticello : ils ne conservent pas les dépendances.

Avant l'avènement de Monticello, les *change sets* étaient la technique majeure d'échange de code entre les Smalltalkiens.

Ils ont l'avantage d'être simples et relativement portables (le fichier d'exportation n'est qu'un fichier texte ; *nous ne vous recommandons pas* d'éditer ce fichier avec un éditeur de texte). Il est assez facile aussi de créer un *change set* qui modifie beaucoup de parties différentes du système sans aucun rapport entre elles — ce pour quoi Monticello n'est pas encore équipé.

Le principal inconvénient des *change sets* par rapport aux paquetages Monticello est leur absence de notion de dépendances. Une exportation de *change set* est un ensemble d'*actions* transformant n'importe quelle image dans laquelle elle est chargée. Pour en charger avec succès, l'image doit être dans un état approprié. Par exemple, le *change set* pourrait contenir une action pour ajouter une méthode à une classe ; ceci ne peut être fait que si la

classe est déjà définie dans l'image. De même, le *change set* pourrait renommer ou re-catégoriser une classe, ce qui ne fonctionnerait évidemment que si la classe est présente dans l'image ; les méthodes pourraient utiliser des variables d'instance déclarées lors de l'exportation mais inexistantes dans l'image dans laquelle elles sont importées. Le problème est que les *change sets* ne contiennent pas explicitement les conditions sous lesquelles ils peuvent être chargés : le fichier en cours de chargement marche *au petit bonheur la chance* jusqu'à ce qu'un message d'erreur énigmatique et un *stack trace* surviennent quand les choses tournent mal. Même si le fichier fonctionne, un *change set* peut annuler silencieusement un changement fait par un autre.

à l'inverse, les paquetages (dits aussi packages) de Monticello représentent le code d'une manière déclarative : ils décrivent l'état que l'image devrait avoir une fois le chargement effectué. Ceci permet à Monticello de vous avertir des conflits (quand deux paquetages ont des objectifs incompatibles) et vous permet de charger une série de paquetages dans un ordre de dépendances.

Malgré de ces imperfections, les *change sets* restent utiles ; vous pouvez, en particulier, en trouver sur Internet pour en observer le contenu, voire les utiliser. Maintenant que nous avons vu comment exporter des *change sets* avec le Change Sorter, nous allons voir comment les importer. Cette étape requiert l'usage d'un autre outil, le File List Browser.

6.9 Le navigateur de fichiers File List Browser

Le navigateur de fichiers ou File List Browser est en réalité un outil générique pour naviguer au travers d'un système de fichiers (et aussi sur des serveurs FTP) depuis Pharo. Vous pouvez l'ouvrir depuis le menu **World > Tools ... > File Browser**. Ce que vous y voyez dépend bien sûr du contenu de votre système de fichiers local mais une vue typique du navigateur est illustrée sur la figure 6.36.

Quand vous ouvrez un navigateur de fichiers, il pointera tout d'abord le répertoire actuel, *c-à-d.* celui depuis lequel vous avez démarré Pharo. La barre de titre montre le chemin de ce répertoire. Le panneau de gauche est utilisé pour naviguer dans le système de fichiers de manière conventionnelle.

Quand un répertoire est sélectionné, les fichiers qu'ils contiennent (mais pas les répertoires) sont affichés sur la droite. Cette liste de fichiers peut être filtrée en entrant dans la petite boîte dans la zone supérieure gauche de la fenêtre un modèle de filtrage ou *pattern* dans le style Unix. Initialement, ce *pattern* est `*`, ce qui est égal à l'ensemble des fichiers, mais vous pouvez entrer une chaîne de caractères différente et l'accepter pour changer ce filtre.

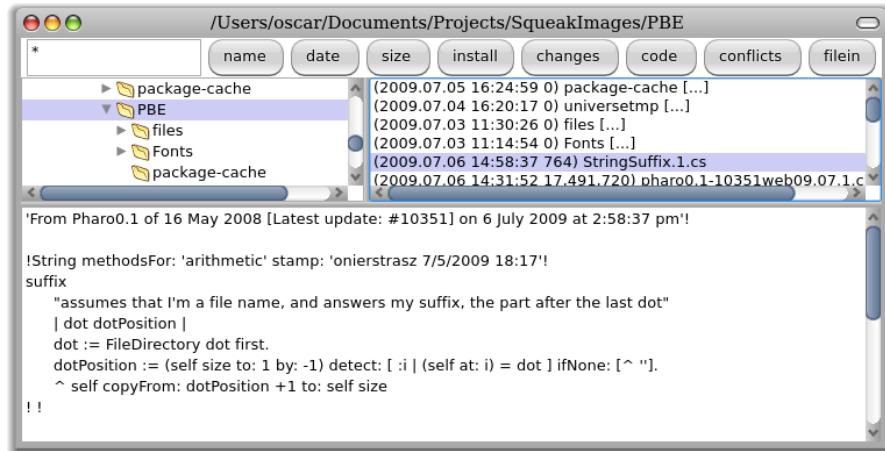


FIGURE 6.36 – Le File List Browser.

Notez qu'un * est implicitement joint ou pré-joint au *pattern* que vous entrez. L'ordre de tri des fichiers peut être modifié via les boutons [name] (par nom), [date] (par date) et [size] (par taille). Le reste des boutons dépend du nom du fichier sélectionné dans le navigateur. Dans la figure 6.36, le nom des fichiers a le suffixe .cs, donc le navigateur suppose qu'il s'agit de *change set* et ajoute les boutons [install] (pour *l'importer* dans un nouveau *change set* dont le nom est dérivé de celui du fichier), [changes] (pour naviguer dans le changement du fichier), [code] (pour l'examiner) et [filein] (pour charger le code dans le *change set actuel*). Vous pourriez penser que le bouton [conflicts] vous informerait des modifications du *change set* pouvant être source de conflits dans le code existant dans l'image mais ça n'est pas le cas. En réalité, il vérifie juste d'éventuels problèmes dans le fichier (tel que la présence de sauts de lignes ou *linefeeds*) pouvant indiquer qu'il ne pourrait pas être proprement chargé.

Puisque le choix des boutons affichés dépend du *nom* du fichier et non de son contenu, parfois le bouton dont vous avez besoin pourrait ne pas être affiché. De toutes façons, le jeu complet des options est toujours disponible grâce à l'option *more ...* du menu contextuel accessible en cliquant avec le bouton d'action, ainsi vous pouvez facilement contourner ce problème.

Le bouton [code] est certainement le plus utile pour travailler avec les *change sets*; il ouvre un navigateur sur le contenu du fichier. Un exemple est présenté dans la figure 6.37. Le File Contents Browser est proche d'un Browser à l'exception des catégories; seuls les classes, les protocoles et les méthodes sont présentés. Pour chaque classe, ce navigateur précise si la classe existe déjà dans le système ou non et si elle est définie dans le fichier (mais

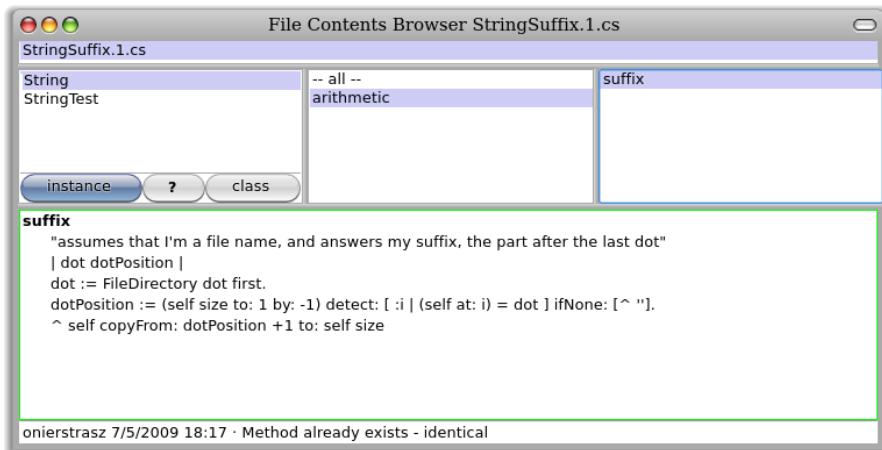


FIGURE 6.37 – Le File Contents Browser.

pas si les définitions sont identiques). Il affichera les méthodes de chaque classe ainsi que les différences entre la version actuelle et celle dans le fichier ; ce que nous montre la figure 6.37. Les options du menu contextuel de chacun des quatre panneaux supérieurs vous permettra de charger (en anglais, *file in*) le *change set* complet, la classe, le protocole ou la méthode correspondante.

6.10 En Smalltalk, pas de perte de codes

Pharo peut parfois planter : en tant que système expérimental, Pharo vous permet de changer n’importe quoi dont les éléments vitaux qui font que Pharo fonctionne !

Pour crasher malicieusement Pharo, évaluez Object become: nil.

La bonne nouvelle est que vous ne perdez jamais votre travail, même si votre image plante et revient dans l’état de la dernière version sauvegardée il y a de cela peut être des heures. La raison en est que tout code exécuté est sauvégarde dans le fichier *.changes*. Tout ceci inclut les expressions que vous évaluez dans un espace de travail Workspace, tout comme le code que vous ajoutez à une classe en la programmant.

Ainsi, voici les instructions permettant de retrouver ce code. Il n’est pas utile de lire ce qui suit tant que vous n’en avez pas besoin. Cependant, quand vous en aurez besoin, vous saurez où le trouver.

Dans le pire des cas, vous pouvez toujours utiliser un éditeur de texte sur le fichier `.changes`, mais quand celui-ci pèse plusieurs mégaoctets, cette technique pourrait s'avérer lente et peu recommandable. Pharo vous offre de meilleures façons de vous en sortir.

La pêche au code

Redémarrez Pharo depuis la sauvegarde (ou *snapshot*) la plus récente et sélectionnez `World > Tools ... > Recover lost changes`.

Vous aurez ainsi l'opportunité de décider jusqu'où vous souhaitez revenir dans l'historique. Normalement, naviguer dans les changements depuis la dernière sauvegarde est suffisant (et vous pouvez obtenir le même effet en éditant `ChangeList browseRecent: 2000` en tâtonnant sur le chiffre empirique `2000`).

Une fois que vous avez le navigateur des modifications récentes nommé *Recent Changes Browser* vous affichant les changements, disons, depuis votre dernière sauvegarde, vous aurez une liste de tout ce que vous avez effectué dans Pharo durant tout ce temps. Vous pouvez effacer des articles de cette liste en utilisant le menu accessible en cliquant avec le bouton d'action. Quand vous êtes satisfait, vous pouvez charger (c'est-à-dire faire un *file-in*) ce qui a été laissé et ainsi incorporer les modifications dans un nouveau *change set*.

Une chose utile à faire dans le *Recent Changes Browser* est d'effacer les évaluations *do it via remove doIts*. Habituellement vous ne voudriez pas charger (c'est-à-dire re-exécuter) ses expressions. Cependant, il existe une exception. Créer une classe apparaît comme un *dolt*. Avant de charger les méthodes d'une classe, la classe doit exister. Donc, si vous avez créer des nouvelles classes, chargez en premier lieu les *doIts* créateur de classes, ensuite utilisez *remove doIts* (pour ne pas charger les expressions d'un espace de travail) et enfin charger les méthodes.

Quand j'en ai fini avec le recouvrement (en anglais, *recover*), j'aime exporter (par *file-out*) mon nouveau *change set*, quitter Pharo sans sauvegarder l'image, redémarrer et m'assurer que mon nouveau fichier se charge parfaitement.

6.11 Résumé du chapitre

Pour développer efficacement avec Pharo, il est important d'investir quelques efforts dans l'apprentissage des outils disponibles dans l'environnement.

- Le navigateur de classes standard ou *Browser* est votre principale in-

terface pour naviguer dans les catégories, les classes, les protocoles et les méthodes existants et pour en définir de nouveaux. Ce navigateur offre plusieurs **boutons** pour accéder directement aux *senders* (*c-à-d.* les méthodes émettrices) ou aux *implementors* (*c-à-d.* les méthodes contenantes) d'un message, aux versions d'une méthode, etc.

- Plusieurs navigateurs différents existent (tel que OmniBrowser et le **Refactoring Browser**) et plusieurs sont spécialisés (comme le Hierarchy Browser) pour fournir différentes vues sur les classes et les méthodes.
- Depuis n'importe quel outil, vous pouvez sélectionner en surlignant le nom d'une classe ou celui d'une méthode pour obtenir immédiatement un navigateur en utilisant le raccourci-clavier CMD–b.
- Vous pouvez aussi naviguer dans le système Smalltalk de manière programmatique en envoyant des messages à `SystemNavigation default`.
- *Monticello* est un outil d'import-export, de versionnage (organisation et maintien de versions, en anglais, *versioning*) et de partage de paquetages de classes et de méthodes nommés aussi *packages*. Un paquetage Monticello comprend une catégorie, des sous-catégories et des protocoles de méthodes associés dans d'autres catégories.
- L'*Inspector* et l'*Explorer* sont deux outils utiles pour explorer et interagir avec les objets vivants dans votre image. Vous pouvez même inspecter des outils en cliquant avec le bouton d'action pour afficher leur *halo* et en sélectionnant l'icône *debug* .
- Le *Debugger* ou débogueur est un outil qui non seulement vous permet d'inspecter la pile d'exécution (*runtime stack*) de votre programme lorsqu'une erreur est signalée, mais aussi, vous assure une interaction avec tous les objets de votre application, incluant le code source. Souvent, vous pouvez modifier votre code source depuis le Debugger et continuer l'exécution. Ce débogueur est particulièrement efficace comme outil pour le développement orienté test (ou, en anglais, *test-first development*) en tandem avec SUnit (le chapitre 7).
- Le *Process Browser* ou navigateur de processus vous permet de piloter (monitoring), chercher (querying) et interagir avec les processus courants lancés dans votre image.
- Le *Method Finder* et le *Message Names Browser* sont deux outils destinés à la localisation de méthodes. Le premier excelle lorsque vous n'êtes pas sûr du nom mais que vous connaissez le comportement. Le second dispose d'une interface de navigation plus avancée pour le cas où vous connaissez au moins une partie du nom.
- Les *change sets* sont des journaux de bord (ou log) automatiquement générés pour tous les changements du code source dans l'image. Bien que rendus obsolètes par la présence de Monticello comme moyen de stockage et d'échange des versions de votre code source, ils sont toujours utiles, en particulier pour réparer des erreurs catastrophiques aussi rares soient-elles.
- Le *File List Browser* est un programme pour parcourir le système de fi-

chiers. Il vous permet aussi d'insérer du code source depuis le système de fichiers via `fileIn`.

- Dans le cas où votre image plante¹⁵ avant que vous l'ayez sauvegardée ou que vous ayez enregistré le code source avec Monticello, vous pouvez toujours retrouver vos modifications les plus récentes en utilisant un *Change List Browser*. Vous pouvez alors sélectionner les changements ou *changes* (en anglais) que vous voulez reprendre et les charger dans la copie la plus récente de votre image.

15. Nous parlons de *crash*, en anglais.

Chapitre 7

SUnit

7.1 Introduction

SUnit est un *framework*. À la fois simple et puissant, il est destiné à la création et au déploiement de tests. Comme son nom l'indique, SUnit est conçu plus particulièrement pour les *tests unitaires*, mais en fait, il peut être aussi utilisé pour des tests d'intégration ou des tests fonctionnels. SUnit a été développé par Ken Beck et ensuite grandement étendu par d'autres développeurs, dont notamment Joseph Pelrine, avec la prise en compte de la notion de ressource décrite dans la section 7.6. L'intérêt pour le test et le développement dirigé par les tests ne se limite pas à Pharo ou Smalltalk. L'automatisation des tests est devenue une pratique fondamentale des méthodes de développement agiles et tout développeur concerné par l'amélioration de la qualité du logiciel ferait bien de l'adopter. En effet, de nombreux développeurs apprécient la puissance du test unitaire et des versions de *xUnit* sont maintenant disponibles pour de nombreux langages dont Java, Python, Perl, .Net et Oracle.

Ce chapitre décrit SUnit 3.3 (la version courante lors de l'écriture de ce document) ; le site officiel de SUnit est sunit.sourceforge.net, dans lequel les mises à jour sont disponibles.

Le test et la construction de lignes de tests ne sont pas des pratiques nouvelles : il est largement reconnu que les tests sont utiles pour débusquer les erreurs. En considérant le test comme une pratique fondamentale et en promouvant les tests *automatisés*, l'eXtreme Programming a contribué à rendre le test productif et excitant plutôt qu'une corvée routinière dédaignée des développeurs. La communauté liée à Smalltalk bénéficie d'une longue tradition du test grâce au style de programmation incrémental supporté par l'environnement de développement. Traditionnellement, un programmeur Smalltalk écrirait des tests dans un *Workspace* dès qu'une méthode est ache-

vée. Quelquefois, un test serait intégré comme commentaire en tête de méthode en cours de mise au point, ou bien les tests plus élaborés seraient inclus dans la classe sous la forme de méthodes exemples. L'inconvénient de ces pratiques est que les tests édités dans un Workspace ne sont pas disponibles pour les autres développeurs qui modifient le code ; les commentaires et les méthodes exemples sont de ce point de vue préférables mais ne permettent toujours pas ni leur suivi ni leur automatisation. Les tests qui ne sont pas exécutés ne vous aident pas à trouver les bugs ! De plus, une méthode exemple ne donne au lecteur aucune information concernant le résultat attendu : vous pouvez exécuter l'exemple et voir le — peut-être surprenant — résultat, mais vous ne saurez pas si le comportement observé est correct.

SUnit est productif car il nous permet d'écrire des tests capables de s'auto-vérifier : le test définit lui-même quel est le résultat attendu. SUnit nous aide aussi à organiser les tests en groupes, à décrire le contexte dans lequel les tests doivent être exécutés, et à exécuter automatiquement un groupe de tests. En utilisant SUnit, vous pouvez écrire des tests en moins de deux minutes ; alors, au lieu d'écrire des portions de code dans un Workspace, nous vous encourageons à utiliser SUnit et à bénéficier de tous les avantages de tests sauvegardés et exécutables automatiquement.

Dans ce chapitre, nous commencerons par discuter de la raison des tests et de ce qu'est un bon test. Nous présenterons alors une série de petits exemples montrant comment utiliser SUnit. Finalement, nous étudierons l'implémentation de SUnit, de façon à ce que vous compreniez comment Smalltalk utilise la puissance de la réflexivité pour la mise en œuvre de ses outils.

7.2 Pourquoi tester est important

Malheureusement, beaucoup de développeurs croient perdre leur temps avec les tests. Après tout, *ils* n'écrivent pas de bug — seulement les *autres* programmeurs le font. La plupart d'entre nous avons dit, à un moment ou à un autre : “j'écrirais des tests si j'avais plus de temps”. Si vous n'écrivez jamais de bugs, et si votre code n'est pas destiné à être modifié dans le futur, alors, en effet, les tests sont une perte de temps. Pourtant, cela signifie très probablement que votre application est triviale, ou qu'elle n'est pas utilisée, ni par vous, ni par quelqu'un d'autre. Pensez aux tests comme à un investissement sur le futur : disposer d'une suite de tests est dès à présent tout à fait utile, mais sera extrêmement utile dans le futur, lorsque votre application ou votre environnement dans lequel elle s'exécute évoluera.

Les tests jouent plusieurs rôles. Premièrement, ils fournissent une documentation pour la fonctionnalité qu'ils couvrent. De plus, la documentation est active : l'observation des passes de tests vous indique que votre documen-

tation est à jour. Deuxièmement, les tests aident les développeurs à garantir que certaines modifications qu'ils viennent juste d'apporter à un package n'ont rien cassé dans le système — et à trouver quelles parties sont cassées si leur confiance s'avère contredite. Finalement, écrire des tests en même temps que — ou même avant de — programmer vous force à penser à la fonctionnalité que vous désirez concevoir et à *comment elle devrait apparaître au client*, plutôt qu'à comment la mettre en œuvre. En écrivant les tests en premier — avant le code — vous êtes contraint d'établir le contexte dans lequel votre fonctionnalité s'exécutera, la façon dont elle interagira avec le code client et les résultats attendus. Votre code s'améliorera : essayez donc !

Nous ne pouvons pas tester tous les aspects d'une application réaliste. Couvrir une application complète est tout simplement impossible et ne devrait pas être l'objectif du test. Même avec une bonne suite de tests, certains bugs seront quand même présents dans votre application, sommeillant en attendant l'occasion d'endommager votre système. Si vous constatez que c'est arrivé, tirez-en parti ! Dès que vous découvrez le bug, écrivez un test qui le met en évidence, exécutez le test et observez qu'il échoue. Alors vous pourrez commencer à corriger le bug : le test vous indiquera quand vous en aurez fini.

7.3 De quoi est fait un bon test ?

Écrire de bons tests constitue un savoir-faire qui peut s'apprendre facilement par la pratique. Regardons comment concevoir les tests de façon à en tirer le maximum de bénéfices.

1. Les tests doivent pouvoir être réitérés. Vous devez pouvoir exécuter un test aussi souvent que vous le voulez et vous devez toujours obtenir la même réponse.
2. Les tests doivent pouvoir s'exécuter sans intervention humaine. Vous devez même être capable de les exécuter pendant la nuit.
3. Les tests doivent vous raconter une histoire. Chaque test doit couvrir un aspect d'une partie de code. Un test doit agir comme un scénario que vous ou quelqu'un d'autre peut lire de façon à comprendre une partie de fonctionnalité.
4. Les tests doivent changer moins fréquemment que la fonctionnalité qu'ils couvrent : vous ne voulez pas changer tous vos tests à chaque fois que vous modifiez votre application. Une façon d'y parvenir est d'écrire des tests basés sur l'interface publique de la classe que vous êtes en train de tester. Il est possible d'écrire un test pour une méthode utilitaire privée si vous sentez que la méthode est suffisamment compliquée pour nécessiter le test, mais vous devez être conscient qu'un tel

test est susceptible d'être modifié ou intégralement supprimé quand vous pensez à une meilleure mise en œuvre.

Une conséquence du point (3) est que le nombre de tests doit être proportionnel au nombre de fonctions à tester : changer un aspect du système ne doit pas altérer tous les tests mais seulement un nombre limité. C'est important car avoir 100 échecs de tests doit constituer un signal beaucoup plus fort que d'en avoir 10. Cependant, cet idéal n'est pas toujours possible à atteindre : en particulier, si une modification casse l'initialisation d'un objet ou la mise en place du test, une conséquence probable peut être l'échec de tous les tests.

L'eXtreme Programming recommande d'écrire des tests avant de coder. Cela semble contredire nos instincts profonds de développeur. Tout ce que nous pouvons dire est : allez de l'avant et essayez donc ! Nous trouvons qu'écrire les tests avant le code nous aide à déterminer ce que nous voulons coder, nous aide à savoir quand nous avons terminé et nous aide à conceptualiser la fonctionnalité d'une classe et à concevoir son interface. De plus, le développement «*tester d'abord*» (test-first) nous donne le courage d'avancer rapidement parce que nous n'avons pas peur d'oublier quelque chose d'important.

7.4 SUnit par l'exemple

Avant de considérer SUnit en détails, nous allons montrer un exemple, étape par étape. Nous utilisons un exemple qui teste la classe Set. Essayez de saisir le code au fur et à mesure que nous avançons.

Étape 1 : créer la classe de test

 Créez tout d'abord une nouvelle sous-classe de TestCase nommée ExampleSetTest. Ajoutez-lui deux variables d'instance de façon à ce que votre classe ressemble à ceci :

Classe 7.1 – Un exemple de classe de test pour Set

```
TestCase subclass: #ExampleSetTest
instanceVariableNames: 'full empty'
classVariableNames: ''
poolDictionaries: ''
category: 'MySetTest'
```

Nous utiliserons la classe ExampleSetTest pour regrouper tous les tests relatifs à la classe Set. Elle définit le contexte dans lequel les tests s'exécuteront. Ici, le contexte est décrit par les deux variables d'instance full et empty qui seront utilisées pour représenter respectivement un Set plein et un Set vide.

Le nom de la classe n'est pas fondamental, mais par convention il devrait se terminer par Test. Si vous définissez une classe nommée Pattern et que vous nommez la classe de test correspondante PatternTest, les deux classes seront présentées ensemble, par ordre alphabétique, dans le System Browser (en considérant qu'elles sont dans la même catégorie). Il est indispensable que votre classe soit une sous-classe de TestCase.

Étape 2 : initialiser le contexte du test

La méthode setUp (en anglais, *configurer*) définit le contexte dans lequel les tests vont s'exécuter, un peu comme la méthode initialize. setUp est invoquée avant l'exécution de chaque méthode de test définie dans la classe de test.

 Définissez la méthode setUp de la façon suivante pour initialiser la variable empty, de sorte qu'elle référence un Set vide ; et la variable full, de sorte qu'elle référence un Set contenant deux éléments.

Méthode 7.2 – Mettre au point une installation

```
ExampleSetTest»setUp
empty := Set new.
full := Set with: 5 with: 6
```

Dans le jargon du test, le contexte est appelé *l'installation* du test (en anglais, *fixture*).

Étape 3 : écrire quelques méthodes de test

Créons quelques tests en définissant quelques méthodes dans la classe ExampleSetTest. Chaque méthode représente un test ; le nom de la méthode devrait commencer par la chaîne "test" pour que SUnit les regroupe en suites de tests. Les méthodes de test ne prennent pas d'arguments.

Définissez les méthodes de test suivantes.

Le premier test, nommé `testIncludes`, teste la méthode `includes`: de Set. Le test dit que, envoyer le message `includes: 5` à un Set contenant 5 devrait retourner true. Clairement, ce test repose sur le fait que la méthode `setUp` s'est déjà exécutée.

Méthode 7.3 – Tester l'appartenance à un Set

```
ExampleSetTest»testIncludes
```

```
self assert: (full includes: 5).
self assert: (full includes: 6)
```

Le second test nommé `testOccurrences` vérifie que le nombre d'occurrences de 5 dans le Set `full` est égal à un, même si nous ajoutons un autre élément 5 au Set.

Méthode 7.4 – Tester des occurrences

```
ExampleSetTest»testOccurrences
```

```
self assert: (empty occurrencesOf: 0) = 0.
self assert: (full occurrencesOf: 5) = 1.
full add: 5.
self assert: (full occurrencesOf: 5) = 1
```

Finalement, nous testons que le Set n'a plus d'élément 5 après que nous l'ayons supprimé.

Méthode 7.5 – Tester la suppression

```
ExampleSetTest»testRemove
```

```
full remove: 5.
self assert: (full includes: 6).
self deny: (full includes: 5)
```

Notez l'utilisation de la méthode `deny:` pour garantir que quelque chose ne doit pas être vrai. `aTest deny: anExpression` est équivalent à `aTest assert: anExpression not`, mais en beaucoup plus lisible.

Étape 4 : exécuter les tests

L'exécution la plus simple des tests s'effectue en utilisant directement le Browser. Cliquez avec le bouton d'action sur le paquetage, le nom de la classe ou une méthode de tests et, de là, sélectionnez `run the tests (t)`. Les méthodes de tests seront alors, entièrement ou partiellement, signalées par une puce rouge ou verte selon le succès complet, partiel ou bien l'échec des tests.

Vous pouvez aussi sélectionner des séries de tests à lancer et obtenir un journal (en anglais, un *log*) plus détaillé des résultats en lançant l'exécuteur

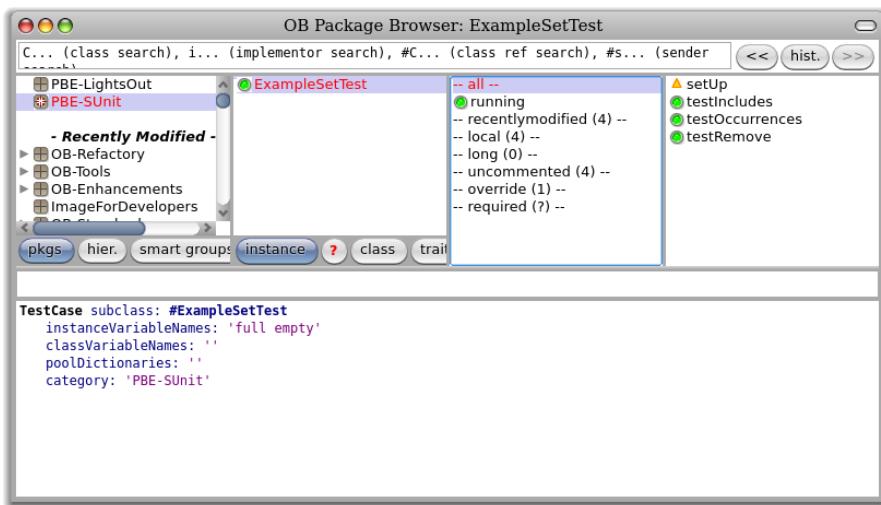


FIGURE 7.1 – Lancer les tests SUnit depuis le Browser.

de tests *Test Runner* de SUnit depuis le menu `World > Test Runner`. L'exécuteur de tests, montré dans la figure 7.2, est conçu pour faciliter l'exécution de groupes de tests. Le panneau le plus à gauche présente toutes les catégories qui contiennent des classes de test (*c-à-d.* sous-classes de `TestCase`). Lorsque certaines de ces catégories sont sélectionnées, les classes de test qu'elles contiennent apparaissent dans le panneau de droite. Les classes abstraites sont en italique et la hiérarchie des classes de test est visible par l'indentation, ainsi les sous-classes de `ClassTestCase` sont plus indentées que les sous-classes de `TestCase`.

Ouvrez le *Test Runner*, sélectionnez la catégorie `MySetTest` et cliquez le bouton `Run Selected`.

Introduisez un bug dans `ExampleSetTest>>testRemove` et évaluez le test à nouveau. Par exemple, remplacez 5 par 4.

Les tests qui ne sont pas passés (s'il y en a) sont listés dans les panneaux de droite du *Test Runner*. Si vous voulez en déboguer un et voir pourquoi il échoue, il suffit juste de cliquer sur le nom.

Étape 5 : interpréter les résultats

La méthode `assert:`, définie dans la classe `TestCase`, prend un booléen en argument ; habituellement la valeur d'une expression testée. Quand cet

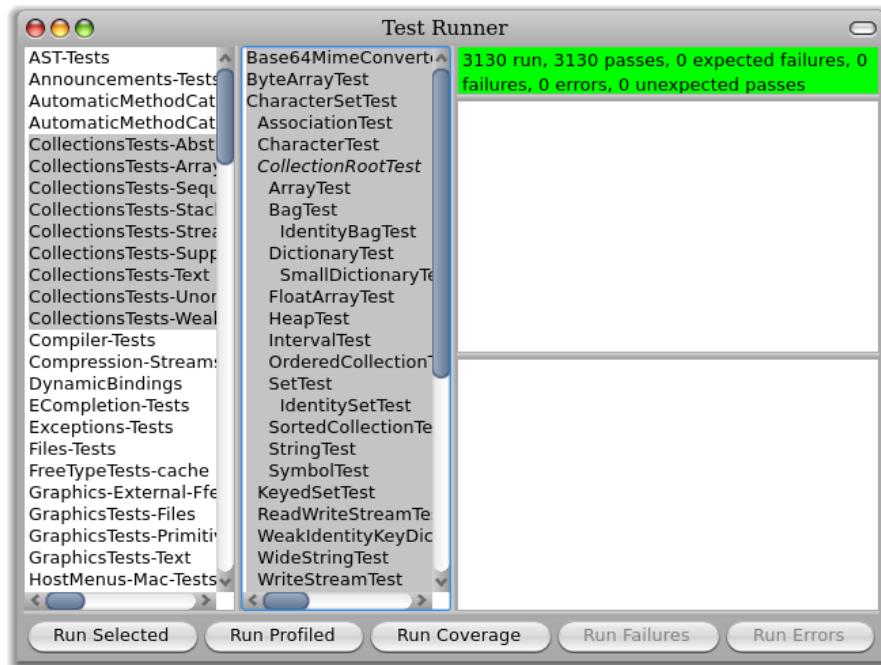


FIGURE 7.2 – SUnit, l'exécuteur de test de Pharo.

argument est à vrai (`true`), le test est réussi ; quand cet argument est à faux (`false`), le test échoue.

Il y a actuellement trois résultats possibles pour un test. Le résultat espéré est que toutes les assertions du test soient vraies, dans ce cas le test réussit. Dans l'exécuteur de tests (`TestRunner`), quand tous les tests réussissent, la barre du haut devient verte. Toutefois, il reste deux possibilités pour que quelque chose se passe mal quand vous évaluez le test. Le plus évident est qu'une des assertions peut être fausse, entraînant l'échec du test. Toutefois, il est aussi possible qu'une erreur intervienne pendant l'exécution du test, telle qu'une erreur *message non compris* ou une erreur d'*indice hors limites*.

Si une erreur survient, les assertions de la méthode de test peuvent ne pas avoir été exécutées du tout, ainsi nous ne pouvons pas dire que le test a échoué. Toutefois, quelque chose est clairement faux ! Dans l'exécuteur de tests (`TestRunner`), la barre du haut devient jaune pour les tests en échec et ces tests sont listés dans le panneau du milieu à droite, alors que pour les tests erronés, la barre devient rouge et ces tests sont listés dans le panneau en bas à droite.

Modifiez vos tests de façon à provoquer des erreurs et des échecs.

7.5 Les recettes pour SUnit

Cette section vous donne plus d'informations sur la façon d'utiliser SUnit. Si vous avez utilisé un autre *framework* de tests comme JUnit¹, ceci vous sera familier puisque tous ces *frameworks* sont issus de SUnit. Normalement, vous utiliserez l'IHM² de SUnit pour exécuter les tests à l'exception de certains cas.

Autres assertions

En supplément de `assert:` et `deny:`, il y a plusieurs autres méthodes pouvant être utilisées pour spécifier des assertions.

Premièrement, `assert:description:` et `deny:description:` prennent un second argument qui est un message sous la forme d'une chaîne de caractères pouvant être utilisé pour décrire la raison de l'échec au cas où elle n'apparaît pas évidente à la lecture du test lui-même. Ces méthodes sont décrites dans la section 7.7.

Ensuite, SUnit dispose de deux méthodes supplémentaires, `should:raise:` et `shouldnt:raise:` pour la propagation des exceptions de test. Par exemple, `self should: aBlock raise: anException` vous permet de tester si une exception particulière est levée pendant l'exécution de `aBlock`. La méthode 7.6 illustre l'utilisation de `should:raise:`.

 *Essayez d'évaluer ce test.*

Notez que le premier argument des méthodes `should:` et `shouldnt:` est un bloc qui contient l'expression à évaluer.

Méthode 7.6 – Tester la levée d'une erreur

```
ExampleSetTest»testIllegal
self should: [empty at: 5] raise: Error.
self should: [empty at: 5 put: #zork] raise: Error
```

SUnit est portable : il peut être utilisé avec tous les dialectes de Smalltalk. Afin de rendre SUnit portable, ses développeurs ont retiré les parties dépendantes des dialectes. La méthode de classe `TestResult class»error` retourne la classe erreur du système de façon indépendante du dialecte. Vous pouvez en profiter aussi : si vous voulez écrire des tests qui fonctionnent quel que soit le dialecte de Smalltalk, vous pouvez écrire la méthode 7.6 ainsi :

1. <http://junit.org>

2. Interface Homme Machine.

Méthode 7.7 – Gestion portable des erreurs

ExampleSetTest»testIllegal

```
self should: [empty at: 5] raise: TestResult error.  
self should: [empty at: 5 put: #zork] raise: TestResult error
```

 *Essayez-le !*

Exécuter un test simple

Normalement, vous exécuterez vos tests avec l'exécuteur de tests (TestRunner). Vous pouvez aussi le lancer en faisant un `print it` du code TestRunner open. Vous pouvez exécuter un simple test de la façon suivante :

```
ExampleSetTest run: #testRemove → 1 run, 1 passed, 0 failed, 0 errors
```

Exécuter tous les tests d'une classe de test

Toute sous-classe de TestCase répond au message `suite` qui construira une suite de tests contenant toutes les méthodes de la classe dont le nom commence par la chaîne "test". Pour exécuter les tests de la suite, envoyez-lui le message `run`. Par exemple :

```
ExampleSetTest suite run → 5 run, 5 passed, 0 failed, 0 errors
```

Dois-je sous-classer TestCase ?

Avec JUnit, vous pouvez construire un TestSuite dans n'importe quelle classe contenant des méthodes `test*`. En Smalltalk, vous pouvez faire la même chose mais vous aurez à créer une suite manuellement et votre classe devra mettre en œuvre toutes les méthodes essentielles de TestCase comme `assert:`. Nous ne vous le recommandons pas. Le *framework* est déjà là : utilisez-le.

7.6 Le *framework* SUnit

Comme l'illustre la figure 7.3, SUnit consiste en quatre classes principales : TestCase, TestSuite, TestResult et TestResource. La notion de *ressource de test* a été introduite dans SUnit 3.1 pour représenter une ressource coûteuse à installer mais qui peut être utilisée par toute une série de tests. Un TestResource spécifie une méthode `setUp` qui est exécutée une seule fois avant la suite de tests ; à la différence de la méthode TestCase»`setUp` qui est exécutée avant chaque test.

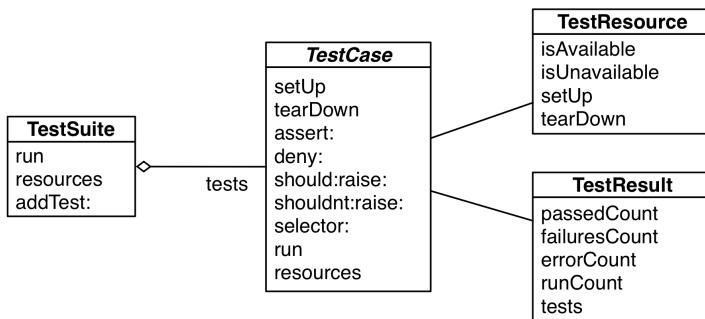


FIGURE 7.3 – Les quatres classes constituant le noyau de SUnit.

TestCase

TestCase est une classe abstraite conçue pour avoir des sous-classes ; chacune de ses sous-classes représente un groupe de tests qui partagent un contexte commun (ce qui constitue une suite de tests). Chaque test est évalué par la création d'une nouvelle instance d'une sous-classe de **TestCase** par l'exécution de `setUp`, par l'exécution de la méthode de test elle-même puis par l'exécution de `tearDown`³.

Le contexte est porté par des variables d'instance de la sous-classe et par la spécialisation de la méthode `setUp` qui initialise ces variables d'instance. Les sous-classes de **TestCase** peuvent aussi surcharger la méthode `tearDown` qui est invoquée après l'exécution de chaque test et qui peut être utilisée pour libérer tous les objets alloués pendant `setUp`.

TestSuite

Les instances de la classe **TestSuite** contiennent une collection de cas de tests. Une instance de **TestSuite** contient des tests et d'autres suites de tests. En fait, une suite de tests contient des instances de sous-classes de **TestCase** et de **TestSuite**. Individuellement, les **TestCases** et les **TestSuites** comprennent le même protocole, ainsi elles peuvent être traitées de la même façon ; par exemple, elles comprennent toutes `run`. Il s'agit en fait de l'application du patron de conception *Composite* pour lequel **TestSuite** est le composite et les **TestCases** sont les feuilles — voir les *Design Patterns* pour plus d'informations sur ce patron⁴.

3. En français, démolir.

4. Erich Gamma *et al.*, *Design Patterns : Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison Wesley, 1995, ISBN 0-201-63361-2-(3).

TestResult

La classe `TestResult` représente les résultats de l'exécution d'un `TestSuite`. Elle mémorise le nombre de tests passés, le nombre de tests en échec et le nombre d'erreurs signalées.

TestResource

Une des caractéristiques importantes d'une suite de tests est que les tests doivent être indépendants les uns des autres : l'échec d'un test ne doit pas entraîner l'échec des autres tests qui en dépendent ; l'ordre dans lequel les tests sont exécutés ne doit pas non plus importer. Évaluer `setUp` avant chaque test et `tearDown` après permet de renforcer cette indépendance.

Malgré tout, il y a certains cas pour lesquels la préparation du contexte nécessaire est simplement trop lent pour qu'il soit réalisable de le faire avant l'exécution de chaque test. De plus, si nous savons que les tests n'altèrent pas les ressources qu'ils utilisent, alors il est prohibitif de les initialiser pour chaque test ; il est suffisant de les initialiser une seule fois pour chaque suite de tests. Supposez, par exemple, qu'une suite de tests ait besoin d'interroger une base de données ou d'effectuer certaines analyses sur du code compilé. Pour ces situations, elle est censée initialiser et ouvrir une connexion vers la base de données ou compiler du code source avant l'exécution des tests.

Où pourrions nous conserver ces ressources de façon à ce qu'elles puissent être partagées par les tests d'une suite ? Les variables d'instance d'une sous-classe de `TestCase` particulière ne le pourraient pas parce que ses instances ne subsistent que pendant la durée d'un seul test. Une variable globale ferait l'affaire, mais utiliser trop de variables globales pollue l'espace de nommage et la relation entre la variable globale et les tests qui en dépendent ne serait pas explicite. Une meilleure solution est de placer les ressources nécessaires dans l'objet singleton d'une certaine classe. La classe `TestResource` est définie pour avoir des sous-classes utilisées comme classes de ressource. Chaque sous-classe de `TestResource` comprend le message `current` qui retournera son instance singleton. Les méthodes `setUp` et `tearDown` doivent être surchargées dans la sous-classe pour permettre à la ressource d'être initialisée et libérée.

Une chose demeure : d'une certaine façon, SUnit doit être informé de quelles ressources sont associées avec quelle suite de tests. Une ressource est associée à une sous-classe particulière de `TestCase` par la surcharge de la méthode de classe `resources`. Par défaut, les ressources d'un `TestSuite` sont constituées par l'union des ressources des `TestCases` qu'il contient.

Voici un exemple. Nous définissons une sous-classe de `TestResource` nommée `MyTestResource` et nous l'associons à `MyTestCase` en spécialisant la méthode de classe `resources` de sorte qu'elle retourne un tableau contenant les

classes de test qu'il utilisera.

Classe 7.8 – Un exemple de sous-classe de TestResource

```
TestResource subclass: #MyTestResource
  instanceVariableNames: ""

MyTestCase class»resources
  "associe la ressource avec cette classe de test"
  ↑{ MyTestResource }
```

7.7 Caractéristiques avancées de SUnit

En plus de TestResource, la version courante de SUnit dispose de la description des assertions avec des chaînes, d'une gestion des traces et de la reprise sur un test en échec (cette dernière faisant appel aux méthodes avec terme anglophone *resumable*).

Description des assertions avec des chaînes de caractères

Le protocole des assertions de TestCase comprend un certain nombre de méthodes permettant au programmeur de fournir une description de l'assertion. La description est une chaîne de caractères ; si le test échoue, cette chaîne est affichée par l'exécuteur de tests. Bien sûr, cette chaîne peut être construite dynamiquement.

```
| e |
e := 42.
self assert: e = 23
  description: 'attendu 23, obtenu ', e printString
```

Les méthodes correspondantes de TestCase sont :

```
#assert:description:
#deny:description:
#should:description:
#shouldnt:description:
```

Gestion des traces

Les chaînes descriptives présentées précédemment peuvent aussi être tracées dans un flux de données Stream tel que le Transcript ou un flux associé à un fichier. Vous pouvez choisir de tracer ou non en surchargeant TestCase»isLogging dans votre classe de test ; vous devez aussi choisir dans quoi tracer en surchargeant TestCase»failureLog de façon à fournir un stream approprié.

Continuer après un échec

SUnit nous permet aussi d'indiquer si un test doit ou non continuer après un échec. Il s'agit d'une possibilité vraiment puissante qui utilise les mécanismes d'exception offerts par Smalltalk. Pour comprendre dans quel cas l'utiliser, voyons un exemple. Observez l'expression de test suivante :

```
aCollection do: [ :each | self assert: each even]
```

Dans ce cas, dès que le test trouve le premier élément de la collection qui n'est pas pair (en anglais, even), le test s'arrête. Pourtant, habituellement, nous voudrions bien continuer et voir aussi quels éléments (et donc combien) ne sont pas pairs (*c-à-d.* ne répondent pas à even) et peut-être aussi tracer cette information. Vous pouvez le faire de la façon suivante :

```
aCollection do:
  [:each |
  self
    assert: each even
    description: each printString , ' n"est pas pair'
    resumable: true]
```

Pour chaque élément en échec, un message sera affiché dans le flux des traces. Les échecs ne sont pas cumulés, *c-à-d.* si l'assertion échoue 10 fois dans la méthode de test, vous ne verrez qu'un seul échec. Toutes les autres méthodes d'assertion que nous avons vues ne permettent pas la reprise ; assert: p description: s est équivalente à assert: p description: s resumable: false.

7.8 La mise en œuvre de SUnit

La mise en œuvre de SUnit constitue un cas d'étude intéressant d'un *framework* Smalltalk. Étudions quelques aspects clés de la mise en œuvre en suivant l'exécution d'un test.

Exécuter un test

Pour exécuter un test, nous évaluons l'expression (aTestClass selector: aSymbol) run.

La méthode TestCase»run crée une instance de TestResult qui collectera les résultats des tests ; ensuite, elle s'envoie le message run: (voir la figure 7.4).

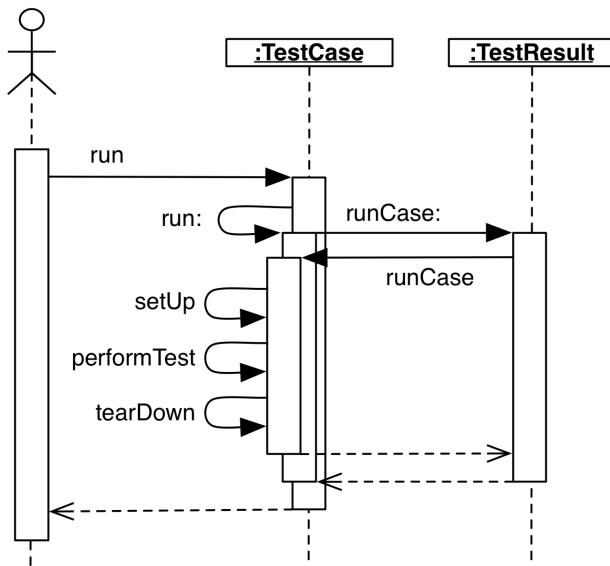


FIGURE 7.4 – Exécuter un test.

Méthode 7.9 – Exécuter un cas de test

```

TestCase»run
| result |
result := TestResult new.
self run: result.
^result
  
```

La méthode TestCase»run: envoie le message runCase: au résultat de test de classe TestResult :

Méthode 7.10 – Passage du cas de test au TestResult

```

TestCase»run: aResult
aResult runCase: self
  
```

La méthode TestResult»runCase: envoie le message runCase à un seul test pour l'exécuter. TestResult»runCase s'arrange avec toute exception qui pourrait être levée pendant l'exécution d'un test, évalue un TestCase en lui envoyant le message runCase et compte les erreurs, les échecs et les passes.

Méthode 7.11 – Capture des erreurs et des échecs de test

```

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]
```

La méthode `TestCase»runCase` envoie les messages `setUp` et `tearDown` comme indiqué ci-dessous.

Méthode 7.12 – Modèle de méthode de test

```
TestCase»runCase
  [self setUp.
  self performTest] ensure: [self tearDown]
```

Exécuter un TestSuite

Pour exécuter plus d'un test, nous envoyons le message `run` à un `TestSuite` qui contient les tests adéquats. `TestCase` class procure des fonctionnalités lui permettant de construire une suite de tests. L'expression `MyTestCase buildSuiteFromSelectors` retourne une suite contenant tous les tests définis dans la classe `MyTestCase`. Le cœur de ce processus est :

Méthode 7.13 – Auto-construction de la suite de test

```
TestCase»testSelectors
  ↑self selectors asSortedCollection asOrderedCollection select: [:each |
  ('test*' match: each) and: [each numArgs isZero]]
```

La méthode `TestSuite»run` crée une instance de `TestResult`, vérifie que toutes les ressources sont disponibles avec `areAllResourcesAvailable` puis envoie elle-même le message `run:` qui exécute tous les tests de la suite. Toutes les ressources sont alors libérées.

Méthode 7.14 – Exécuter une suite de tests

```
TestSuite»run
  | result |
  self resources do: [ :res |
```

```

res isAvailable ifFalse: [↑res signalInitializationError].
[self run: result] ensure: [self resources do: [:each | each reset]].
↑result

```

Méthode 7.15 – Passage de la suite de tests au résultat de test

```

TestSuite»run: aResult
self tests do: [:each |
  self changed: each.
  each run: aResult]

```

La classe `TestResource` et ses sous-classes conservent la trace de leurs instances en cours (une par classe) pouvant être accédées et créées en utilisant la méthode de classe `current`. Cette instance est nettoyée quand les tests ont fini de s'exécuter et que les ressources sont libérées.

Comme le montre la méthode de classe `TestResource class»isAvailable` (en anglais, *est-disponible*), le contrôle de la disponibilité de la ressource permet de la recréer en cas de besoin. Pendant sa création, l'instance de `TestResource` est initialisée et la méthode `setUp` est invoquée.

Méthode 7.16 – Disponibilité de la ressource de test

```

TestResource class»isAvailable
↑self current notNil and: [self current isAvailable]

```

Méthode 7.17 – Création de la ressource de test

```

TestResource class»current
current isNil ifTrue: [current := self new].
↑current

```

Méthode 7.18 – Initialisation de la ressource de test

```

TestResource»initialize
super initialize.
self setUp

```

7.9 Quelques conseils sur les tests

Bien que les mécanismes de tests soient simples, il n'est pas toujours facile d'en écrire de bons. Voici quelques conseils pour leur conception.

Les règles de Feathers. Michael Feathers, un auteur et consultant en processus agile écrit⁵ :

5. Voir <http://www.artima.com/weblogs/viewpost.jsp?thread=126923> – 9 Septembre 2005.

Un test n'est pas un test unitaire si :

- *il communique avec une base de données,*
- *il communique au travers du réseau,*
- *il modifie le système de fichiers,*
- *il ne peut pas s'exécuter en même temps qu'un autre de vos tests unitaires ou*
- *vous devez préparer votre environnement de façon particulière pour l'exécuter (comme éditer un fichier de configuration).*

Des tests qui s'exécutent ainsi ne sont pas mauvais. Souvent ils valent la peine d'être écrits et ils peuvent être développés au sein d'un environnement de tests. Cependant, il est important de pouvoir les séparer des vrais tests unitaires de façon à ce qu'il soit possible de maintenir un ensemble de tests que nous pouvons exécuter rapidement à chaque fois que nous apportons nos modifications.

Ne vous placez jamais dans une situation où vous ne voulez pas lancer votre suite de tests unitaires parce que cela prend trop de temps.

Tests unitaires contre tests d'acceptation. Des tests unitaires capturent une partie de la fonctionnalité et, comme tels, permettent de faciliter l'identification des bugs de cette fonctionnalité. Essayez d'avoir, autant que possible, des tests unitaires pour chaque méthode pouvant potentiellement poser problème et regroupez-les par classe. Cependant, pour des situations profondément récursives ou complexes à installer, il est plus facile d'écrire des tests qui représentent un scénario cohérent pour l'application visée ; ce sont des tests d'acceptation ou tests fonctionnels. Des tests qui violent les principes de Feathers peuvent faire de bons tests d'acceptation. Groupez les tests d'acceptation en cohérence avec la fonctionnalité qu'ils testent. Par exemple, si vous écrivez un compilateur, vous pourriez écrire des tests d'acceptation avec des assertions qui concernent le code généré pour chaque instruction utilisable du langage source. De tels tests pourraient concerner beaucoup de classes et pourraient prendre beaucoup de temps pour s'exécuter parce qu'ils modifient le système de fichiers. Vous pouvez les écrire avec SUnit, mais vous ne voudriez pas les exécuter à chaque modification mineure, ainsi ils doivent être séparés des vrais tests unitaires.

Les règles de Black. Pour tous les tests du système, vous devriez être en mesure d'identifier une propriété pour laquelle le test renforce votre confiance. Il est évident qu'il ne devrait pas y avoir de propriété importante que vous ne testez pas. Cette règle établit le fait moins évident qu'il ne devrait pas y avoir de tests sans valeur ajoutée de nature à accroître votre confiance envers une propriété utile. Par exemple, il n'est pas bon d'avoir plusieurs tests pour la même propriété. En fait, c'est néfaste pour deux raisons. D'abord, ils rendent le comportement de la classe plus difficile à déduire à la lecture des tests. Ensuite, un bug dans

le code est susceptible de casser beaucoup de tests et donc il est plus difficile de jauger du nombre de bogues restants dans le code. Ainsi, ne pensez qu'à une seule propriété quand vous écrivez un test.

7.10 Résumé du chapitre

Ce chapitre a expliqué en quoi les tests constituent un investissement important pour le futur de votre code. Nous avons expliqué, étape par étape, comment spécifier quelques tests pour la classe Set. Ensuite, nous avons décrit simplement le cœur du *framework* SUnit en présentant les classes TestCase, TestResult, TestSuite et TestResources. Finalement, nous avons détaillé SUnit en suivant l'exécution d'un test et d'une suite de tests.

- Pour maximiser leur potentiel, des tests unitaires devraient être rapides, réitérables, indépendants d'une intervention humaine et couvrir une seule partie de fonctionnalité.
- Les tests pour la classe nommée MyClass sont dans la classe nommée MyClassTest qui devrait être implantée comme une sous-classe de TestCase.
- Initialisez vos données de test dans une méthode setUp.
- Chaque méthode de test devrait commencer par le mot "test".
- Utilisez les méthodes de TestCase comme assert:, deny: et autres, pour établir vos assertions.
- Exécutez les tests en utilisant l'exécuteur de tests SUnit (dans l'onglet Tools).

Chapitre 8

Les classes de base

Une grande partie de la magie de Smalltalk ne réside pas dans son langage mais dans ses bibliothèques de classes. Pour programmer efficacement en Smalltalk, vous devez apprendre comment les bibliothèques de classes servent le langage et l'environnement. Les bibliothèques de classes sont entièrement écrites en Smalltalk et peuvent facilement être étendues, puisqu'un paquetage peut ajouter une nouvelle fonctionnalité à une classe même s'il ne définit pas cette classe.

Notre but ici n'est pas de présenter en détail l'intégralité des bibliothèques de classes de Pharo, mais plutôt d'indiquer quelles classes et méthodes clés vous devrez utiliser ou surcharger pour programmer efficacement. Ce chapitre couvre les classes de base qui vous seront utiles dans la plupart de vos applications : Object, Number et ses sous-classes, Character, String, Symbol et Boolean.

8.1 Object

Dans tous les cas, Object est la racine de la hiérarchie d'héritage. En réalité, dans Pharo, la vraie racine de la hiérarchie est ProtoObject qui est utilisée pour définir les entités minimales qui se font passer pour des objets, mais nous pouvons ignorer ce point pour l'instant.

La classe Object peut être trouvée dans la catégorie *Kernel-Objects*. Étonnamment, nous y trouvons plus de 400 méthodes (avec les extensions). En d'autres termes, toutes les classes que vous définirez seront automatiquement munies de ces 400 méthodes, que vous sachiez ou non ce qu'elles font. Notez que certaines de ces méthodes devraient être supprimées et que dans les nouvelles versions de Pharo certaines méthodes superflues pourraient l'être.

Le commentaire de la classe Object indique :

Object est la classe racine de la plupart des autres classes dans la hiérarchie des classes. Les exceptions sont ProtoObject (super-classe de Object) et ses sous-classes. La classe Object fournit le comportement par défaut, commun à tous les objets classiques, comme l'accès, la copie, la comparaison, le traitement des erreurs, l'envoi de messages et la réflexion. Les messages utiles auxquels tous les objets devraient répondre sont également définis ici. Object n'a pas de variable d'instance, aucune ne devrait être créée. Ceci est dû aux nombreuses classes d'objets qui héritent de Object et qui ont des implémentations particulières (SmallInteger et UndefinedObject par exemple) ou à certaines classes standards que la VM connaît et pour lesquelles leur structure et leur organisation sont importantes.

Si nous naviguons dans les catégories des méthodes d'instance de Object, nous commençons à voir quelques-uns des comportements-clé qu'elle offre.

Impression

Tout objet en Smalltalk peut renvoyer une forme imprimée de lui-même. Vous pouvez sélectionner n'importe quelle expression dans un Workspace et sélectionner le menu **print it** : ceci exécute l'expression et demande à l'objet renvoyé de s'imprimer. En réalité le message `printString` est envoyé à l'objet retourné. La méthode `printString`, qui est une méthode générique, envoie le message `printOn:` à son receveur. Le message `printOn:` est un point d'entrée qui peut être spécialisé.

`Object»printOn:` est une des méthodes que vous surchargerez le plus souvent. Cette méthode prend comme argument un flux (`Stream`) dans lequel une représentation en chaîne de caractères (`String`) de l'objet sera écrite. L'implémentation par défaut écrit simplement le nom de la classe précédée par "a" ou "an". `Object»printString` retourne la chaîne de caractères (`String`) qui est écrite.

Par exemple, la classe `Browser` ne redéfinit pas la méthode `printOn:` et, envoier le message `printString` à une de ces instances exécute les méthodes définies dans `Object`.

```
Browser new printString → 'a Browser'
```

La classe `Color` montre un exemple de spécialisation de `printOn:`. Elle imprime le nom de la classe suivi par le nom de la méthode de classe utilisée pour générer cette couleur, comme le montre le code ci-dessous qui imprime une instance de cette classe.

Méthode 8.1 – Redéfinir `printOn:`

```
Color»printOn: aStream
| name |
(name := self name) ifNotNil:
[ ↑ aStream
nextPutAll: 'Color ';
nextPutAll: name].
self storeOn: aStream
```

Color red printString → 'Color red'

Notez que le message `printOn:` n'est pas le même que `storeOn:`. Le message `storeOn:` ajoute au flux passé en argument une expression pouvant être utilisée pour recréer le receveur. Cette expression est évaluée quand le flux est lu avec le message `readFrom:.` `printOn:` retourne simplement une version textuelle du receveur. Bien sûr, il peut arriver que cette représentation textuelle puisse représenter le receveur sous la forme d'une expression auto-évaluée.

Un mot à propos de la représentation et de la représentation auto-évaluée. En programmation fonctionnelle, les expressions retournent des valeurs quand elles sont évaluées. En Smalltalk, les messages (expressions) retournent des objets (valeurs). Certains objets ont la propriété sympathique d'être eux-mêmes leur propre valeur. Par exemple, la valeur de l'objet `true` est lui-même, *c-à-d.* l'objet `true`. Nous appelons de tels objets des *objets auto-évalués*. Vous pouvez voir une version *imprimée* de la valeur d'un objet quand vous imprimez l'objet dans un Workspace. Voici quelques exemples de telles expressions auto-évaluées.

true	→	true
3@4	→	3@4
\$a	→	\$a
#(1 2 3)	→	#(1 2 3)
Color red	→	Color red

Notez que certains objets comme les tableaux sont auto-évalués ou non suivant les objets qu'ils contiennent. Par exemple, un tableau de booléens est auto-évalué alors qu'un tableau de personnes ne l'est pas. L'exemple suivant montre qu'un tableau dynamique est auto-évalué seulement si ses éléments le sont :

{10@10. 100@100}	→	{10@10. 100@100}
{Browser new . 100@100}	→	an Array(a Browser 100@100)

Rappelez-vous que les tableaux littéraux ne peuvent contenir que des littéraux. Ainsi le tableau suivant ne contient pas deux éléments mais six éléments littéraux.

#(10@10 100@100)	→	#(10 #@ 10 100 #@ 100)
------------------	---	------------------------

Beaucoup de spécialisations de la méthode printOn: implémentent le comportement d'auto-évaluation. Les implementations de Point>printOn: et Interval >printOn: sont auto-évaluées.

Méthode 8.2 – Auto-évaluation de Point

```
Point>printOn: aStream
```

"The receiver prints on aStream in terms of infix notation."

```
x printOn: aStream.  
aStream nextPut: $@.  
y printOn: aStream
```

Le commentaire de cette méthode dit que le receveur imprime sur le flux aStream avec une insertion dans la notation.

Méthode 8.3 – Auto-évaluation de Interval

```
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) "les intervalles sont auto-évalués"

Identité et égalité

En Smalltalk, le message = teste l'*égalité* d'objets (*c-à-d.* si deux objets représentent la même valeur) alors que == teste l'*identité* (*c-à-d.* si deux expressions représentent le même objet).

L'implémentation par défaut de l'égalité entre objets teste l'identité d'objets :

Méthode 8.4 – Égalité par défaut

```
Object>= anObject
```

"Answer whether the receiver and the argument represent the same object."

If = is redefined in any subclass, consider also redefining the message hash."

↑ self == anObject

C'est une méthode que vous voudrez souvent surcharger. Considérez le cas de la classe des nombres complexes Complex :

(1 + 2 i) = (1 + 2 i) → true "même valeur"
(1 + 2 i) == (1 + 2 i) → false "mais objets différents"

Ceci fonctionne parce que Complex surcharge = comme suit :

Méthode 8.5 – Égalité des nombres complexes

```
Complex»= anObject
anObject isComplex
ifTrue: [↑ (real = anObject real) & (imaginary = anObject imaginary)]
ifFalse: [↑ anObject adaptToComplex: self andSend: #=]
```

L'implémentation par défaut de Object»~= renvoie simplement l'inverse de Object»= et ne devrait normalement pas être modifiée.

```
(1 + 2 i) ~= (1 + 4 i) → true
```

Si vous surchargez =, vous devriez envisager de surcharger hash. Si des instances de votre classe sont utilisées comme clés dans un dictionnaire (Dictionary), vous devrez alors vous assurer que les instances qui sont considérées égales ont la même valeur de hachage (hash) :

Méthode 8.6 – hash doit être ré-implémentée pour les nombres complexes

```
Complex>hash
"Hash is reimplemented because = is implemented."
↑ real hash bitXor: imaginary hash.
```

Alors que vous devez surcharger à la fois = et hash, vous ne devriez jamais surcharger == puisque la sémantique de l'identité d'objets est la même pour toutes les classes. == est une méthode primitive de ProtoObject.

Notez que Pharo a certains comportements étranges comparé à d'autres Smalltalks : par exemple, un symbole et une chaîne de caractères peuvent être égaux si la chaîne de caractères associée au symbole est égale à la chaîne de caractères (nous considérons ce comportement comme un bug, pas comme une fonctionnalité).

```
#'lulu' = 'lulu' → true
'lulu' = #'lulu' → true
```

Appartenance à une classe

Plusieurs méthodes vous permettent de connaître la classe d'un objet.

class Vous pouvez demander à tout objet sa classe en utilisant le message class.

```
1 class → SmallInteger
```

Inversement, vous pouvez demander si un objet est une instance (isMemberOf:) d'une classe spécifique :

```

1 isMemberOf: SmallInteger → true "doit être précisément cette classe"
1 isMemberOf: Integer → false
1 isMemberOf: Number → false
1 isMemberOf: Object → false

```

Puisque Smalltalk est écrit en lui-même, vous pouvez vraiment naviguer au travers de sa structure en utilisant la bonne combinaison de messages superclass et class (voir le chapitre 13).

isKindOf: Object»isKindOf: répond true si la classe du receveur est la même ou une des sous-classes de la classe de l'argument.

```

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, qui est une Fraction, est aussi une sorte de nombre (Number), puisque la classe Number est une super-classe de la classe Fraction, mais 1/3 n'est pas un entier (Integer).

respondsTo: Object»respectsTo: répond true si le receveur comprend le message dont le sélecteur est passé en argument.

```
1 respectsTo: #, → false
```

C'est normalement une mauvaise idée de demander sa classe à un objet ou de lui demander quels messages il comprend. Au lieu de prendre des décisions basées sur la classe d'un objet, vous devriez simplement envoyer un message à cet objet et le laisser décider (c-à-d. sur la base de sa classe) comment il doit se comporter.

Copie

Copier des objets introduit quelques problèmes subtils. Puisque les variables d'instance sont accessibles par référence, une *copie superficielle*, les références portées par les variables d'instance devraient être partagées entre l'objet produit par la copie et l'objet original :

```

a1 := { { 'harry' } }.
a1 → #(#{'harry'})

```

```
a2 := a1 shallowCopy.
a2 → #( #'( 'harry' ))
(a1 at: 1) at: 1 put: 'sally'.
a1 → #( #'( 'sally' ))
a2 → #( #'( 'sally' )) "le tableau contenu est partagé"
```

Object»shallowCopy est une méthode primitive qui crée une copie superficielle d'un objet. Puisque a2 est seulement une copie superficielle de a1, les deux tableaux partagent une référence au tableau (Array) qu'ils contiennent.

Object»shallowCopy est une "interface publique" pour Object»copy et devrait être surchargée si les instances sont uniques. C'est le cas, par exemple, avec les classes Boolean, Character, SmallInteger, Symbol et UndefinedObject.

Object»copyTwoLevel est utilisée quand une simple copie superficielle ne suffit pas :

```
a1 := { { 'harry' } } .
a2 := a1 copyTwoLevel.
(a1 at: 1) at: 1 put: 'sally'.
a1 → #( #'( 'sally' ))
a2 → #( #'( 'harry' )) "état complètement indépendant"
```

Object»deepCopy effectue une copie profonde et arbitraire d'un objet.

```
a1 := { { { 'harry' } } } .
a2 := a1 deepCopy.
(a1 at: 1) at: 1 put: 'sally'.
a1 → #( #'( 'sally' ))
a2 → #( #'( #'( 'harry' )))
```

Le problème avec deepCopy est qu'elle ne se termine pas si elle est appliquée à une structure mutuellement récursive :

```
a1 := { 'harry' }.
a2 := { a1 }.
a1 at: 1 put: a2.
a1 deepCopy → ... ne se termine jamais
```

Même s'il est possible de surcharger deepCopy pour qu'elle fonctionne mieux, Object»copy offre une meilleure solution :

Méthode 8.7 – Modèle de méthode pour la copie d'objets

Object»copy

*"Answer another instance just like the receiver.
Subclasses typically override postCopy;
they typically do not override shallowCopy."*

↑self shallowCopy postCopy

Comme le dit le commentaire de la méthode, vous pouvez surcharger `postCopy` pour copier une variable d'instance qui ne devrait pas être partagée. `postCopy` doit toujours exécuter super `postCopy`.

Débogage

La méthode la plus importante ici est `halt`. Pour placer un point d'arrêt dans une méthode, il suffit d'insérer l'envoi de message `self halt` à une certaine position dans le corps de la méthode. Quand ce message est envoyé, l'exécution est interrompue et un débogueur s'ouvre à cet endroit de votre programme (voir le chapitre 6 pour plus de détails sur le débogueur).

Un autre message important est `assert`, qui prend un bloc comme argument. Si le bloc renvoie `true`, l'exécution se poursuit. Autrement une exception sera levée. Si cette exception n'est pas interceptée, le débogueur s'ouvrira à ce point pendant l'exécution. `assert` est particulièrement utile pour la *programmation par contrat*. L'utilisation la plus typique consiste à vérifier des pré-conditions non triviales pour des méthodes publiques. `Stack»pop` (`Stack` est la classe des piles) aurait pu aisément être implementée de la façon suivante (en commentaire de la méthode : "renvoie le premier élément et l'enlève de la pile") :

Méthode 8.8 – Vérifier une pré-condition

`Stack»pop`

"Return the first element and remove it from the stack."

`self assert: [self isEmpty not].`

`↑self linkedList removeFirst element`

Il ne faut pas confondre `Object»assert` avec `TestCase»assert`, méthode de l'environnement de test SUnit (voir le chapitre 7). Alors que la première attend un bloc en argument¹, la deuxième attend un Boolean. Même si les deux sont utiles pour déboguer, elles ont chacune un but très différent.

Gestion des erreurs

Ce protocole contient plusieurs méthodes utiles pour signaler les erreurs d'exécution.

Envoyer `self deprecated : unChaineExplicative` indique que la méthode courante ne devrait plus être utilisée si le paramètre `deprecation` a été activé dans le protocole `debug` du navigateur des préférences (Preference Brower). L'argument `String` devrait proposer une alternative.

1 `dolfNotNil: [:arg | arg printString, ' n'est pas nil']`

1. En fait, elle peut prendre n'importe quel argument qui comprend `value`, dont un Boolean.

→ *SmallInteger(Object)»doIfNotNil : has been deprecated. use ifNotNilDo :*

L'impression via `printIf` de la méthode précédente répond que l'usage de la méthode `doIfNotNil`: a été considéré comme désapprouvé (en anglais, *deprecated* ; *deprecation* signifiant désapprobation). Il est dit que nous devons plutôt utiliser `ifNotNilDo`:

`doesNotUnderstand:` est envoyé à chaque fois que la recherche d'un message échoue. L'implémentation par défaut, *c-à-d.* `Object»doesNotUnderstand:` déclenchera l'ouverture d'un débogueur à cet endroit. Il peut être utile de surcharger `doesNotUnderstand` : pour introduire un autre comportement.

`Object»error` et `Object»error:` sont des méthodes génériques qui peuvent être utilisées pour lever des exceptions (il est généralement préférable de lever vos propres exceptions, pour que vous puissiez distinguer les erreurs levées par votre code de celles levées par les classes du système).

Les méthodes abstraites en Smalltalk sont implémentées par convention avec le corps `self subclassResponsibility`. Si une classe abstraite est instanciée par accident, alors l'appel à une méthode abstraite provoquera l'évaluation de `Object»subclassResponsibility`.

Méthode 8.9 – Indiquer qu'une méthode est abstraite

`Object»subclassResponsibility`

"This message sets up a framework for the behavior of the class' subclasses.

Announce that the subclass should have implemented this message."

`self error: 'My subclass should have overridden ', thisContext sender selector`

`printString`

Son commentaire dit que "ce message installe un cadre pour le comportement des sous-classes de la classe. Il affirme que la sous-classe devrait avoir implémenté ce message". La phrase-argument de l'envoi du message d'erreur `error:` vous prévient que la méthode devra être surchargée dans une sous-classe concrète.

Magnitude, Number et Boolean sont des exemples classiques de classes abstraites que nous verrons rapidement dans ce chapitre.

`Number new + 1` → *Error : My subclass should have overridden #+*

`self shouldNotImplement` est envoyée par convention pour signaler qu'une méthode héritée est inappropriée pour cette sous-classe. C'est généralement le signe que quelque chose ne va pas dans la conception de la hiérarchie de classes. à cause des limitations de l'héritage simple, malgré tout, il est des fois très difficile d'éviter de telles solutions.

Un exemple classique est la méthode `Collection»remove:` qui est héritée de `Dictionary` mais marquée comme non implémentée (`Dictionary` fournit la méthode `removeKey:` à la place).

Test

Les méthodes de `test` n'ont aucun rapport avec SUnit ! Une méthode de test vous permet de poser une question sur l'état du receveur et retourne un booléen (Boolean).

De nombreuses méthodes de test sont fournies par Object. Nous avons déjà vu `isComplex`. Il existe également `isArray`, `isBoolean`, `isBlock`, `isCollection`, parmi d'autres. Généralement ces méthodes sont à éviter car demander sa classe à un objet est une forme de violation de l'encapsulation. Au lieu de tester la classe d'un objet, nous devrions simplement envoyer un message et laisser l'objet décider de sa propre réaction.

Cependant certaines de ces méthodes de test sont indéniablement utiles. Les plus utiles sont probablement `ProtoObject»isNil` et `Object»notNil` (bien que le patron de conception Null Object² permet d'éviter le besoin de ces méthodes également).

Initialisation

`initialize` est une méthode-clé qui ne se trouve pas dans Object mais dans ProtoObject. Comme le texte de commentaire de la méthode l'indique, vos sous-classes devront redéfinir cette méthode pour faire des initialisations dans la phase de création d'instance.

Méthode 8.10 – La méthode générique `initialize`

`ProtoObject»initialize`

"Subclasses should redefine this method to perform initializations on instance creation"

Ceci est important parce que, dans Pharo, la méthode `new`, définie pour chaque classe du système, envoie `initialize` aux instances nouvellement créées.

Méthode 8.11 – Modèle pour la méthode de classe `new`. Le commentaire dit : "Répond une nouvelle instance initialisée du receveur (qui est une classe) sans aucune variables indexées. Échoue si la classe est indexée"

`Behavior»new`

"Answer a new initialized instance of the receiver (which is a class) with no indexable variables. Fail if the class is indexable."

\uparrow `self basicNew initialize`

Ceci signifie qu'en surchargeant simplement la méthode générique `initialize`, les nouvelles instances de votre classe seront automatiquement initialisées. La méthode `initialize` devrait normalement exécuter `super initialize`

2. Bobby Woolf, Null Object. dans Robert Martin, Dirk Riehle et Frank Buschmann (éd.), Pattern Languages of Program Design 3. Addison Wesley, 1998.

pour établir les invariants de la classe pour toutes les variables d'instance héritées. Notons que ceci n'est *pas* le comportement standard dans les autres Smalltalks.

8.2 Les nombres

Il faut remarquer que les nombres en Smalltalk ne sont pas des données primitives mais de vrais objets. Bien sûr les nombres sont implémentés efficacement dans la machine virtuelle, mais la hiérarchie de la classe Number est aussi accessible et extensible que n'importe quelle autre portion de la hiérarchie de classe de Smalltalk.

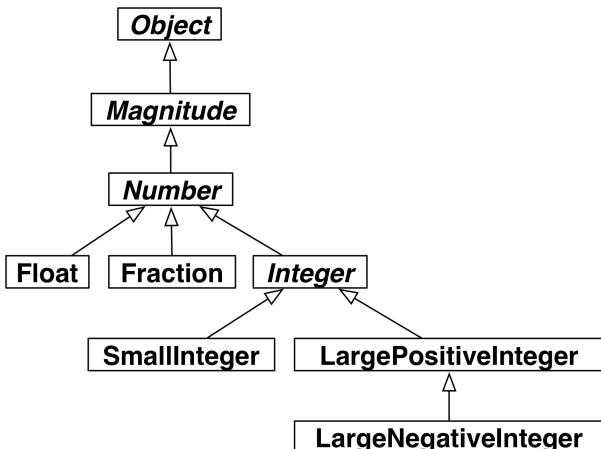


FIGURE 8.1 – La hiérarchie de la classe Number.

On trouve les nombres dans la catégorie *Kernel-Numbers*. La racine abstraite de cette catégorie est Magnitude, qui représente toutes les sortes de classes qui supportent les opérateurs de comparaison. La classe Number ajoute plusieurs opérateurs arithmétiques et autres, principalement des méthodes abstraites. Float et Fraction représentent, respectivement, les nombres à virgule flottante et les valeurs fractionnaires. Integer est également une classe abstraite et contient trois sous-classes SmallInteger, LargePositiveInteger et LargeNegativeInteger. Le plus souvent les utilisateurs n'ont pas à connaître la différence entre les trois classes d'entiers, car les valeurs sont automatiquement converties si besoin est.

Magnitude

Magnitude n'est pas seulement la classe parente des classes de nombres, mais également des autres classes supportant les opérateurs de comparaison, comme Character, Duration et Timespan (les nombres complexes (classe Complex) ne sont pas comparables et n'héritent pas de la classe Number).

Les méthodes < et = sont abstraites. Les autres opérateurs sont définis de manière générique. Par exemple :

Méthode 8.12 – Méthodes de comparaison abstraites

Magnitude» < aMagnitude

"Answer whether the receiver is less than the argument."

↑self subclassResponsibility

Magnitude» > aMagnitude

"Answer whether the receiver is greater than the argument."

↑aMagnitude < self

Number

De la même manière, la classe Number définit +, -, *, / comme des méthodes abstraites, mais tous les autres opérateurs arithmétiques sont définis de manière générique.

Tous les nombres supportent plusieurs opérateurs de conversion, comme asFloat et asInteger. Il existe également des constructeurs numériques, comme i, qui convertit une instance de Number en une instance de Complex avec une partie réelle nulle, ainsi que d'autres méthodes qui génèrent des durées, instances de Duration, comme hour, day et week (respectivement : heure, jour et semaine).

Les nombres supportent directement les *fonctions mathématiques* telles que sin, log, raiseTo: (puissance), squared (carré), sqrt (racine carrée).

Number»printOn: utilise la méthode abstraite Number»printOn:base: (la base par défaut est 10).

Les méthodes de test comprennent entre autres even (pair), odd (impair), positive (positif) et negative (négatif). Logiquement, Number surcharge isNumber (test d'appartenance à la hiérarchie de la classe des nombres). Plus intéressant, isInfinite (test d'infinité) renvoie false.

Les méthodes de *troncature* incluent entre autres, floor (arrondi à l'entier inférieur), ceiling (arrondi à l'entier supérieur), integerPart (partie entière), fractionPart (partie après la virgule).

1 + 2.5 → 3.5 "Addition de deux nombres"

3.4 * 5 → 17.0 "Multiplication de deux nombres"

8 / 2	→	4	"Division de deux nombres"
10 - 8.3	→	1.7	"Soustraction de deux nombres"
12 = 11	→	false	"Égalité entre deux nombres"
12 ~= 11	→	true	"Teste si deux nombres sont différents"
12 > 9	→	true	"Plus grand que"
12 >= 10	→	true	"Plus grand ou égal à"
12 < 10	→	false	"Plus petit que"
100@10	→	100@10	"Création d'un point"

L'exemple suivant fonctionne étonnamment bien en Smalltalk :

```
1000 factorial / 999 factorial → 1000
```

Notons que 1000 factorial est réellement calculée alors que dans beaucoup d'autres langages il peut être difficile de le faire. Ceci est un excellent exemple de conversion automatique et d'une gestion exacte des nombres.

 *Essayez d'afficher le résultat de 1000 factorial. Il faut plus de temps pour l'afficher que pour le calculer !*

Float

Float implémente les méthodes de Number abstraites pour les nombres à virgule flottante.

Plus intéressant, Float class (c-à-d. le côté classe de Float) contient des méthodes pour renvoyer les constantes : e, infinity (infini), nan (acronyme de *Not A Number* c-à-d. "n'est pas un nombre" : résultat d'un calcul numérique indéterminé) et pi.

Float pi	→	3.141592653589793
Float infinity	→	Infinity
Float infinity isNaN	→	true

Fraction

Les fractions sont représentées par des variables d'instance pour le numérateur et le dénominateur, qui devraient être des entiers. Les fractions sont normalement créées par division d'entiers (plutôt qu'en utilisant le constructeur Fraction»numerator:denominator:) :

```
6/8 → (3/4)
(6/8) class → Fraction
```

Multiplier une fraction par un entier ou par une autre fraction peut renvoyer un entier :

6/8 * 4 → 3

Integer

`Integer` est le parent abstrait de trois implémentations concrètes d'entiers. En plus de fournir une implémentation concrète de beaucoup de méthodes abstraites de la classe `Number`, il ajoute également quelques méthodes spécifiques aux entiers, telles que `factorial` (fractionnelle), `atRandom` (nombre aléatoire entre 1 et le receveur), `isPrime` (test de nombre premier), `gcd`: (le plus grand dénominateur commun) et beaucoup d'autres.

La classe `SmallInteger` est particulière dans le sens que ses instances sont représentées de manière compacte — au lieu d'être stockées comme référence, une instance de `SmallInteger` est directement représentée en utilisant les bits qui seraient normalement utilisés pour contenir la référence. Le premier bit de la référence à un objet indique si l'objet est une instance de `SmallInteger` ou non.

Les méthodes de classe `minVal` et `maxVal` nous donne la plage de valeurs d'une instance de `SmallInteger` :

```
SmallInteger maxVal = ((2 raisedTo: 30) - 1) → true
SmallInteger minVal = (2 raisedTo: 30) negated → true
```

Quand un `SmallInteger` dépasse cette plage de valeurs, il est automatiquement converti en une instance de `LargePositiveInteger` ou de `LargeNegativeInteger`, selon le besoin :

```
(SmallInteger maxVal + 1) class → LargePositiveInteger
(SmallInteger minVal - 1) class → LargeNegativeInteger
```

Les grands entiers sont de la même manière convertis en petits entiers quand il le faut.

Comme dans la plupart des langages de programmation, les entiers peuvent être utiles pour spécifier une itération. Il existe une méthode dédiée `timesRepeat:` pour l'évaluation répétitive d'un bloc. Nous avons déjà vu des exemples similaires dans le chapitre le chapitre 3 :

```
n := 2.
3 timesRepeat: [ n := n*n ].
n → 256
```

8.3 Les caractères

Character est défini dans la catégorie *Collections-Strings* comme une sous-classe de Magnitude. Les caractères imprimables sont représentés en Pharo par `$⟨caractère⟩`. Par exemple :

```
$a < $b → true
```

Les caractères non imprimables sont générés par différentes méthodes de classe. Character class»value: prend la valeur entière Unicode (ou ASCII) comme argument et renvoie le caractère correspondant. Le protocole *accessing untypeable characters* contient un certain nombre de constructeurs utiles tels que backspace (retour arrière), cr (retour-chariot), escape (échappement), euro (signe €), space (espace), tab (tabulation), parmi d'autres.

```
Character space = (Character value: Character space asciiValue) → true
```

La méthode printOn: est assez adroite pour savoir laquelle des trois manières utiliser pour générer les caractères de la façon la plus appropriée :

```
Character value: 1 → Character home
Character value: 2 → Character value: 2
Character value: 32 → Character space
Character value: 97 → $a
```

Il existe plusieurs méthodes de test utiles : isAlphaNumeric (si alphanumérique), isCharacter (si caractère), isDigit (si numérique), isLowercase, (si minuscule), isVowel (si voyelle non-accentuée, voir page 62), parmi d'autres.

Pour convertir un caractère en une chaîne de caractères contenant uniquement ce caractère, il faut lui envoyer le message asString. Dans ce cas asString et printString donnent des résultats différents :

```
$a asString → 'a'
$a → $a
$a printString → '$a'
```

Chaque caractère ASCII est une instance unique, stockée dans la variable de classe CharacterTable :

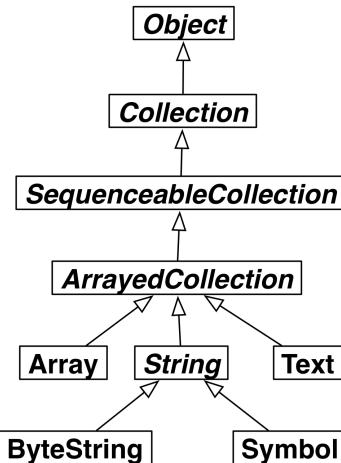
```
(Character value: 97) == $a → true
```

Cependant, les caractères au delà de la plage 0 à 255 ne sont pas uniques :

```
Character characterTable size → 256
(Character value: 500) == (Character value: 500) → false
```

8.4 Les chaînes de caractères

La classe `String` est également définie dans la catégorie *Collections-Strings*. Une chaîne de caractères est une collection indexée contenant uniquement des caractères.



La hiérarchie de `String`.

En fait, `String` est une classe abstraite et les chaînes de caractères de Pharo sont en réalité des instances de la classe concrète `ByteString`.

```
'Bonjour Squeak' class → ByteString
```

Une autre sous-classe importante de `String` est `Symbol`. La différence fondamentale est qu'il n'y a toujours qu'une instance unique de `Symbol` pour une valeur donnée (ceci est quelques fois appelé "la propriété de l'instance unique"). À l'opposé, deux chaînes construites séparément et contenant la même séquence de caractères seront souvent des objets différents.

```
'Sal','ut' == 'Salut' → false
```

```
('Sal','ut') asSymbol == #Salut → true
```

Une autre différence importante est que `String` est modifiable (mutable), alors que `Symbol` ne l'est pas.

```
'hello' at: 2 put: $u; yourself → 'hullo'
```

```
#hello at: 2 put: $u → erreur
```

Il est facile d'oublier que, puisque les chaînes de caractères sont des collections, elles comprennent les mêmes messages que les autres collections (ici, la méthode `indexOf:` de `Collections` donne la position du premier caractère rencontré) :

```
#hello indexOf: $o → 5
```

Bien que `String` n'hérite pas de `Magnitude`, la classe supporte les méthodes de *comparaison* , `<`, `=`, etc. De plus, `String»match:` est utile pour les recherches simples d'expressions régulières :

```
'*or*' match: 'zorro' → true
```

Si vous avez besoin d'un meilleur support pour les expressions régulières, vous pouvez jeter un coup œil sur le paquetage *Regex* de Vassili Bykov.

Les chaînes de caractères supportent un grand nombre de méthodes de conversion. Beaucoup sont des constructeurs-raccourci pour d'autres classes, comme `asDate` (pour créer une date) ou `asFileName` (pour créer un nom de fichier). Il existe également un certain nombre de méthodes utiles pour transformer une chaîne de caractères en une autre, comme `capitalized` (pour capitaliser) et `translateToLowerCase` (pour mettre en minuscule).

Pour plus d'informations sur les chaînes de caractères et les collections, rendez-vous au chapitre 9.

8.5 Les booléens

La classe `Boolean` offre un aperçu fascinant de la manière dont Smalltalk est construit autour de la bibliothèque de classes. `Boolean` est la super-classe abstraite des classes singltons (de patron *Singleton*) : `True` et `False`.

La plupart des comportements des booléens peuvent être compris en regardant la méthode `ifTrue:ifFalse:` (en français, `si vrai: si faux:`), qui prend deux blocs comme arguments.

```
(4 factorial > 20) ifTrue: [ 'plus grand' ] ifFalse: [ 'plus petit' ] → 'plus grand'
```

La méthode est abstraite dans `Boolean`. Les implémentations dans les sous-classes concrètes sont toutes les deux triviales :

Méthode 8.13 – *Implémentations de ifTrue :ifFalse :*

```
True»ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
```

```
↑trueAlternativeBlock value
```

```
False»ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
```

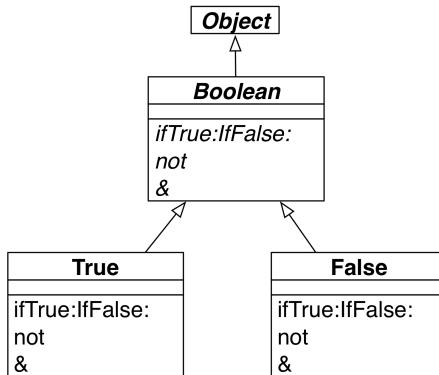


FIGURE 8.2 – La hiérarchie des booléens.

\uparrow falseAlternativeBlock value

En fait, ceci est l'essence même de la programmation orientée objet (POO) : quand un message est envoyé à un objet, l'objet lui-même détermine quelle méthode sera utilisée pour répondre. Dans ce cas, une instance de True évalue simplement l'alternative *vraie*, alors qu'une instance de False évalue l'alternative *fausse*. Toutes les méthodes abstraites de la classe Boolean sont implémentées de cette manière pour True et False. Par exemple :

Méthode 8.14 – Implémenter la négation

```
True>not
"Negation--answer false since the receiver is true."
↑false
```

Le commentaire de la méthode not (négation logique) nous informe que la réponse est toujours fausse (*false*) puisque le receveur est vrai (*true*, instance de True).

La classe Boolean offre plusieurs méthodes utiles, comme *ifTrue:*, *ifFalse:*, *ifFalse:ifTrue*. Vous avez également le choix entre les conjonctions et disjonctions optimisées ou paresseuses.

(1>2) & (3<4)	→	false	"doit évaluer les deux côtés"
(1>2) and: [3<4]	→	false	"évalue seulement le receveur"
(1>2) and: [(1/0) > 0]	→	false	"le bloc passé en argument n'est jamais évalué, ainsi, pas d'exception"

Dans le premier exemple, les deux sous-expressions booléennes sont évaluées, puisque *&* (et logique) prend un argument booléen. Dans le second et troisième exemple, uniquement la première est évaluée, car *and:* (et non-évaluant) attend un bloc comme argument. Le bloc est évalué uniquement si le premier argument vaut *true*.

💡 *Essayez d'imaginer comment and: et or: (ou non-évaluant) sont implémentés. Vérifiez les implementations dans Boolean, True et False.*

8.6 Résumé du chapitre

Nous avons vu que :

- si vous surchargez = alors vous devez également surcharger la méthode de hachage, hash ;
- surchargez postCopy pour implémenter correctement la copie de vos objets ;
- envoyez self halt pour créer un point d'arrêt ;
- renvoyez self subclassResponsibility pour faire une méthode abstraite ;
- pour donner la représentation en chaîne de caractères d'un objet String, vous devez surcharger printOn: ;
- surchargez la méthode générique initialize pour instancier correctement vos objets ;
- les méthodes de la classe Number assurent, si nécessaire, les conversions automatiques entre flottants, fractions et entiers ;
- les fractions représentent vraiment des nombres réels plutôt que des nombres à virgule flottante ;
- les caractères sont des instances uniques ;
- les chaînes de caractères sont modifiables (mutables) mais les symboles ne le sont pas ; cependant faites attention à ne pas modifier les chaînes de caractères littérales !
- ces symboles sont uniques mais les chaînes de caractères ne le sont pas ;
- les chaînes de caractères et les symboles sont des collections et donc, supportent les méthodes usuelles de la classe Collection.

Chapitre 9

Les collections

9.1 Introduction

Les classes de collections forment un groupe de sous-classes de `Collection` et de `Stream` (pour flux de données) faiblement couplées destiné à un usage générique. Ce groupe de classes mentionné dans la bible de Smalltalk nommée “Blue Book”¹ (le fameux livre bleu) comprend 17 sous-classes de `Collection` et 9 issues de la classe `Stream`. Formant un total de 28 classes, elles ont déjà été remodelées maintes fois avant la sortie du système Smalltalk-80. Ce groupe de classes est souvent considéré comme un exemple pragmatique de modélisation orientée objet.

Dans Pharo, les classes abstraites `Collection` et `Stream` disposent respectivement de 101 et de 50 sous-classes mais beaucoup d’entre elles (comme `Bitmap`, `FileStream` et `CompiledMethod`) sont des classes d’usage spécifique définies pour être employées dans d’autres parties du système ou dans des applications et ne sont par conséquent pas organisées dans la catégorie “Collections”. Dans ce chapitre, nous réunirons `Collection` et ses 47 sous-classes aussi présentes dans les catégories-système de la forme `Collections-*` sous le terme de “hiérarchie de Collections” et `Stream` et ses 9 sous-classes de la catégorie `Collections-Streams` sous celui de “hiérarchie de Streams”. Ces 56 classes répondent à 982 messages définissant un total de 1609 méthodes !

Dans ce chapitre, nous nous attarderons principalement sur le sous-ensemble de classes de collections montré sur la figure 9.1. Les flux de données ou *streams* seront abordés séparément dans le chapitre 10.

1. Adele Goldberg et David Robson, *Smalltalk 80 : the Language and its Implementation*. Reading, Mass.: Addison Wesley, mai 1983, ISBN 0-201-13688-0.

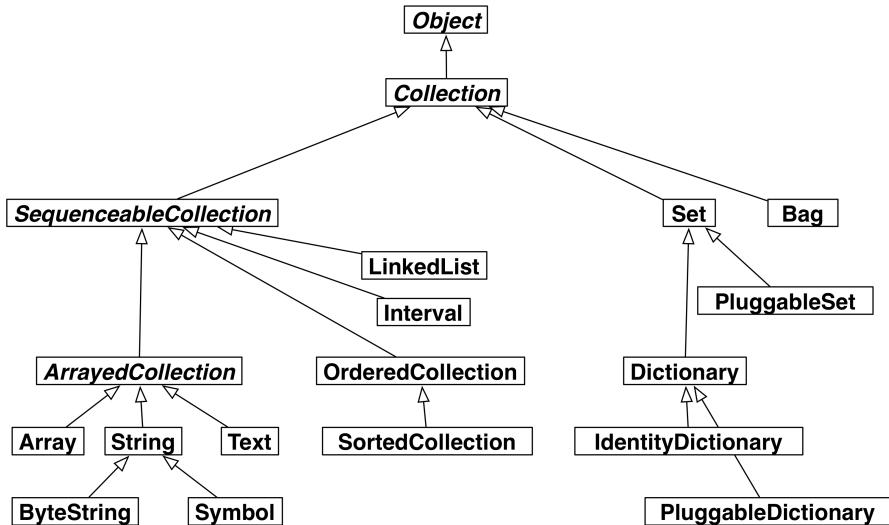


FIGURE 9.1 – Certaines des classes majeures de collection de Pharo.

9.2 Des collections très variées

Pour faire bon usage des classes de collections, le lecteur devra connaître au moins superficiellement l'immense variété de collections que celles-ci implémentent ainsi que leurs similitudes et leurs différences.

Programmer avec des collections plutôt qu'avec des éléments indépendants est une étape importante pour accroître le degré d'abstraction d'un programme. La fonction `map` dans le langage Lisp est un exemple primaire de cette technique de programmation : cette fonction applique une fonction entrée en argument à tout élément d'une liste et retourne une nouvelle liste contenant le résultat. Smalltalk-80 a adopté la programmation basée sur les collections comme précepte central. Les langages modernes de programmation fonctionnelle tels que ML et Haskell ont suivi l'orientation de Smalltalk.

Pourquoi est-ce une si bonne idée ? Partons du principe que nous avons une structure de données contenant une collection d'enregistrements d'étudiants appelé `students` (pour étudiants, en anglais) et que nous voulons accomplir une certaine action sur tous les étudiants remplissant un certain critère. Les programmeurs éduqués aux langages impératifs vont se retrouver immédiatement à écrire une boucle. Mais le développeur en Smalltalk écrira :

```
students select: [ :each | each gpa < threshold ]
```

ce qui donnera une nouvelle collection contenant précisement les éléments de students (étudiants) pour lesquels la fonction entre crochets renvoie une réponse positive *c-à-d.* true². Le code Smalltalk a la simplicité et l'élégance des langages dédiés ou *Domain-Specific Language* souvent abrégés en DSL.

Le message select: est compris par toutes les collections de Smalltalk. Il n'est pas nécessaire de chercher si la structure de données des étudiants est un tableau ou une liste chaînée : le message select: est reconnu par les deux. Notez donc que c'est assez différent de l'usage d'une boucle avec laquelle nous devons nous interroger pour savoir si students est un tableau ou une liste chaînée avant que cette boucle puisse être configurée.

En Smalltalk, lorsque quelqu'un parle d'une collection sans être plus précis sur le type de la collection, il mentionne un objet qui supporte des protocoles bien définis pour tester l'appartenance et énumérer les éléments. Toutes les collections acceptent les messages de la catégorie des tests nommée *testing* tels que includes: (test d'inclusion), isEmpty (teste si la collection est vide) et occurrencesOf: (test d'occurrences d'un élément). Toutes les collections comprennent les messages du protocole *enumeration* comme do: (action sur chaque élément), select: (sélection de certains éléments), reject: (rejet à l'opposé de select:), collect: (identique à la fonction map de Lisp), detect:ifNone: (déttection tolérante à l'absence) inject:into: (accumulation ou opération par réduction comme avec une fonction fold ou reduce dans d'autres langages) et beaucoup plus encore. C'est plus l'ubiquité de ce protocole que sa diversité qui le rend si puissant.

La figure 9.2 résume les protocoles standards supportés par la plupart des classes de la hiérarchie de collections. Ces méthodes sont définies, redéfinies, optimisées ou parfois même interdites par les sous-classes de Collections.

Au-delà de cette homogénéité apparente, il y a différentes sortes de collections soit, supportant des protocoles différents soit, offrant un comportement différent pour une même requête. Parcourons brièvement certaines de ces divergences essentielles :

- **Les séquentielles ou Sequenceable** : les instances de toutes les sous-classes de SequenceableCollection débutent par un premier élément dit first et progresse dans un ordre bien défini jusqu'au dernier élément dit last. Les instances de Set, Bag (ou multiensemble) et Dictionary ne sont pas des collections séquentielles.
- **Les triées ou Sortable** : une SortedCollection maintient ses éléments dans un ordre de tri.
- **Les indexées ou Indexable** : la majorité des collections séquentielles sont aussi indexées, *c-à-d.* que ses éléments peuvent être extraits par at: qui peut se traduire par l'expression "à l'endroit indiqué". Le tableau Array est une structure de données indexées familière avec une taille

2. L'expression entre crochets (brackets en anglais) peut être vue comme une expression λ définissant une fonction anonyme $\lambda x.x \text{ gpa} < \text{threshold}$.

Protocole	Méthodes
<i>accessing</i>	<i>size, capacity, at : anIndex, at : anIndex put : anElement</i>
<i>testing</i>	<i>isEmpty, includes : anElement, contains : aBlock, occurrencesOf : anElement</i>
<i>adding</i>	<i>add : anElement, addAll : aCollection</i>
<i>removing</i>	<i>remove : anElement, remove : anElement ifAbsent : aBlock, removeAll : aCollection</i>
<i>enumerating</i>	<i>do : aBlock, collect : aBlock, select : aBlock, reject : aBlock, detect : aBlock, detect : aBlock ifNone : aNoneBlock, inject : aValue into : aBinaryBlock</i>
<i>converting</i>	<i>asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection : aBlock</i>
<i>creation</i>	<i>with : anElement, with :with :, with :with :with :, with :with :with :, withAll : aCollection</i>

FIGURE 9.2 – Les protocoles standards de collections

fixe ; *anArray at: n* récupère le n^{e} élément de *anArray* alors que, *anArray at: n put: v* change le n^{e} élément par *v*. Les listes chaînées de classe *LinkedList* et les listes à enjambements de classe *SkipList* sont séquentielles mais non-indexées ; autrement dit, elles acceptent *first* et *last*, mais pas *at*.

- **Les collections à clés ou *Keyed*** : les instances du dictionnaire *Dictionary* et ses sous-classes sont accessibles via des clés plutôt que par des indices.
- **Les collections modifiables ou *Mutable*** : la plupart des collections sont dites *mutables c-à-d.* modifiables, mais les intervalles *Interval* et les symboles *Symbol* ne le sont pas. Un *Interval* est une collection non-modifiable ou *immutable* représentant une rangée d'entiers *Integer*. Par exemple, *5 to: 16 by: 2* est un intervalle *Interval* qui contient les éléments 5, 7, 9, 11, 13 et 15. Il est indexable avec *at:* mais ne peut pas être changé avec *at:put:..*
- **Les collections extensibles** : les instances d'*Interval* et de *Array* sont toujours de taille fixe. D'autres types de collections (les collections triées *SortedCollection*, ordonnées *OrderedCollection* et les listes chaînées *LinkedList*) peuvent être étendues après leur création.
La classe *OrderedCollection* est plus générale que le tableau *Array* ; la taille d'une *OrderedCollection* grandit à la demande et elle a aussi bien des méthodes d'ajout en début *addFirst:* et en fin *addLast:* que des méthodes *at:* et *at:put:..*
- **Les collections à *duplicat*** : un *Set* filtrera les *duplicata* ou doublons mais un *Bag* (sac, en français) ne le fera pas. Les collections non-ordonnées *Dictionary*, *Set* et *Bag* utilisent la méthode = fournie par les éléments ; les variantes *Identity* de ces classes (*IdentityDictionary*, *IdentitySet*)

Collections en tableaux (Arrayed)	Collections ordonnées (Ordered)	Collections à hachage (Hashed)	Collections chaînées (Linked)	Collections à intervalles (Interval)
Array String Symbol	OrderedCollection SortedCollection Text Heap	Set IdentitySet PluggableSet Bag IdentityBag Dictionary IdentityDictionary PluggableDictionary	LinkedList SkipList	Interval

FIGURE 9.3 – Certaines classes de collections rangées selon leur technique d'implémentation.

et IdentityBag) utilisent la méthode `==` qui teste si les arguments sont le même objet et les variantes Pluggable emploient une équivalence arbitraire définie par le créateur de la collection.

- **Les collections hétérogènes :** La plupart des collections stockent n'importe quel type d'élément. Un String, un CharacterArray ou Symbol ne contiennent cependant que des caractères de classe Character. Un Array pourra inclure un mélange de différents objets mais un tableau d'octets ByteArray ne comprendra que des octets Byte ; tout comme un IntegerArray n'a que des entiers Integers et qu'un FloatArray ne peut contenir que des réels à virgule flottante de classe Float. Une liste chaînée LinkedList est contrainte à ne pouvoir contenir que des éléments qui sont conformes au protocole *Link ▷ accessing*.

9.3 Les implémentations des collections

Considérer ces catégorisations par fonctionnalité n'est pas suffisant ; nous devons aussi regarder les classes de collections selon leur implémentation. Comme nous le montre la figure 9.3, cinq techniques d'implementations majeures sont employées.

1. Les tableaux ou Arrays stockent leurs éléments dans une variable d'instance indexable de l'objet collection lui-même ; dès lors, les tableaux doivent être de taille fixe mais peuvent être créés avec une simple allocation de mémoire.
2. Les collections ordonnées OrderedCollection et triées SortedCollection contiennent leurs éléments dans un tableau qui est référencé par une des variables d'instance de la collection. En conséquence, le tableau interne peut être remplacé par un plus grand si la collection grossit au delà des capacités de stockage.
3. Les différents types d'ensemble (ou set) et les dictionnaires sont aussi

référencés par un tableau de stockage subsidiaire mais ils utilisent ce tableau comme une table de hachage (ou *hash table*). Les ensembles dits sacs ou *bags* (de classe `Bag`) utilisent un dictionnaire `Dictionary` pour le stockage avec pour clés des éléments du `Bag` et pour valeurs leur nombre d'occurrences.

4. Les listes chaînées `LinkedList` utilisent une représentation standard simplement chaînée.
5. Les intervalles `Interval` sont représentés par trois entiers qui enregistrent les deux points extrêmes et la taille de pas.

En plus de ces classes, il y a aussi les variantes de `Array`, de `Set` et de plusieurs sortes de dictionnaires dites à liaisons faibles ou “weak”. Ces collections maintiennent faiblement leurs éléments, *c-à-d.* de manière à ce qu’elles n’empêchent pas ses éléments d’être recyclés par le ramasse-miettes ou *garbage collector*. La machine virtuelle Pharo est consciente de ces classes et les gère d’une façon particulière.

Les lecteurs intéressés dans l’apprentissage avancé des collections de Smalltalk sont renvoyés à la lecture de l’excellent livre de LaLonde et Pugh³.

9.4 Exemples de classes importantes

Nous présentons maintenant les classes de collections les plus communes et les plus importantes via des exemples de code simples. Les protocoles principaux de collections sont : `at:`, `at:put:` — pour accéder à un élément, `add:`, `remove:` — pour ajouter ou enlever un élément, `size`, `isEmpty`, `include:` — pour obtenir des informations respectivement sur la taille, la virginité (collection vide) et l’inclusion dans la collection, `do:, collect:, select:` — pour agir en itérations à travers la collection. Chaque collection implémente ou non de tels protocoles et quand elle le fait, elle les interprète pour être en adéquation avec leurs sémantiques. Nous vous suggérons de naviguer dans les classes elles-même pour identifier par vous-même les protocoles spécifiques et plus avancés.

Nous nous focaliserons sur les classes de collections les plus courantes : `OrderedCollection`, `Set`, `SortedCollection`, `Dictionary`, `Interval` et `Array`.

Les protocoles communs de création. Il existe plusieurs façons de créer des instances de collections. La technique la plus générale consiste à utiliser les méthodes `new:` et `with:.. new:` `anInteger` crée une collection de taille `anInteger` dont les éléments seront tous nuls *c-à-d.* de valeur `nil`. `with: anObject` crée une

3. Wilf LaLonde et John Pugh, *Inside Smalltalk : Volume 1*. Prentice Hall, 1990, ISBN 0-13-468414-1.

collection et ajoute anObject à la collection créée. Les collections réalisent cela de différentes manières.

Vous pouvez créer des collections avec des éléments initiaux en utilisant les méthodes with:, with:with: etc ; et ce jusqu'à six éléments (donc six with:).

```
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)
```

Vous pouvez aussi utiliser la méthode addAll: pour ajouter tous les éléments d'une classe à une autre :

```
(1 to: 5) asOrderedCollection addAll: '678'; yourself → an OrderedCollection(1 2 3
4 5 $6 $7 $8)
```

Prenez garde au fait que addAll: renvoie aussi ses arguments et non pas le receveur !

Vous pouvez aussi créer plusieurs collections avec les méthodes withAll: ou newFrom:

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)

Notez que ces méthodes ne sont pas identiques. En particulier, Dictionary class »withAll: interprète ses arguments comme un collection de valeurs alors que Dictionary class»newFrom: s'attend à une collection d'associations.

Le tableau Array

Un tableau Array est une collection de taille fixe dont les éléments sont accessibles par des indices entiers. Contrairement à la convention établie

dans le langage C, le premier élément d'un tableau Smalltalk est à la position 1 et non à la position 0. Le protocole principal pour accéder aux éléments d'un tableau est la méthode `at:` et la méthode `at:put:.` `at: anInteger` renvoie l'élément à l'index `anInteger`. `at: anInteger put: anObject` met `anObject` à l'index `anInteger`. Comme les tableaux sont des collections de taille fixe nous ne pouvons pas ajouter ou enlever des éléments à la fin du tableau. Le code suivant crée un tableau de taille 5, place des valeurs dans les 3 premières cases et retourne le premier élément.

```
anArray := Array new: 5.
anArray at: 1 put: 4.
anArray at: 2 put: 3/2.
anArray at: 3 put: 'ssss'.
anArray at: 1 → 4
```

Il y a plusieurs façons de créer des instances de la classe `Array`. Nous pouvons utiliser `new:.`, `with:.` et les constructions basées sur `#()` et `{ }.`.

Création avec new: `new: anInteger` crée un tableau de taille `anInteger`. `Array new: 5` crée un tableau de taille 5.

Création avec with: les méthodes `with:.` permettent de spécifier la valeur des éléments. Le code suivant crée un tableau de trois éléments composés du nombre 4, de la fraction 3/2 et de la chaîne de caractères 'lulu'.

```
Array with: 4 with: 3/2 with: 'lulu' → {4 . (3/2) . 'lulu'}
```

Création littéral avec #(). `#()` crée des tableaux littéraux avec des éléments statiques qui doivent être connus quand l'expression est compilée et non lorsqu'elle est exécutée. Le code suivant crée un tableau de taille 2 dans lequel le premier élément est le nombre 1 et le second la chaîne de caractères 'here' : tous deux sont des littéraux.

```
#{1 'here'} size → 2
```

Si vous évaluez désormais `#{1+2}`, vous n'obtenez pas un tableau avec un unique élément 3 mais vous obtenez plutôt le tableau `#{1 #+ 2}` c-à-d. avec les trois éléments : 1, le symbole `#+` le chiffre 2.

```
#{1+2} → #{1 #+ 2}
```

Ceci se produit parce que la construction `#()` fait que le compilateur interprète littéralement les expressions contenues dans le tableau. L'expression est analysée et les éléments résultants forment un nouveau tableau. Les tableaux littéraux contiennent des nombres, l'élément `nil`, des booléens `true` et `false`, des symboles et des chaînes de caractères.

Création dynamique avec { }. Vous pouvez finalement créer un tableau dynamique en utilisant la construction suivante : {}. { a . b } est équivalent à Array with : a with : b. En particulier, les expressions incluses entre { et } sont exécutées. Chaque expression est séparée de la précédente par un point.

```
{ 1 + 2 } → #(3)
{(1/2) asFloat} at: 1 → 0.5
{10 atRandom . 1/3} at: 2 → (1/3)
```

L'accès aux éléments. Les éléments de toutes les collections séquentielles peuvent être accédés avec les messages at: et at:put:.

```
anArray := #(1 2 3 4 5 6) copy.
anArray at: 3 → 3
anArray at: 3 put: 33.
anArray at: 3 → 33
```

Soyez attentif au fait que le code modifie les tableaux littéraux ! Le compilateur essaie d'allouer l'espace nécessaire aux tableaux littéraux. à moins que vous ne copiez le tableau, la seconde fois que vous évaluez le code, votre tableau "littéral" pourrait ne pas avoir la valeur que vous attendez. (sans clonage, la seconde fois, le tableau littéral #(1 2 3 4 5 6) sera en fait #(1 2 33 4 5 6) !) Les tableaux dynamiques n'ont pas ce problème.

La collection ordonnée OrderedCollection

OrderedCollection est une des collections qui peut s'étendre et auxquelles des éléments peuvent être adjoints séquentiellement. Elle offre une variété de méthodes telles que add:, addFirst:, addLast: et addAll:.

```
ordCol := OrderedCollection new.
ordCol add: 'Seaside'; add: 'SqueakSource'; addFirst: 'Monticello'.
ordCol → an OrderedCollection('Monticello' 'Seaside' 'SqueakSource')
```

Effacer des éléments. La méthode remove: anObject efface la première occurrence d'un objet dans la collection. Si la collection n'inclut pas l'objet, elle lève une erreur.

```
ordCol add: 'Monticello'.
ordCol remove: 'Monticello'.
ordCol → an OrderedCollection('Seaside' 'SqueakSource' 'Monticello')
```

Il y a une variante de remove: nommée remove:ifAbsent: qui permet de spécifier comme second argument un bloc exécuté dans le cas où l'élément à effacer n'est pas dans la collection.

```
res := ordCol remove: 'zork' ifAbsent: [33].
res → 33
```

La conversion. Il est possible d'obtenir une collection ordonnée OrderedCollection depuis un tableau Array (ou n'importe quelle autre collection) en envoyant le message asOrderedCollection :

```
#(1 2 3) asOrderedCollection → an OrderedCollection(1 2 3)
'hello' asOrderedCollection → an OrderedCollection($h $e $l $l $o)
```

L'intervalle Interval

La classe Interval représente une suite de nombres. Par exemple, l'intervalle compris entre 1 et 100 est défini comme suit :

```
Interval from: 1 to: 100 → (1 to: 100)
```

L'imprimé ou l'affichage en mode printString de cet intervalle nous révèle que la classe nombre Number (représentant les nombres) dispose d'une méthode de convenance appelée to: (dans le sens de l'expression "jusqu'à") pour générer les intervalles :

```
(Interval from: 1 to: 100) = (1 to: 100) → true
```

Nous pouvons utiliser Interval class>from:to:by: (mot à mot : depuis-jusque-par) ou Number>to:by: (jusque-par) pour spécifier le pas entre les deux nombres comme suit :

```
(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)
```

Le dictionnaire Dictionary

Les dictionnaires sont des collections importantes dont les éléments sont accessibles via des clés. Parmi les messages de dictionnaire les plus couramment utilisés, vous trouverez at:, at:put:, at:ifAbsent:, keys et values (*keys* et *values* sont les mots anglais pour clés et valeurs respectivement).

```
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}

Les dictionnaires comparent les clés par égalité. Deux clés sont considérées comme étant la même si elles retournent *true* lorsqu'elles sont comparées par `=`. Une erreur commune et difficile à identifier est d'utiliser un objet dont la méthode `=` a été redéfinie mais pas sa méthode de hachage `hash`. Ces deux méthodes sont utilisées dans l'implémentation du dictionnaire et lorsque des objets sont comparés.

La classe `Dictionary` illustre clairement que la hiérarchie de collections est basée sur l'héritage et non sur du sous-typage. Même si `Dictionary` est une sous-classe de `Set`, nous ne voudrions normalement pas utiliser un `Dictionary` là où un `Set` est attendu. Dans son implémentation pourtant un `Dictionary` peut clairement être vu comme étant constitué d'un ensemble d'associations de valeurs et de clés créé par le message `->`. Nous pouvons créer un `Dictionary` depuis une collection d'associations ; nous pouvons aussi convertir un dictionnaire en tableau d'associations.

```
colors := Dictionary newFrom: { #blue->Color blue . #red->Color red . #yellow->Color yellow }.
colors removeKey: #blue.
colors associations → {#yellow->Color yellow . #red->Color red}
```

IdentityDictionary. Alors qu'un dictionnaire utilise le résultat des messages `=` et `hash` pour déterminer si deux clés sont identiques, la classe `IdentityDictionary` utilise l'identité (*c*-à-*d*. le message `==`) de la clé au lieu de celle de ses valeurs, *c*-à-*d*. qu'il considère deux clés comme égales *seulement* si elles sont le même objet.

Souvent les symboles de classe `Symbol` sont utilisés comme clés, dans les cas où le choix de `IdentityDictionary` s'impose, car un symbole est toujours certain d'être globalement unique. Si d'un autre côté, vos clés sont des chaînes de caractères `String`, il est préférable d'utiliser un `Dictionary` sinon vous pourriez avoir des ennuis :

```
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'
```

Comme `a` et `b` sont des objets différents, ils sont traités comme des objets différents. Le littéral `'foobar'` est alloué une seule fois et ce n'est vraiment pas

le même objet que a. Vous ne voulez pas que votre code dépende d'un tel comportement ! Un simple Dictionary vous donnerait la même valeur pour n'importe quelle clé égale à 'foobar'.

Vous ne vous tromperez pas en utilisant seulement des Symbols comme clé d'IdentityDictionary et des Strings (ou d'autres objets) comme clé de Dictionary classique.

Notez que l'objet global Smalltalk est une instance de SystemDictionary sous-classe de IdentityDictionary ; de ce fait, toutes ses clés sont des Symbols (en réalité, des symboles de la classe ByteSymbol qui contiennent des caractères de 8 bits).

```
Smalltalk keys collect: [ :each | each class ] → a Set(ByteSymbol)
```

Envoyer keys ou values à un Dictionary nous renvoie un ensemble Set ; nous explorerons cette collection dans la section qui suit.

L'ensemble Set

La classe Set est une collection qui se comporte comme un ensemble dans le sens mathématique c-à-d. comme une collection sans doublons et sans aucun ordre particulier. Dans un Set, les éléments sont ajoutés en utilisant le message add: (signifiant "ajoute" en anglais) et ils ne peuvent pas être accessibles par le message de recherche par indice at:. Les objets à inclure dans Set doivent implémenter les méthodes hash et =.

```
s := Set new.  
s add: 4/2; add: 4; add:2.  
s size → 2
```

Vous pouvez aussi créer des ensembles via Set class»newFrom: ou par le message de conversion Collection»asSet :

```
(Set newFrom: #( 1 2 3 1 4 )) = #(1 2 3 4 3 2 1) asSet → true
```

La méthode asSet offre une façon efficace pour éliminer les doublons dans une collection :

```
{ Color black. Color white. (Color red + Color blue + Color green) } asSet size → 2
```

Notez que rouge (message red) + bleu (message blue) + vert (message green) donne du blanc (message white).

Une collection Bag ou sac est un peu comme un Set qui autorise le duplificate :

```
{ Color black. Color white. (Color red + Color blue + Color green) } asBag size → 3
```

Les opérations sur les ensembles telles que l'*union*, l'*intersection* et le test d'*appartenance* sont implémentées respectivement par les messages de Collection `union:`, `intersection:` et `includes:`. Le receveur est d'abord converti en un Set, ainsi ces opérations fonctionnent pour toute sorte de collections !

```
(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
```

Comme nous l'avons expliqué plus haut les éléments de Set sont accessibles en utilisant des *méthodes d'itérations (itérateurs)* (voir la section 9.5).

La collection triée SortedCollection

Contrairement à une collection ordonnée `OrderedCollection`, une `SortedCollection` maintient ses éléments dans un ordre de tri. Par défaut, une collection triée utilise le message `<=` pour établir l'ordre du tri, autrement dit, elle peut trier des instances de sous-classes de la classe abstraite `Magnitude` qui définit le protocole d'objets comparables (`<`, `=`, `>`, `>=`, `between:and:...`). (voir le chapitre 8.)

Vous pouvez créer une `SortedCollection` en créant une nouvelle instance et en lui ajoutant des éléments :

```
SortedCollection new add: 5; add: 2; add: 50; add: -10; yourself. → a
SortedCollection(-10 2 5 50)
```

Le message `asSortedCollection` nous offre une bonne technique de conversion souvent utilisée.

```
#(5 2 50 -10) asSortedCollection → a SortedCollection(-10 2 5 50)
```

Cet exemple répond à la FAQ suivante :

FAQ : Comment trier une collection ?
RÉPONSE : En lui envoyant le message `asSortedCollection`.

```
'hello' asSortedCollection → a SortedCollection($e $h $l $l $o)
```

Comment retrouver une chaîne de caractères `String` depuis ce résultat ? Malheureusement `asString` retourne une représentation descriptive en `printString` ; ce n'est bien sûr pas ce que nous voulons :

```
'hello' asSortedCollection asString → 'a SortedCollection($e $h $l $l $o)'
```

La bonne réponse est d'utiliser les messages de classe String class>newFrom: ou String class>withAll: ; ou bien le message de conversion générique Object>as: :

'hello' asSortedCollection as: String	→	'ehllo'
String newFrom: ('hello' asSortedCollection)	→	'ehllo'
String withAll: ('hello' asSortedCollection)	→	'ehllo'

Avoir différents types d'éléments dans une SortedCollection est possible tant qu'ils sont comparables. Par exemple nous pouvons mélanger différentes sortes de nombres tels que des entiers, des flottants et des fractions :

```
{ 5. 2/-3. 5.21 } asSortedCollection → a SortedCollection((-2/3) 5 5.21)
```

Imaginez que vous vouliez trier des objets qui ne définissent pas la méthode <= ou que vous vouliez trier selon un critère bien spécifique. Vous pouvez le faire en spécifiant un bloc à deux arguments. Par exemple, la classe de couleur Color n'est pas une Magnitude et ainsi il n'implémente pas <= mais nous pouvons établir un bloc signalant que les couleurs devrait être triées selon leur luminance (une mesure de la brillance).

```
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)
```

La chaîne de caractères String

Un String en Smalltalk représente une collection de Characters. Il est séquentiel, indexé, modifiable (*mutable*) et homogène, ne contenant que des instances de Character. Comme Array, String a une syntaxe dédiée et est créée normalement en déclarant directement une chaîne de caractères littérale avec de simples guillemets (symbole *apostrophe* sur votre clavier), mais les méthodes habituelles de création de collection fonctionnent aussi.

'Hello'	→	'Hello'
String with: \$A	→	'A'
String with: \$h with: \$i with: \$!	→	'hi!'
String newFrom: #(\$h \$e \$l \$l \$o)	→	'hello'

En fait, String est abstrait. Lorsque vous instanciez un String, vous obtenez en réalité soit un ByteString en 8 bits ou un WideString⁴ en 32 bits. Pour simplifier, nous ignorons habituellement la différence et parlons simplement d'instances de String.

Deux instances de String peuvent être concaténées avec une virgule (en anglais, *comma*).

4. Wide a le sens : étendu

```
s := 'no', ',', 'worries'.
s → 'no worries'
```

Comme une chaîne de caractères est modifiable nous pouvons aussi la changer en utilisant la méthode `at:put:`.

```
s at: 4 put: $h; at: 5 put: $u.
s → 'no hurries'
```

Notez que la méthode virgule est définie dans la classe `Collection`. Elle marche donc pour n'importe quelle sorte de collections !

```
(1 to: 3) , '45' → #(1 2 3 $4 $5)
```

Nous pouvons aussi modifier une chaîne de caractères existante en utilisant les méthodes `replaceAll:with:` (pour remplacer tout avec quelque chose d'autre) ou `replaceFrom:to:with:` (pour remplacer depuis tant jusqu'à un certain point par quelque chose) comme nous pouvons le voir ci-dessous. Notez que le nombre de caractères et l'intervalle doivent être de la même taille.

```
s replaceAll: $n with: $N.
s → 'No hurries'
s replaceFrom: 4 to: 5 with: 'wo'.
s → 'No worries'
```

D'une manière différente, `copyReplaceAll:` crée une nouvelle chaîne de caractères (curieusement, les arguments dans ce cas sont des sous-chaînes et non des caractères indépendants et leur taille n'a pas à être identique).

```
s copyReplaceAll: 'rries' with: 'mbats' → 'No wombats'
```

Un rapide aperçu de l'implémentation de ces méthodes nous révèle qu'elles ne sont pas seulement définies pour les instances de `String`, mais également pour toutes sortes de collections séquentielles `SequenceableCollection`; du coup, l'expression suivante fonctionne aussi :

```
(1 to: 6) copyReplaceAll: (3 to: 5) with: { 'three'. 'etc.' } → #(1 2 'three' 'etc.' 6)
```

Appariement de chaînes de caractères Il est possible de demander si une chaîne de caractères s'apparie à une expression-filtre ou *pattern* en envoyant le message `match:.` Ce *pattern* ou filtre peut spécifier `*` pour comparer une série arbitraire de caractères et `#` pour représenter un simple caractère quelconque. Notez que `match:.` est envoyé au filtre et non pas à la chaîne de caractères à apparier.

```
'Linux *' match: 'Linux mag' → true
'GNU/Linux #ag' match: 'GNU/Linux tag' → true
```

`findString:` est une autre méthode utile.

```
'GNU/Linux mag' findString: 'Linux'      → 5
'GNU/Linux mag' findString: 'linux' startingAt: 1 caseSensitive: false → 5
```

Des techniques d'appariements plus avancées par filtre offrant les mêmes possibilités que Perl sont disponibles dans le paquetage [Regex](#).

Quelques essais avec les chaînes de caractères. L'exemple suivant illustre l'utilisation de `isEmpty`, `includes:` et `anySatisfy:` (ce dernier spécifiant si la collection satisfait le test passé en argument-bloc, au moins en un élément) ; ces messages ne sont pas seulement définis pour `String` mais plus généralement pour toute collection.

```
'Hello' isEmpty. → false
'Hello' includes: $a → false
'JOE' anySatisfy: [:c | c isLowercase] → false
'Joe' anySatisfy: [:c | c isLowercase] → true
```

Les gabarits ou *String templating*. Il y a 3 messages utiles pour gérer les gabarits ou templating : `format:`, `expandMacros` et `expandMacrosWith::`.

```
{1} est {2}' format: {'Pharo' . 'extra'} → 'Pharo est extra'
```

Les messages de la famille `expandMacros` offre une substitution de variables en utilisant `<n>` pour le retour-charriot, `<t>` pour la tabulation, `<1s>`, `<2s>`, `<3s>` pour les arguments (`<1p>`, `<2p>` entourent la chaîne avec des simples guillemets), et `<1?value1:value2>` pour les clauses conditionnelles.

```
'regardez-<t>-ici' expandMacros → 'regardez- -ici'
'<1s> est <2s>' expandMacrosWith: 'Pharo' with: 'extra' → 'Pharo est extra'
'<2s> est <1s>' expandMacrosWith: 'Pharo' with: 'extra' → 'extra est Pharo'
'<1p> ou <1s>' expandMacrosWith: 'Pharo' with: 'extra' → '"Pharo" ou Pharo'
'<1?Quentin:Thibaut> joue' expandMacrosWith: true → 'Quentin joue'
'<1?Quentin:Thibaut> joue' expandMacrosWith: false → 'Thibaut joue'
```

Des méthodes utilitaires en vrac. La classe `String` offre de nombreuses fonctionnalités incluant les messages `asLowercase` (pour mettre en minuscule), `asUppercase` (pour mettre en majuscule) et `capitalized` (pour mettre avec la première lettre en capitale).

```
'XYZ' asLowercase → 'xyz'
'xyz' asUppercase → 'XYZ'
'hilaire' capitalized → 'Hilaire'
'1.54' asNumber → 1.54
```

```
'cette phrase est sans aucun doute beaucoup trop longue' contractTo: 20 → 'cette  
phr...p longue'
```

Remarquez qu'il y a généralement une différence entre demander une représentation descriptive de l'objet en chaîne de caractères en envoyant le message `printString` et en le convertissant en une chaîne de caractères via le message `asString`. Voici un exemple de différence :

```
#ASymbol printString → '#ASymbol'  
#ASymbol asString → 'ASymbol'
```

Un symbole `Symbol` est similaire à une chaîne de caractères mais nous sommes garantis de son unicité globale. Pour cette raison, les symboles sont préférés aux `String` comme clé de dictionnaire, en particulier pour les instances de `IdentityDictionary`. Voyez aussi le chapitre 8 pour plus d'informations sur `String` et `Symbol`.

9.5 Les collections itératrices ou iterators

En Smalltalk, les boucles et les clauses conditionnelles sont simplement des messages envoyés à des collections ou d'autres objets tels que des entiers ou des blocs (voir aussi le chapitre 3). En plus des messages de bas niveau comme `to:do:` qui évalue un bloc avec un argument qui parcourt les valeurs entre un nombre initial et final, la hiérarchie de collections Smalltalk offre de nombreux itérateurs de haut niveau. Ceci vous permet de faire un code plus robuste et plus compact.

L'itération par (`do :)`

La méthode `do:` est un itérateur de collections basique. Il applique son argument (un bloc avec un simple argument) à chaque élément du receveur. L'exemple suivant imprime toutes les chaînes de caractères contenues dans le receveur vers le Transcript.

```
#"bob' 'joe' 'toto') do: [:each | Transcript show: each; cr].
```

Les variantes. Il existe de nombreuses variantes de `do:`, telles que `do:without:`, `doWithIndex:` et `reverseDo:`; pour les collections indexées (`Array`, `OrderedCollection`, `SortedCollection`), la méthode `doWithIndex:` vous donne accès aussi à l'indice courant. Cette méthode est reliée à `to:do:` qui est définie dans la classe `Number`.

```
#"bob' 'joe' 'toto') doWithIndex: [:each :i | (each = 'joe') ifTrue: [↑ i]] → 2
```

Pour des collections ordonnées, `reverseDo:` parcourt la collection dans l'ordre inverse.

Le code suivant montre un message intéressant : `do:separatedBy:` exécute un second bloc à insérer entre les éléments.

```
res := ".
#('bob' 'joe' 'toto') do: [:e | res := res, e ] separatedBy: [res := res, '.'].
res    →  'bob.joe.toto'
```

Notez que ce code n'est pas très efficace puisqu'il crée une chaîne de caractères intermédiaire ; il serait préférable d'utiliser un flux de données en écriture ou `write stream` pour stocker le résultat dans un tampon (voir le chapitre 10) :

```
String streamContents: [:stream | #('bob' 'joe' 'toto') asStringOn: stream delimiter: '.' ]
→ 'bob.joe.toto'
```

Les dictionnaires. Quand la méthode `do:` est envoyée à un dictionnaire, les éléments pris en compte sont les valeurs et non pas les associations. Les méthodes appropriées sont `keysDo:`, `valuesDo:` et `associationsDo:` pour itérer respectivement sur les clés, les valeurs ou les associations.

```
colors := Dictionary newFrom: { #yellow -> Color yellow. #blue -> Color blue. #red ->
Color red }.
colors keysDo: [:key | Transcript show: key; cr].           "affiche les clés"
colors valuesDo: [:value | Transcript show: value;cr].       "affiche les valeurs"
colors associationsDo: [:value | Transcript show: value;cr]. "affiche les associations"
```

Collecter les résultats avec `collect` :

Si vous voulez traiter les éléments d'une collection et produire une nouvelle collection en résultat, vous devez utiliser plutôt le message `collect:` ou d'autres méthodes d'itérations au lieu du message `do:..`. La plupart peuvent être trouvés dans le protocole *enumerating* de la classe `Collection` et de ses sous-classes.

Imaginez que nous voulions qu'une collection contienne le double des éléments d'une autre collection. En utilisant la méthode `do:`, nous devons écrire le code suivant :

```
double := OrderedCollection new.
#(1 2 3 4 5 6) do: [:e | double add: 2 * e].
double   →  an OrderedCollection(2 4 6 8 10 12)
```

La méthode `collect:` exécute son bloc-argument pour chaque élément et renvoie une collection contenant les résultats. En utilisant désormais `collect:`, notre code se simplifie :

```
#(1 2 3 4 5 6) collect: [:e | 2 * e] → #(2 4 6 8 10 12)
```

Les avantages de `collect:` sur `do:` sont encore plus démonstratifs sur l'exemple suivant dans lequel nous générerons une collection de valeurs absolues d'entiers contenues dans une autre collection :

```
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)
```

Comparez le code ci-dessus avec l'expression suivante beaucoup plus simple :

```
#(2 -3 4 -35 4 -11) collect: [:each | each abs] → #(2 3 4 35 4 11)
```

Le fait que cette seconde solution fonctionne aussi avec les `Set` et les `Bag` est un autre avantage.

Vous devriez généralement éviter d'utiliser `do:` à moins que vous vouliez envoyer des messages à chaque élément d'une collection.

Notez que l'envoi du message `collect:` renvoie le même type de collection que le receveur. C'est pour cette raison que le code suivant échoue. (Un `String` ne peut pas stocker des valeurs entières.)

```
'abc' collect: [:ea | ea asciiValue] "erreur!"
```

Au lieu de ça, nous devons convertir d'abord la chaîne de caractères en `Array` ou un `OrderedCollection` :

```
'abc' asArray collect: [:ea | ea asciiValue] → #(97 98 99)
```

En fait, `collect:` ne garantit pas spécifiquement de retourner exactement la même classe que celle du receveur, mais seulement une classe de la même “espèce”. Dans le cas d'`Interval`, l'espèce est en réalité un tableau `Array` ! En effet, dans ce cas, nous ne sommes pas assurés que le résultat pourra être transformé en intervalle.

```
(1 to: 5) collect: [:ea | ea * 2] → #(2 4 6 8 10)
```

Sélectionner et rejeter des éléments

`select:` renvoie les éléments du receveur qui satisfont une condition particulière :

```
(2 to: 20) select: [:each | each isPrime] → #(2 3 5 7 11 13 17 19)
```

reject: fait le contraire :

```
(2 to: 20) reject: [:each | each isPrime] → #(4 6 8 9 10 12 14 15 16 18 20)
```

Identifier un élément avec detect :

La méthode detect: renvoie le premier élément du receveur qui rend vrai le test passé en bloc-argument. isVowel retourne vrai c-à-d. true si le receveur est une voyelle non-accentuée (pour plus d'explications, voir page 62).

```
'through' detect: [:each | each isVowel] → $o
```

La méthode detect:ifNone: est une variante de la méthode detect:. Son second bloc est évalué quand il n'y a pas d'élément trouvé dans le bloc.

```
Smalltalk allClasses detect: [:each | '*cobol*' match: each asString] ifNone: [ nil ] → nil
```

Accumuler les résultats avec inject:into :

Les langages de programmation fonctionnelle offrent souvent une fonction d'ordre supérieur appelée *fold* ou *reduce* pour accumuler un résultat en appliquant un opérateur binaire de manière itérative sur tous les éléments d'une collection. Pharo propose pour ce faire la méthode Collection»inject:into:.

Le premier argument est une valeur initiale et le second est un bloc-argument à deux arguments qui est appliqué au résultat (sum) et à chaque élément (each) à chaque tour.

Une application triviale de inject:into: consiste à produire la somme de nombres stockés dans une collection. A la mémoire du mathématicien Gauss, nous pouvons écrire, en Pharo, cette expression pour sommer les 100 premiers entiers :

```
(1 to: 100) inject: 0 into: [:sum :each | sum + each] → 5050
```

Un autre exemple est le bloc suivant à un argument pour calculer la factorielle :

```
factorial := [:n | (1 to: n) inject: 1 into: [:product :each | product * each]].  
factorial value: 10 → 3628800
```

D'autres messages

count: le message count: (pour compter) renvoie le nombre d'éléments satisfaisant le bloc-argument :

```
Smalltalk allClasses count: [:each | '*Collection*' match: each asString ] → 3
```

includes: le message includes: vérifie si l'argument est contenu dans la collection.

```
colors := {Color white . Color yellow. Color red . Color blue . Color orange}.
colors includes: Color blue. → true
```

anySatisfy: le message anySatisfy: renvoie vrai si au moins un élément satisfait à une condition.

```
colors anySatisfy: [:c | c red > 0.5] → true
```

9.6 Astuces pour tirer profit des collections

Une erreur courante avec add: l'erreur suivante est une des erreurs les plus fréquentes en Smalltalk.

```
collection := OrderedCollection new add: 1; add: 2.
collection → 2
```

Ici la variable collection ne contient pas la collection nouvellement créée mais par le dernier nombre ajouté. En effet, la méthode add: renvoie l'élément ajouté et non le receveur.

Le code suivant donne le résultat attendu :

```
collection := OrderedCollection new.
collection add: 1; add: 2.
collection → an OrderedCollection(1 2)
```

Vous pouvez aussi utiliser le message yourself pour renvoyer le receveur d'une cascade de messages :

```
collection := OrderedCollection new add: 1; add: 2; yourself → an
OrderedCollection(1 2)
```

Enlever un élément d'une collection en cours d'itération. Une autre erreur que vous pouvez faire est d'effacer un élément d'une collection que vous êtes en train de parcourir de manière itérative en utilisant `remove:`.

```
range := (2 to: 20) asOrderedCollection.
range do: [:aNumber | aNumber isPrime ifFalse: [ range remove: aNumber ]].
range → an OrderedCollection(2 3 5 7 9 11 13 15 17 19)
```

Ce résultat est clairement incorrect puisque 9 et 15 auraient du être filtrés !

La solution consiste à copier la collection avant de la parcourir.

```
range := (2 to: 20) asOrderedCollection.
range copy do: [:aNumber | aNumber isPrime ifFalse: [ range remove: aNumber ]].
range → an OrderedCollection(2 3 5 7 11 13 17 19)
```

Redéfinir à la fois = et hash. Une erreur difficile à identifier se produit lorsque vous redéfinissez `=` mais pas `hash`. Les symptômes sont la perte d'éléments que vous mettez dans des ensembles ainsi que d'autres phénomènes plus étranges. Une solution proposée par Kent Beck est d'utiliser `xor:` pour redéfinir `hash`. Supposons que nous voulions que deux livres soient considérés comme égaux si leurs titres et leurs auteurs sont les mêmes. Alors nous redéfinissons non seulement `=` mais aussi `hash` comme suit :

Méthode 9.1 – Redéfinir `=` et `hash`.

```
Book»= aBook
self class = aBook class ifFalse: [↑ false].
↑ title = aBook title and: [ authors = aBook authors]
```

```
Book»hash
↑ title hash xor: authors hash
```

Un autre problème ennuyeux peut surgir lorsque vous utilisez des objets modifiables ou *mutables* : ils peuvent changer leur code de hachage constamment quand ils sont éléments d'un Set ou clés d'un dictionnaire. Ne le faites donc pas à moins que vous aimiez vraiment le débogage !

9.7 Résumé du chapitre

La hiérarchie des collections en Smalltalk offre un vocabulaire commun pour la manipulation uniforme d'une grande famille de collections.

- Une distinction essentielle est faite entre les collections séquentielles ou `SequenceableCollections` qui stockent leurs éléments dans un ordre donné, les dictionnaires de classe `Dictionary` ou de ses sous-classes qui

enregistrent des associations clé–valeur et les ensembles (Set) ou multi-ensembles (Bag) qui sont eux désordonnés.

- Vous pouvez convertir la plupart des collections en d'autres sortes de collections en leur envoyant des messages tels que `asArray`, `asOrderedCollection` etc.
- Pour trier une collection, envoyez-lui le message `asSortedCollection`.
- Les tableaux littéraux ou *literal Array* sont créés grâce à une syntaxe spéciale : `#(...)`. Les tableaux dynamiques sont créés avec la syntaxe `{ ... }`.
- Un dictionnaire `Dictionary` compare ses clés par égalité. C'est plus utile lorsque les clés sont des instances de `String`. Un `IdentityDictionary` utilise l'identité entre objets pour comparer les clés. Il est souhaitable que des `Symbols` soient utilisés comme clés ou que la correspondance soit établie entre les références d'objets et les valeurs.
- Les chaînes de caractères de classe `String` comprennent aussi les messages habituels de la collection. En plus, un `String` supporte une forme simple d'appariement de formes ou *pattern-matching*. Pour des applications plus avancées, vous aurez besoin du paquetage d'expressions régulières Regex.
- Le message de base pour l'itération est `do:`. Il est utile pour du code impératif tel que la modification de chaque élément d'une collection ou l'envoi d'un message sur chaque élément.
- Au lieu d'utiliser `do:`, il est d'usage d'employer `collect:`, `select:`, `reject:`, `includes:`, `inject:into:` et d'autres messages de haut niveau pour un traitement uniforme des collections.
- Ne jamais effacer un élément d'une collection que vous parcourrez itérativement. Si vous devez la modifier, itérez plutôt sur une copie.
- Si vous surchargez `=`, souvenez-vous d'en faire de même pour le message `hash` qui renvoie le code de hachage !

Chapitre 10

Streams : les flux de données

Les flux de données ou *streams* sont utilisés pour itérer dans une séquence d'éléments comme des collections, des fichiers ou des flux réseau. Les *streams* peuvent être en lecture ou en écriture ou les deux. La lecture et l'écriture est toujours relative à la position actuelle dans le *stream*. Les *streams* peuvent être facilement convertis en collections (enfin presque toujours) et les collections en *streams*.

10.1 Deux séquences d'éléments

Voici une bonne métaphore pour comprendre ce qu'est un flux de données : un flux de données ou *stream* peut être représenté comme deux séquences d'éléments : une séquence d'éléments passée et une séquence d'éléments future. Le *stream* est positionné entre les deux séquences. Comprendre ce modèle est important car toutes les opérations sur les *streams* en Smalltalk en dépendent. C'est pour cette raison que la plupart des classes Stream sont des sous-classes de PositionableStream. La figure 10.1 présente un flux de données contenant cinq caractères. Ce *stream* est dans sa position originale *c-à-d.* qu'il n'y a aucun élément dans le passé. Vous pouvez revenir à cette position en envoyant le message *reset*.

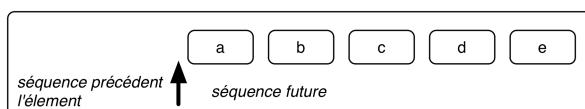


FIGURE 10.1 – Un flux de données positionné à son origine.

Lire un élément revient conceptuellement à effacer le premier élément de la séquence d'éléments future et le mettre après le dernier élément dans la séquence d'éléments passée. Après avoir lu un élément avec le message `next`, l'état de votre *stream* est celui de la figure 10.2.

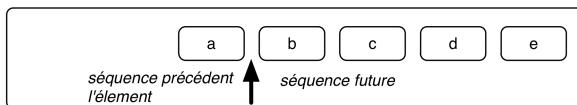


FIGURE 10.2 – Le même flux de données après l'exécution de la méthode `next` : le caractère a est “dans le passé” alors que b, c, d and e sont “dans le futur”.

Écrire un élément revient à remplacer le premier élément de la séquence future par le nouveau et le déplacer dans le passé. La figure 10.3 montre l'état du même *stream* après avoir écrit un x via le message `nextPut: anElement`.

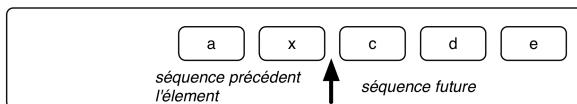


FIGURE 10.3 – Le même flux de données après avoir écrit un x.

10.2 Streams contre Collections

Le protocole des collections supporte le stockage, l'effacement et l'énumération des éléments d'une collection mais il ne permet pas que ces opérations soient combinées ensemble. Par exemple, si les éléments d'une `OrderedCollection` sont traités par une méthode `do:`, il n'est pas possible d'ajouter ou d'enlever des éléments à l'intérieur du bloc `do:`. Ce protocole ne permet pas non plus d'itérer dans deux collections en même temps en choisissant quelle collection on itère, laquelle on n'itère pas. De telles procédures requièrent qu'un index de parcours ou une référence de position soit maintenu hors de la collection elle-même : c'est exactement le rôle de `ReadStream` (pour la lecture), `WriteStream` (pour l'écriture) et `ReadWriteStream` (pour les deux).

Ces trois classes sont définies pour *glisser à travers*¹ une collection. Par exemple, le code suivant crée un *stream* sur un intervalle puis y lit deux éléments.

1. En anglais, nous dirions “stream over”.

```
r := ReadStream on: (1 to: 1000).
r next.    → 1
r next.    → 2
r atEnd.   → false
```

Les WriteStreams peuvent écrire des données dans la collection :

```
w := WriteStream on: (String new: 5).
w nextPut: $a.
w nextPut: $b.
w contents. → 'ab'
```

Il est aussi possible de créer des ReadWriteStreams qui supportent les protocoles de lecture et d'écriture.

Le principal problème de WriteStream et de ReadWriteStream est que, dans Pharo, ils ne supportent que les tableaux et les chaînes de caractères. Cette limitation est en cours de disparition grâce au développement d'une nouvelle librairie nommée *Nile*². mais en attendant, vous obtiendrez une erreur si vous essayez d'utiliser les *streams* avec un autre type de collection :

```
w := WriteStream on: (OrderedCollection new: 20).
w nextPut: 12. → lève une erreur
```

Les *streams* ne sont pas seulement destinés aux collections mais aussi aux fichiers et aux *sockets*. L'exemple suivant crée un fichier appelé `test.txt`, y écrit deux chaînes de caractères, séparées par un retour-chariot et enfin ferme le fichier.

```
StandardFileStream
fileNamed: 'test.txt'
do: [:str | str
      nextPutAll: '123';
      cr;
      nextPutAll: 'abcd'].
```

Les sections suivantes s'attardent sur les protocoles.

10.3 Utiliser les streams avec les collections

Les *streams* sont vraiment utiles pour traiter des collections d'éléments. Ils peuvent être utilisés pour la lecture et l'écriture d'éléments dans des collections. Nous allons explorer maintenant les caractéristiques des *streams* dans le cadre des collections.

2. Disponible à www.squeaksource.com/Nile.html

Lire les collections

Cette section présente les propriétés utilisées pour lire des collections. Utiliser les flux de données pour lire une collection repose essentiellement sur le fait de disposer d'un pointeur sur le contenu de la collection. Vous pouvez placer où vous voulez ce pointeur qui avancera dans le contenu pour lire. La classe ReadStream devrait être utilisée pour lire les éléments dans les collections.

Les méthodes next et next: sont utilisées pour récupérer un ou plusieurs éléments dans la collection.

```
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'
```

Le message peek est utilisé quand vous voulez connaître l'élément suivant dans le *stream* sans avancer dans le flux.

```
stream := ReadStream on: '-143'.
negative := (stream peek = $-). "regardez le premier élément sans le lire"
negative. →      true
negative ifTrue: [stream next]. "ignore le caractère moins"
number := stream upToEnd.
number. →      '143'
```

Ce code affecte la variable booléenne negative en fonction du signe du nombre dans le *stream* et number est assigné à sa valeur absolue. La méthode upToEnd (qui en français se traduirait par “jusqu'à la fin”) renvoie tout depuis la position courante jusqu'à la fin du flux de données et positionne ce dernier à sa fin. Ce code peut être simplifié grâce à peekFor: qui déplace le pointeur si et seulement si l'élément est égal au paramètre passé en argument.

```
stream := '-143' readStream.
(stream peekFor: $-) →      true
stream upToEnd →      '143'
```

peekFor: retourne aussi un booléen indiquant si le paramètre est égal à l'élément courant.

Vous avez dû remarquer une nouvelle façon de construire un *stream* dans l'exemple précédent : vous pouvez simplement envoyer readStream à

une collection séquentielle pour avoir un flux de données en lecture seule sur une collection.

Positionner. Il existe des méthodes pour positionner le pointeur du *stream*. Si vous connaissez l'emplacement, vous pouvez vous y rendre directement en utilisant `position:`. Vous pouvez demander la position actuelle avec `position`. Souvenez-vous bien qu'un *stream* n'est pas positionné sur un élément, mais entre deux éléments. L'index 0 correspond au début du flux.

Vous pouvez obtenir l'état du *stream* montré dans la figure 10.4 avec le code suivant :

```
stream := 'abcde' readStream.
stream position: 2.
stream peek  →  $c
```

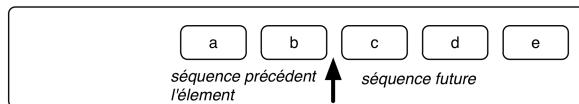


FIGURE 10.4 – Un flux de données à la position 2.

Si vous voulez aller au début ou à la fin, vous pouvez utiliser `reset` ou `setToEnd`. Les messages `skip:` et `skipTo:` sont utilisés pour avancer d'une position relative à la position actuelle : la méthode `skip:` accepte un nombre comme argument et saute sur une distance de ce nombre d'éléments alors que `skipTo:` saute tous les éléments dans le flux jusqu'à trouver un élément égal à son argument. Notez que cette méthode positionne le *stream* après l'élément identifié.

```
stream := 'abcdef' readStream.
stream next.      →  $a  "le flux est à la position juste après a"
stream skip: 3.   →  nil "le flux est après d"
stream position. →  4
stream skip: -2.  →  nil "le flux est après b"
stream position. →  2
stream reset.
stream position. →  0
stream skipTo: $e. →  nil "le flux est après e"
stream next.      →  $f
stream contents. →  'abcdef'
```

Comme vous pouvez le voir, la lettre `e` a été sautée.

La méthode `contents` retourne toujours une copie de l'intégralité du flux de données.

Tester. Certaines méthodes vous permettent de tester l'état d'un *stream* courant : la méthode `atEnd` renvoie `true` si et seulement si aucun élément ne peut être trouvé après la position actuelle alors que `isEmpty` renvoie `true` si et seulement si aucun élément ne se trouve dans la collection.

Voici une implémentation possible d'un algorithme utilisant `atEnd` et prenant deux collections triées comme paramètres puis les fusionnant dans une autre collection triée :

```
stream1 := #(1 4 9 11 12 13) readStream.  
stream2 := #(1 2 3 4 5 10 13 14 15) readStream.
```

"La variable résultante contiendra la collection triée."

```
result := OrderedCollection new.  
[stream1 atEnd not & stream2 atEnd not]  
whileTrue: [stream1 peek < stream2 peek  
  "Enlève le plus petit élément de chaque flux et l'ajoute au résultat"  
  ifTrue: [result add: stream1 next]  
  ifFalse: [result add: stream2 next]].
```

"Un des deux flux peut ne pas être à la position finale. Copie ce qu'il reste"

```
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)
```

Écrire dans les collections

Nous avons déjà vu comment lire une collection en itérant sur ses éléments via un objet `ReadStream`. Apprenons maintenant à créer des collections avec la classe `WriteStream`.

Les flux de données `WriteStream` sont utiles pour adjoindre des données en plusieurs endroits dans une collection. Ils sont souvent utilisés pour construire des chaînes de caractères basées sur des parties à la fois statiques et dynamiques comme dans l'exemple suivant :

```
stream := String new writeStream.  
stream  
nextPutAll: 'Cette image Smalltalk contient: ';  
print: Smalltalk allClasses size;  
nextPutAll: ' classes.';  
cr;  
nextPutAll: 'C'est vraiment beaucoup.'.  
  
stream contents. → 'Cette image Smalltalk contient: 2322 classes. C'est vraiment  
beaucoup.'
```

Par exemple, cette technique est utilisée dans différentes implémentations de la méthode `printOn:`. Il existe une manière plus simple et plus efficace de créer des flux de données si vous êtes seulement intéressé au contenu du `stream` :

```
string := String streamContents:
[:stream | 
  stream
    print: #(1 2 3);
    space;
    nextPutAll: 'size';
    space;
    nextPut: $=;
    space;
    print: 3.].
string. → '#(1 2 3) size = 3'
```

La méthode `streamContents:` crée une collection et un `stream` sur cette collection. Elle exécute ensuite le bloc que vous lui donnez en passant le `stream` comme argument de bloc. Quand le bloc se termine, `streamContents:` renvoie le contenu de la collection.

Les méthodes de `WriteStream` suivantes sont spécialement utiles dans ce contexte :

nextPut : ajoute le paramètre au flux de données ;

nextPutAll : ajoute chaque élément de la collection passé en argument au flux ;

print : ajoute la représentation textuelle du paramètre au flux.

Il existe aussi des méthodes utiles pour imprimer différentes sortes de caractères au `stream` comme `space` (pour un espace), `tab` (pour une tabulation) et `cr` (pour *Carriage Return c-à-d. le retour-chariot*). Une autre méthode s'avère utile pour s'assurer que le dernier caractère dans le flux de données est un espace : il s'agit de `ensureASpace` ; si le dernier caractère n'est pas un espace, il en ajoute un.

Au sujet de la concaténation. L'emploi de `nextPut:` et de `nextPutAll:` sur un `WriteStream` est souvent le meilleur moyen pour concaténer les caractères. L'utilisation de l'opérateur virgule (,) est beaucoup moins efficace :

```
[] temp |
temp := String new.
(1 to: 100000)
do: [:i | temp := temp, i asString, '']] timeToRun → 115176 "(ms)"

[] temp |
```

```
temp := WriteStream on: String new.
(1 to: 100000)
do: [:i | temp nextPutAll: i asString; space].
temp contents] timeToRun → 1262 "(milliseconds)"
```

La raison pour laquelle l'usage d'un *stream* est plus efficace provient du fait que l'opérateur virgule crée une nouvelle chaîne de caractères contenant la concaténation du receveur et de l'argument, donc il doit les copier tous les deux. Quand vous concaténez de manière répétée sur le même receveur, ça prend de plus en plus de temps à chaque fois ; le nombre de caractères copiés s'accroît de façon exponentielle. Cet opérateur implique aussi une surcharge de travail pour le ramasse-miettes qui collecte ces chaînes. Pour ce cas, utiliser un *stream* plutôt qu'une concaténation de chaînes est une optimisation bien connue. En fait, vous pouvez utiliser la méthode de classe `streamContents:` (mentionnée à la page 229) pour parvenir à ceci :

```
String streamContents: [:tempStream |
(1 to: 100000)
do: [:i | tempStream nextPutAll: i asString; space]]
```

Lire et écrire en même temps

Vous pouvez utiliser un flux de données pour accéder à une collection en lecture et en écriture en même temps. Imaginez que vous voulez créer une classe d'historique que nous appelerons `History` et qui gérera les boutons "Retour" (*Back*) et "Avant" (*Forward*) d'un navigateur web. Un historique réagirait comme le montrent les illustrations depuis 10.5 jusqu'à 10.11.



FIGURE 10.5 – Un nouvel historique est vide. Rien n'est affiché dans le navigateur web.

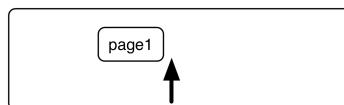


FIGURE 10.6 – L'utilisateur ouvre la page 1.

Ce comportement peut être programmé avec un `ReadWriteStream`.

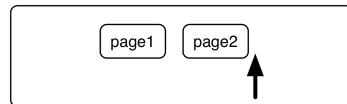


FIGURE 10.7 – L'utilisateur clique sur un lien vers la page 2.

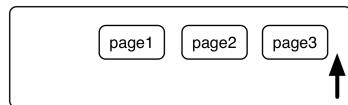


FIGURE 10.8 – L'utilisateur clique sur un lien vers la page 3.

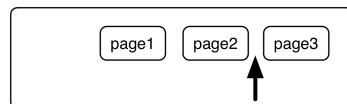


FIGURE 10.9 – L'utilisateur clique sur le bouton “Retour” (Back). Il visite désormais la page 2 à nouveau.

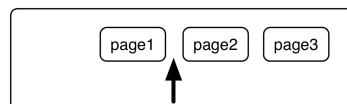


FIGURE 10.10 – L'utilisateur clique sur le bouton “Retour” (Back). La page 1 est affichée maintenant.

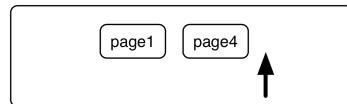


FIGURE 10.11 – Depuis la page 1, l'utilisateur clique sur un lien vers la page 4. L'historique oublie les pages 2 et 3.

```
Object subclass: #History  
instanceVariableNames: 'stream'  
classVariableNames: ''
```

```
poolDictionaries: "
category: 'PBE–Streams'
```

```
History>>initialize
super initialize.
stream := ReadWriteStream on: Array new.
```

Nous n'avons rien de compliqué ici ; nous définissons une nouvelle classe qui contient un *stream*. Ce *stream* est créé dans la méthode *initialize* depuis un tableau.

Nous avons besoin d'ajouter les méthodes *goBackward* et *goForward* pour aller respectivement en arrière (“Retour”) et en avant :

```
History>>goBackward
self canGoBackward ifFalse: [self error: 'Déjà sur le premier élément'].
stream skip: -2.
↑ stream next
```

```
History>>goForward
self canGoForward ifFalse: [self error: 'Déjà sur le dernier élément'].
↑ stream next
```

Jusqu'ici le code est assez simple. Maintenant, nous devons nous occuper de la méthode *goTo*: (que nous pouvons traduire en français par “aller à”) qui devrait être activée quand l'utilisateur clique sur un lien. Une solution possible est la suivante :

```
History>>goTo: aPage
stream nextPut: aPage.
```

Cette version est cependant incomplète. Ceci vient du fait que lorsque l'utilisateur clique sur un lien, il ne devrait plus y avoir de pages futurs c-à-d. que le bouton “Avant” devrait être désactivé. Pour ce faire, la solution la plus simple est d'écrire *nil* juste après la position courante pour indiquer la fin de l'historique :

```
History>>goTo: anObject
stream nextPut: anObject.
stream nextPut: nil.
stream back.
```

Maintenant, seules les méthodes *canGoBackward* (pour dire si oui ou non nous pouvons aller en arrière) et *canGoForward* (pour dire si oui ou non nous pouvons aller en avant) sont à coder.

Un flux de données est toujours positionné entre deux éléments. Pour aller en arrière, il doit y avoir deux pages avant la position courante : une est la page actuelle et l'autre est la page que nous voulons atteindre.

```
History>>canGoBackward  
↑ stream position > 1  
  
History>>canGoForward  
↑ stream atEnd not and: [stream peek notNil]
```

Ajoutons pour finir une méthode pour accéder au contenu du *stream* :

```
History>>contents  
↑ stream contents
```

Faisons fonctionner maintenant notre historique comme dans la séquence illustrée plus haut :

```
History new  
goTo: #page1;  
goTo: #page2;  
goTo: #page3;  
goBackward;  
goBackward;  
goTo: #page4;  
contents → #(#page1 #page4 nil nil)
```

10.4 Utiliser les streams pour accéder aux fichiers

Vous avez déjà vu comment glisser sur une collection d'éléments via un *stream*. Il est aussi possible d'en faire de même avec un flux sur des fichiers de votre disque dur. Une fois créé, un *stream* sur un fichier est comme un *stream* sur une collection : vous pourrez utiliser le même protocole pour lire, écrire ou positionner le flux. La principale différence apparaît à la création du flux de données. Nous allons voir qu'il existe plusieurs manières de créer un *stream* sur un fichier.

Créer un flux pour fichier

Créer un *stream* sur un fichier consiste à utiliser une des méthodes de création d'instance suivantes mises à disposition par la classe `FileStream` :

fileNamed : ouvre en lecture et en écriture un fichier avec le nom donné. Si le fichier existe déjà, son contenu pourra être modifié ou remplacé mais le fichier ne sera pas tronqué à la fermeture. Si le nom n'a pas de chemin spécifié pour répertoire, le fichier sera créé dans le répertoire par défaut.

newFileNamed : crée un nouveau fichier avec le nom donné et retourne un *stream* ouvert en écriture pour ce fichier. Si le fichier existe déjà, il est demandé à l'utilisateur de choisir la marche à suivre.

forceNewFileNamed : crée un nouveau fichier avec le nom donné et répond un *stream* ouvert en écriture sur ce fichier. Si le fichier existe déjà, il sera effacé avant qu'un nouveau ne soit créé.

oldFileNamed : ouvre en lecture et en écriture un fichier existant avec le nom donné. Si le fichier existe déjà, son contenu pourra être modifié ou remplacé mais le fichier ne sera pas tronqué à la fermeture. Si le nom n'a pas de chemin spécifié pour répertoire, le fichier sera créé dans le répertoire par défaut.

readOnlyFileNamed : ouvre en lecture seule un fichier existant avec le nom donné.

Vous devez vous remémorer de fermer le *stream* sur le fichier que vous avez ouvert. Ceci se fait grâce à la méthode *close*.

```
stream := FileStream forceNewFileNamed: 'test.txt'.
stream
nextPutAll: 'Ce texte est écrit dans un fichier nommé ';
print: stream localName.
stream close.

stream := FileStream readOnlyFileNamed: 'test.txt'.
stream contents.    →  'Ce fichier est écrit dans un fichier nommé "test.txt"'
stream close.
```

La méthode *localName* retourne le dernier composant du nom du fichier. Vous pouvez accéder au chemin entier en utilisant la méthode *fullName*.

Vous remarquerez bientôt que la fermeture manuelle de *stream* de fichier est pénible et source d'erreurs. C'est pourquoi *FileStream* offre un message appelé *forceNewFileNamed:do:* pour fermer automatiquement un nouveau flux de données après avoir évalué un bloc qui modifie son contenu.

```
FileStream
forceNewFileNamed: 'test.txt'
do: [:stream |
  stream
  nextPutAll: 'Ce texte est écrit dans un fichier nommé ';
  print: stream localName].
string := FileStream
  readOnlyFileNamed: 'test.txt'
  do: [:stream | stream contents].
string   →  'Ce fichier est écrit dans un fichier nommé "test.txt"'
```

Les méthodes de création de flux de données prenant un bloc comme argument créent d'abord un *stream* sur un fichier, puis exécute un argument

et enfin ferme le *stream*. Ces méthodes retournent ce qui est retourné par le bloc, *c-à-d.* la valeur de la dernière expression dans le bloc. C'est ce que nous avons utilisé dans l'exemple précédent pour récupérer le contenu d'un fichier et le mettre dans la variable *string*.

Les flux binaires

Par défaut, les *streams* créés sont à base textuelle ce qui signifie que vous lirez et écrirez des caractères. Si votre flux doit être binaire, vous devez lui envoyer le message *binary*.

Quand votre *stream* est en mode binaire, vous pouvez seulement écrire des nombres de 0 à 255 (ce qui correspond à un octet). Si vous voulez utiliser *nextPutAll*: pour écrire plus d'un nombre à la fois, vous devez passer comme argument un tableau d'octets de la classe *ByteArray*.

```
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]  
].
```

Voici un autre exemple créant une image dans un fichier nommé "test.pgm". Vous pouvez ouvrir ce fichier avec votre programme de dessin préféré.

```
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
```

]

Cela crée un échiquier 4 par 4 comme nous montre la figure 10.12.

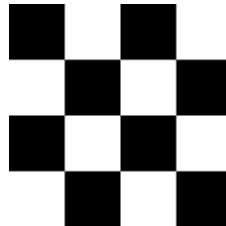


FIGURE 10.12 – Un échiquier 4 par 4 que vous pouvez dessiner en utilisant des *streams* binaires.

10.5 Résumé du chapitre

Par rapport aux collections, les flux de données ou *streams* offrent un bien meilleur moyen de lire et d'écrire de manière incrémentale dans une séquence d'éléments. Il est très facile de passer par conversion de *streams* à collections et vice-versa.

- Les flux peuvent être soit en lecture, soit en écriture, soit à la fois en lecture-écriture.
- Pour convertir une collection en un *stream*, définissez un *stream* sur une collection grâce au message `on:`, *par ex.*, `ReadStream on: (1 to: 1000)`, ou via les messages `readStream`, etc sur la collection.
- Pour convertir un *stream* en collection, envoyer le message `contents`.
- Pour concaténer des grandes collections, il est plus efficace d'abandonner l'emploi de l'opérateur virgule , et de créer un *stream* et y adjoindre les collections avec le message `nextPutAll:` puis extraire enfin le résultat en lui envoyant `contents`.
- Par défaut, les *streams* de fichiers sont à base de caractères. Envoyer le message `binary` en fait explicitement des *streams* binaires.

Chapitre 11

L'interface Morphic

Morphic est le nom de l'interface graphique de Pharo. Elle est écrite en Smalltalk, donc elle est pleinement portable entre différents systèmes d'exploitation ; en conséquence de quoi, Pharo a le même aspect sur Unix, Mac OS X et Windows. L'absence de distingo entre *composition* et *exécution* de l'interface est la principale divergence de Morphic avec la plupart des autres boîtes à outils graphiques : tous ses éléments graphiques peuvent être assemblés et désassemblés à tout moment par l'utilisateur.

Morphic a été développée par John Maloney et Randy Smith pour le langage de programmation orienté objet Self développé chez Sun Microsystems : l'interface de ce langage basé sur le concept de prototypes (comme JavaScript) est apparue en 1993. Maloney réécrivit ensuite une nouvelle version de Morphic pour Pharo tout en conservant de la version originale son aspect *direct* et *vivant*. Dans ce chapitre, nous ferons une immersion dans cet univers d'objets graphiques, les *morphs* et nous apprendrons à les modeler (à la souris ou en programmation), à leur ajouter des fonctionnalités (pour accroître leur capacité d'interaction) et enfin, en préambule d'un exemple complet, nous verrons comment ils s'intègrent non seulement dans l'espace mais aussi dans le temps.

11.1 Première immersion dans Morphic

Réponse au doigt et à l'œil

Le caractère direct de l'interface Morphic se traduit par le fait que toutes les formes graphiques sont des objets inspectables et modifiables directement par la souris.

De plus, le fait que toute action faite par l'utilisateur donne lieu à une

réponse de la part de Morphic définit son caractère vivant : les informations affichées sont constamment mise à jour au fur et à mesure des changements du "monde" que l'interface décrit. Comme preuve de cette vie et de toute la dynamique qui en résulte, nous vous proposons d'isoler une option du menu World et de vous en faire un bouton hors du menu.

 Afficher le menu World. Meta-cliquez une première fois sur le menu World de manière à afficher son halo¹ Morphic. Meta-cliquez à nouveau sur l'option de menu que vous voulez détacher, disons Workspace pour afficher son halo. Déplacez celui-ci n'importe où sur l'écran en glissant la poignée noire , comme le montre la figure 11.1.

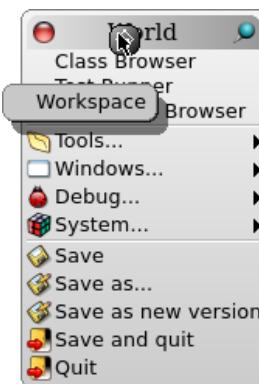


FIGURE 11.1 – Détacher l'option de menu `Workspace` pour en faire un bouton indépendant.

Un monde de morphs

Tous les objets que vous voyez à l'écran dans Pharo sont des *morphs* ; tous sont des instances des sous-classes de Morph. Morph est une grande classe avec de nombreuses méthodes qui permettent d'implémenter des sous-classes ayant un comportement original avec très peu de code. Vous pouvez créer un morph pour représenter n'importe quel objet.

 Pour créer un morph représentant une chaîne de caractères, évaluer le code suivant dans un espace de travail.

```
'Morph' asMorph openInWorld
```

1. Rappelez-vous que vous devez avoir l'option `halosEnabled` activée dans le Preference Browser. Vous pouvez aussi l'activer en évaluant `Preferences enable: #halosEnabled` dans un espace de travail.

Ce code crée un morph pour représenter la chaîne de caractères 'Morph' et l'affiche dans l'écran principal, le "world" (en français, nous dirions "monde" puisque la fenêtre Pharo est un *monde de morphs*). Vous pouvez manipuler cet objet graphique en meta-cliquant.

Personnaliser sa représentation

Revenons maintenant au code qui a créé ce morph. Tout repose sur la méthode qui fabrique un morph à partir d'une chaîne de caractères : cette méthode `asMorph` implémentée dans `String` crée un `StringMorph`. `asMorph` est implementée par défaut dans `Object` donc tout objet peut être représenté par un morph. En réalité, la méthode `asMorph` dans `Object` fait appel à sa méthode dérivée dans `String`. Ainsi, tant qu'une classe n'a pas surchargé cette méthode, elle sera représentée par un `StringMorph`. Par exemple, évaluer `Color orange asMorph openInWorld` ouvrira un `StringMorph` dont le label sera le résultat de `Color orange printString` (comme en faisant un CMD-p sur `Color orange` dans un `Workspace`). Voyons comment obtenir un rectangle de couleur plutôt que ce `StringMorph`.

 Ouvrez un navigateur de classes sur la classe `Color` et ajoutez la méthode suivante dans le protocole `creation` :

Méthode 11.1 – Obtenir un morph d'une instance de `Color`

```
Color»asMorph
    ↑ Morph new color: self
```

Exécutez `Color blue asMorph openInWorld` dans un espace de travail. Fini le texte d'affichage `printString` ! Vous obtenez un joli rectangle bleu.

11.2 Manipuler les morphs

Puisque les morphs sont des objets, nous pouvons les manipuler comme n'importe quel autre objet dans Smalltalk *c-à-d.* par envoi de messages. Dès lors nous pouvons entre autre changer leurs propriétés ou créer de nouvelles sous-classes de `Morph`.

Qu'il soit affiché à l'écran ou non, tout morph a une position et une taille. Tous les morphs sont inclus, par commodité, dans une boîte englobante, *c-à-d.* une région rectangulaire occupant un certain espace de l'écran. Dans le cas des formes irrégulières, leur position et leur taille correspondent à celles du plus petit rectangle qui englobe la forme. Cette boîte englobante définit les limites (ou *bounds*) du morph. La méthode `position` retourne un `Point` qui décrit la position du coin supérieur gauche du morph (*c-à-d.* le coin supérieur gauche de sa boîte englobante). L'origine des coordonnées du système

est le coin supérieur gauche de l'écran : la valeur de la coordonnée *y* augmente en descendant l'écran et la valeur de *x* augmente en allant de gauche à droite. La méthode `extent` renvoie aussi un point, mais ce point définit la largeur et la hauteur du morph plutôt qu'une position.

➊ *Entrez le code suivant dans un espace de travail et évaluez-le (do it) :*

```
joe := Morph new color: Color blue.
joe openInWorld.
bill := Morph new color: Color red.
bill openInWorld.
```

Ce code affiche deux nouveaux morphs répondant aux noms de *joe* et *bill* : par défaut, un morph apparaît comme un rectangle de position (0@0) et de taille (50@40). Saisissez ensuite *joe position* et affichez son résultat par `print it`. Pour déplacer *joe*, exécutez *joe position: (joe position + (10@3))* plusieurs fois. Vous pouvez modifier la taille aussi. Pour avoir la taille de *joe*, vous pouvez évaluer par `print it` l'expression *joe extent*. Pour le faire grandir, exécutez *joe extent: (joe extent * 1.1)*. Pour changer la couleur d'un morph, envoyez-lui le message `color:` avec en argument un objet de classe `Color`, correspondant à la couleur désirée. Par exemple, *joe color: Color orange*. Pour ajouter la transparence, essayez *joe color: (Color blue alpha: 0.5)*.

➋ *Pour faire en sorte que *bill* suive *joe*, vous pouvez exécuter ce code de manière répétée :*

```
bill position: (joe position + (100@0))
```

Si vous déplacez *joe* avec la souris et que vous exécutez ce code, *bill* se déplacera pour se positionner à 100 pixels à droite de *joe*.

11.3 Composer des morphs

Créer de nouvelles représentations graphiques peut se faire en plaçant un morph à l'intérieur d'un autre. C'est ce que nous appelons la *composition* ; les morphs peuvent être composés à l'infini. Pour ce faire, vous pouvez envoyer au morph contenant le message `addMorph:`

➌ *Ajoutez un morph à un autre avec le code suivant :*

```
star := StarMorph new color: Color yellow.
joe addMorph: star.
star position: joe position.
```

La dernière ligne place l'étoile nommée star aux mêmes coordonnées que joe. Notez que les coordonnées du morph contenu sont toujours à la position absolue définie par rapport à l'écran, et non à la position relative définie par rapport au morph contenant. Plusieurs méthodes sont disponibles pour positionner un morph ; naviguez dans les méthodes du protocole *geometry* de la classe Morph pour le constater vous-même. Par exemple, centrer l'étoile dans joe revient à exécuter Star center: joe center.



FIGURE 11.2 – L'étoile de classe StarMorph est contenue dans joe, le morph bleu translucide.

Si vous attrapez l'étoile avec la souris, vous constaterez que vous prenez en réalité joe et que les deux morphs sont ensemble : l'étoile est *inclus* à l'intérieur de joe. Il est possible d'inclure plus de morphs dans joe. Les morphs inclus sont appelés des sous-morphs (en anglais, *submorphs*). Comme l'interface Morphic propose une interactivité directe pour tout morph, nous pouvons aussi faire notre inclusion de morphs en remplaçant la programmation par une simple manipulation à la souris.

11.4 Dessiner ses propres morphs

Bien qu'il soit possible de faire des représentations graphiques utiles et intéressantes par composition de morphs, vous aurez parfois besoin de créer quelque chose de complètement différent. Pour ce faire, vous définissez une sous-classe de Morph et surchargez la méthode drawOn: pour personnaliser son apparence.

L'interface Morphic envoie un message drawOn: à un morph à chaque fois qu'il est nécessaire de rafraîchir l'affichage du morph à l'écran. Le paramètre passé à drawOn: est un type de canevas de classe Canvas ; le morph s'affichera alors lui-même sur ce canevas dans ses limites. Utilisons cette connaissance pour créer un morph en forme de croix.

 Définissez via le Browser une nouvelle classe CrossMorph héritée de Morph :

Classe 11.2 – Définir la classe CrossMorph

```
Morph subclass: #CrossMorph
instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: ""
category: 'PBE–Morphic'
```

Nous pouvons définir la méthode drawOn: ainsi :

Méthode 11.3 – Dessiner un CrossMorph

```
drawOn: aCanvas
"crossHeight est la hauteur de la barre horizontale horizontalBar
et crossWidth est la largeur de la barre verticale verticalBar"
| 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 – Un nouveau morph en forme de croix de classe CrossMorph avec son halo. Vous pouvez redimensionner cette croix grâce à la poignée inférieure droite de couleur jaune.

Envoyer le message bounds à un morph renvoie sa boîte englobante, instance de la classe Rectangle. Les rectangles comprennent plusieurs messages qui créent d'autres rectangles de même géométrie ; dans notre méthode, nous utilisons le message insetBy: avec un point comme argument pour créer une première fois un rectangle de hauteur (en anglais, *height*) réduite, puis pour créer un autre rectangle de largeur (en anglais, *width*) réduite.

 Pour tester votre nouveau morph, évaluer l'expression CrossMorph new openInWorld.

Le résultat devrait être semblable à celui de la figure 11.3. Cependant, vous remarquerez que toute la boîte englobante est sensible à la souris (vous pouvez cliquer en dehors de la croix et interagir ou déplacer celle-ci). Corrigeons ceci en rendant la seule surface de la croix sensible à la souris.

Lorsque la librairie Morphic a besoin de trouver quels morphs se trouvent sous le curseur, elle envoie le message `containsPoint:` à tous les morphs qui ont leur boîte englobante sous le pointeur de la souris. Cette méthode répond vrai lorsque le point-argument est contenu dans la forme définie. Pour limiter la zone sensible du morph à la forme de la croix, vous devez surcharger la méthode `containsPoint:`.

 Définissez la méthode `containsPoint:` dans la classe `CrossMorph` :

Méthode 11.4 – Modeler la zone sensible à la souris des instances de `CrossMorph`

```
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]
```

Cette méthode suit la même logique que la méthode `drawOn:`, nous sommes donc sûrs que les points pour lesquels `containsPoint:` retourne true sont les mêmes points qui seront colorés par `drawOn:`. Notez qu'à la dernière ligne nous avons profité de la méthode `containsPoint:` de la classe `Rectangle` pour faire l'essentiel du travail.

Il reste tout de même deux problèmes avec ce code dans les méthodes 11.3 et 11.4. Le plus remarquable est que nous ayons du code dupliqué. C'est une erreur fondamentale : si vous avez besoin de modifier la façon dont `horizontalBar` ou `verticalBar` sont calculées, vous risquez d'oublier de reporter les changements effectués d'une méthode à l'autre. La solution consiste à éliminer la redondance en refactorisant ces calculs dans deux nouvelles méthodes que nous plaçons dans le protocole private :

Méthode 11.5 – `horizontalBar`

```
horizontalBar
| crossHeight |
crossHeight := self height / 3.0.
↑ self bounds insetBy: 0 @ crossHeight
```

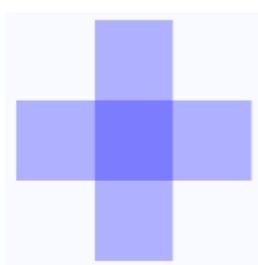


FIGURE 11.4 – Le centre de la croix est rempli deux fois avec la couleur.

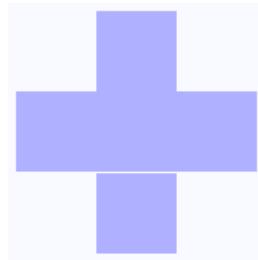


FIGURE 11.5 – Le morph en forme de croix présente une ligne de pixels non remplis.

Méthode 11.6 – verticalBar

```
verticalBar
| crossWidth |
crossWidth := self width / 3.0.
↑ self bounds insetBy: crossWidth @ 0
```

Nous pouvons ensuite définir les méthodes drawOn: et containsPoint: ainsi :

Méthode 11.7 – Refactoriser CrossMorph»drawOn:

```
drawOn: aCanvas
aCanvas fillRectangle: self horizontalBar color: self color.
aCanvas fillRectangle: self verticalBar color: self color
```

Méthode 11.8 – Refactoriser CrossMorph»containsPoint:

```
containsPoint: aPoint
↑ (self horizontalBar containsPoint: aPoint)
or: [self verticalBar containsPoint: aPoint]
```

Ce code est plus simple à comprendre principalement parce que nous avons donné des noms parlants à ces méthodes privées. En fait, notre simplification a mis en avant notre second problème : la zone centrale de notre croix, à la croisée des barres horizontales et verticales, est dessinée deux fois. Ce n'est pas très problématique tant que notre croix est de couleur opaque, mais l'erreur devient clairement apparente si nous dessinons une croix semi-transparente, comme nous pouvons le voir sur la figure 11.4.

Évaluez ligne par ligne le code suivant dans un espace de travail :

```
m := CrossMorph new bounds: (0@0 corner: 300@300).
m openInWorld.
m color: (Color blue alpha: 0.3).
```

La correction repose sur la division de la barre verticale en trois morceaux et sur le remplissage uniquement des deux morceaux supérieurs et inférieurs. Encore une fois, nous trouvons une méthode dans la classe `Rectangle` qui va bien nous aider : `r1 areasOutside: r2` retourne un tableau de rectangles comprenant les parties de `r1` exclus de `r2`.

Le code revisité de la méthode `drawOn:` peut s'écrire comme suit :

Méthode 11.9 – La méthode `drawOn:` revisitée pour ne remplir le centre qu'une seule fois

```
drawOn: aCanvas
    "topAndBottom est un tableau des parties de verticalBar tronqué"
    | topAndBottom |
    aCanvas fillRectangle: self horizontalBar color: self color.
    topAndBottom := self verticalBar areasOutside: self horizontalBar.
    topAndBottom do: [ :each | aCanvas fillRectangle: each color: self color]
```

Ce code semble fonctionner mais, suivant la taille des croix (que vous pouvez obtenir en les dupliquant et en les redimensionnant avec le halo Morphic), vous pouvez constater qu'une ligne d'un pixel de haut peut séparer la base de la croix du reste, comme le montre la figure 11.5. Ceci est du à un problème de troncature : lorsque la taille d'un rectangle à remplir n'est pas un entier, `fillRectangle: color:` semble mal arrondir et laisse donc une ligne de pixels non remplis. Nous pouvons résoudre ce problème en arrondissant explicitement lors du calcul des tailles des barres.

Méthode 11.10 – CrossMorph»horizontalBar avec troncature explicite

```
horizontalBar
    | crossHeight |
    crossHeight := (self height / 3.0) rounded.
    ↑ self bounds insetBy: 0 @ crossHeight
```

Méthode 11.11 – CrossMorph»verticalBar avec troncature explicite

```
verticalBar
    | crossWidth |
    crossWidth := (self width / 3.0) rounded.
    ↑ self bounds insetBy: crossWidth @ 0
```

11.5 Interaction et animation

Pour construire des interfaces utilisateur vivantes avec les morphs, nous avons besoin de pouvoir interagir avec elles en utilisant la souris et le clavier. En outre, les morphs doivent être capable de répondre aux interactions de l'utilisateur en changeant leur apparence et leur position, autrement dit, en s'animant eux-mêmes.

Les événements souris

Quand un bouton de la souris est pressé, Morphic envoie à chaque morph sous le pointeur de la souris le message `handlesMouseDown:`. Si un morph répond `true`, Morphic lui envoie immédiatement le message `mouseDown:`. Lorsque le bouton de la souris est relâché, Morphic envoie `mouseUp:` à ces mêmes morphs qui avaient répondu positivement. Si tous les morphs retournent `false`, Morphic entame une opération de saisie en prévision du glisser-déposer. Comme nous allons le voir, les messages `mouseDown:` et `mouseUp` sont envoyés avec un argument — un objet de classe `MouseEvent` — qui contient les détails de l'action de la souris.

Ajoutons la gestion des événements souris à notre classe `CrossMorph` en commençant par nous assurer que toutes nos croix répondent `true` au message `handlesMouseDown:`.

 Ajoutez la méthode suivante à la classe `CrossMorph` :

Méthode 11.12 – Déclarer que `CrossMorph` réagit aux clics de souris

```
CrossMorph»handlesMouseDown: anEvent
  ↑true
```

Supposons que vous vouliez que la couleur de la croix passe au rouge (`Color red`) à chaque fois que vous cliquez et qu'elle passe au jaune (`Color yellow`) lorsque vous cliquez avec le bouton d'action sur celle-ci. Nous devons créer la méthode 11.13.

À CORRIGER! MARTIAL: FAUDRAIT-IL METTRE DES MÉTHODES `#CLICKBUTTONPRESSED` ET `#ACTCLICKBUTTONPRESSED` DANS PHARO ? !

Méthode 11.13 – Réagir aux clics de la souris en changeant la couleur de la croix

```
CrossMorph»mouseDown: anEvent
  anEvent redButtonPressed "click"
    ifTrue: [self color: Color red].
  anEvent yellowButtonPressed "action-click"
    ifTrue: [self color: Color yellow].
  self changed
```

Remarquez que non seulement cette méthode change la couleur de notre morph, mais qu'elle envoie aussi le message `self changed`. Ce message assure que Morphic envoie `drawOn:` de façon assez rapide.

Notez aussi qu'une fois qu'un morph gère les événements souris, vous ne pouvez plus l'attraper avec la souris pour le déplacer. Dès lors, vous devez utiliser le halo Morphic en [meta-cliquant](#) : les poignées [supérieures noires](#)

et marron vous permettent respectivement de prendre et déplacer ce morph.

L'argument anEvent de mouseDown: est une instance de MouseEvent, sous-classe de MorphicEvent. MouseEvent définit les méthodes redButtonPressed pour la gestion du clic et yellowButtonPressed pour celle du clic d'action. Parcourez cette classe pour en savoir plus sur les autres méthodes disponibles pour la gestion des événements souris.

Les événements clavier

La capture des événements clavier se déroule en trois étapes. Morphic devra :

1. activer votre morph pour la gestion du clavier par la "mise au point" sous une certaine condition, disons, lorsque la souris est au-dessus du morph ;
2. gérer l'événement proprement dit avec la méthode handleKeystroke: — ce message est envoyé au morph quand vous pressez une touche et qu'il a déjà reçu la mise au point (en anglais, *keyboard focus*) ;
3. libérer la mise au point lorsque la condition de la première étape n'est plus remplie, disons, quand la souris n'est plus au-dessus du morph.

Occupons-nous de CrossMorph pour que nos croix réagissent à certaines touches du clavier. Tout d'abord, nous avons besoin d'être informé que la souris est au-dessus de la surface de notre morph : dans ce cas, le morph doit répondre true au message handlesMouseOver:.

Déclarez que CrossMorph réagit lorsque il est sous le pointeur de la souris.

Méthode 11.14 – Gérer les événements souris "mouse over"

```
CrossMorph»handlesMouseOver: anEvent  
  ↑true
```

Ce message est équivalent à handlesMouseDown: utilisé pour la position de la souris. Les messages mouseEnter: et mouseLeave: sont envoyés respectivement lorsque le pointeur de la souris entre dans l'espace du morph ou sort de celui-ci.

Définissez deux méthodes grâce auxquelles un morph CrossMorph peut activer et libérer la mise au point sur le clavier. Créez ensuite une troisième méthode pour gérer l'interaction via la saisie des touches.

Méthode 11.15 – Activer la mise au point sur le clavier lorsque la souris entre dans l'espace du morph

```
CrossMorph»mouseEnter: anEvent
    anEvent hand newKeyboardFocus: self
```

Méthode 11.16 – Libérer la mise au point sur le clavier lorsque la souris sort de l'espace du morph

```
CrossMorph»mouseLeave: anEvent
    anEvent hand newKeyboardFocus: nil
```

Méthode 11.17 – Capturer et gérer les événements clavier

```
CrossMorph»handleKeystroke: anEvent
    | keyValue |
    keyValue := anEvent keyValue.
    keyValue = 30 "flèche du haut"
        ifTrue: [self position: self position - (0 @ 1)].
    keyValue = 31 "flèche du bas"
        ifTrue: [self position: self position + (0 @ 1)].
    keyValue = 29 "flèche de droite"
        ifTrue: [self position: self position + (1 @ 0)].
    keyValue = 28 "flèche de gauche"
        ifTrue: [self position: self position - (1 @ 0)]
```

La méthode que nous venons d'écrire vous permet de déplacer notre croix avec les touches fléchées. Remarquez que lorsque la souris n'est pas sur la croix, le message handleKeystroke: n'est pas envoyé : dans ce cas, la croix ne répond pas aux commandes clavier. Vous pouvez connaître la valeur des touches saisies au clavier en ouvrant une fenêtre Transcript et en ajoutant à méthode 11.17 la ligne Transcript show: anEvent keyValue. L'événement-argument anEvent de handleKeystroke est une instance de la classe KeyboardEvent , sous-classe de MorphicEvent. Naviguez dans cette classe pour connaître les méthodes de gestion des événements clavier.

Les animations Morphic

Pour l'essentiel, Morphic permet de composer et d'automatiser de simples animations grâce à quatre méthodes :

- step qui est envoyé au morph à un *tempo* régulier pour construire le comportement de l'animation ;
- stepTime qui définit l'intervalle de temps en millisecondes entre chaque envoi du message step² ;

2. stepTime est en réalité le temps *minimum* entre les envois du message step. Si vous demandez un *tempo* stepTime de 1 ms, ne soyez pas étonné si Pharo est trop occupé pour que le rythme de l'animation de votre morph tienne cette cadence.

– startStepping démarre l’animation au rythme du métronome stepTime ;
 – stopStepping arrête l’animation.
 à ces méthodes s’ajoute une méthode de test isStepping pour savoir si le morph est en cours d’animation.

 Faites clignoter le CrossMorph en définissant les méthodes suivantes :

Méthode 11.18 – Définir la périodicité de l’animation

```
CrossMorph»stepTime
↑ 100
```

Méthode 11.19 – Construire le comportement de l’animation

```
CrossMorph»step
(self color diff: Color black) < 0.1
  ifTrue: [self color: Color red]
  ifFalse: [self color: self color darker]
```

Pour démarrer l’animation, vous pouvez ouvrir un inspecteur sur votre objet CrossMorph : cliquez sur la poignée de débogage  du halo Morphic de votre croix ([en meta-cliquant](#)) puis choisissez `inspect morph` dans le menu flottant. Entrez l’expression `self startStepping` dans le mini-espace de travail situé dans le bas de l’inspecteur et faites un `do it`. Pour arrêter l’animation, évaluez simplement `self stopStepping` dans l’inspecteur. Pour démarrer et arrêter l’animation de façon plus efficace, vous pouvez ajouter des contrôles supplémentaires au clavier. Par exemple, vous pouvez modifier la méthode `handleKeystroke:` pour que la touche + démarre le clignotement de la croix et que la touche – le stoppe.

 Ajoutez le code suivant à méthode 11.17 :

```
keyValue = $+ asciiValue
  ifTrue: [self startStepping].
keyValue = $- asciiValue
  ifTrue: [self stopStepping].
```

Les interacteurs

Morphic dispose de morphs commodes pour créer en quelques lignes de code des interactions avec l’utilisateur. Parmi eux, nous avons la classe UIManager offre un grand nombre de boîtes de dialogue prêtes à l’emploi. La méthode `request:initialAnswer:` renvoie une chaîne de caractères entrée par l’utilisateur (voir la figure 11.6).

```
UIManager default request: 'Quel est votre nom?' initialAnswer: 'sans nom'
```



FIGURE 11.6 – Une boîte de dialogue affichée par UIManager request: 'Quel est votre nom?' initialAnswer: 'sans nom'.



FIGURE 11.7 – Un menu flottant.

Pour afficher le menu flottant (en anglais, *pop-up menu*), [utilisez une des méthodes chooseFrom: \(voir la figure 11.7\)](#) :

```
UIManager default
chooseFrom: #(' cercle' ' ovale' ' carré' ' rectangle' ' triangle')
lines: #(2 4) message: ' Choisissez une forme'
```

11.6 Le glisser-déposer

Morphic supporte aussi le glisser-déposer. Étudions l'exemple suivant. Créons tout d'abord un morph receveur qui n'acceptera un morph que si le dépôt de ce morph se fait dans une certaine condition. Créons ensuite un second morph que nous appelons morph déposé. Le fait que le morph soit bleu (Color blue) sera notre condition pour que le glisser-déposé se fasse ici.

Définissez la classe pour le morph receveur et créez une méthode dinitialisation comme suit :

Classe 11.20 – Définir un morph sur lequel un autre morph pourra être déposé

```
Morph subclass: #ReceiverMorph
instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: ""
category: 'PBE-Morphic'
```

Méthode 11.21 – Initialiser un objet ReceiverMorph

```
ReceiverMorph»initialize
super initialize.
color := Color red.
bounds := 0 @ 0 extent: 200 @ 200
```

Comment décidons-nous si le receveur va accepter ou refuser le morph déposé ? En général, ces deux morphs devront s'accorder sur leur interaction. Le receveur fait cela en répondant au message `wantsDroppedMorph:event:` ; le premier argument est le morph que nous voulons déposer et le second est l'événement souris. Ce dernier argument permet, par exemple, au receveur de savoir si une (ou plusieurs) touche de modification a été maintenue enfoncée durant la phase de dépôt de l'autre morph. Le morph déposé, quant à lui, se doit de vérifier s'il est compatible avec le morph sur lequel il est déposé ; le message `wantsToBeDroppedInto:` doit répondre true si le morph receveur passé en argument est défini comme compatible. L'implémentation de cette méthode dans la classe mère des morphs Morph renvoie toujours true donc, par défaut, tous les morphs sont acceptés en tant que receveur.

Méthode 11.22 – Accepter les morphs déposés selon leur couleur

```
ReceiverMorph»wantsDroppedMorph: aMorph event: anEvent
↑ aMorph color = Color blue
```

Qu'arrive-t-il au morph déposé si le morph receveur ne veut pas de lui ? Le comportement par défaut de l'interface Morphic est de ne rien faire, c-à-d. de laisser le morph déposé au-dessus du morph receveur sans aucune interaction avec celui-ci. Le morph déposé aurait un comportement plus intuitif s'il retournait à sa position d'origine en cas de refus. Nous pouvons faire cela en disant au receveur de répondre true au message `repelsMorph:event:` lorsque celui-ci ne veut pas du morph déposé :

Méthode 11.23 – Changer le comportement du morph déposé lorsqu'il est rejeté

```
ReceiverMorph»repelsMorph: aMorph event: anEvent
↑ (self wantsDroppedMorph: aMorph event: anEvent) not
```

C'est tout ce dont nous avons besoin.

 Créez des instances de ReceiverMorph et de EllipseMorph dans un espace de travail :

```
ReceiverMorph new openInWorld.
EllipseMorph new openInWorld.
```

Essayez de faire un glisser-déposer de l'ellipse jaune EllipseMorph sur le morph receveur rouge. Il sera rejeté et retournera à sa position initiale.

 Changez la couleur de l'ellipse pour du bleu via l'inspecteur (que vous pouvez activer avec le menu de la poignée du débogage du halo Morphic en cliquant sur `inspect morph`) : évaluez `self color: Color blue`. Les morphs bleus étant acceptés par le `ReceiverMorph` : essayez à nouveau le glisser-déposer.

Bravo ! Vous venez de faire un glisser-déposer.

Continuons à explorer le glisser-déposer en créant un morph déposé spécifique nommé `DroppedMorph`, sous-classe de `Morph` :

Classe 11.24 – *Définir un morph que nous pouvons glisser-déposer sur un ReceiverMorph*

```
Morph subclass: #DroppedMorph
instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: ""
category: 'PBE–Morphic'
```

Méthode 11.25 – *Initialiser DroppedMorph*

```
DroppedMorph»initialize
super initialize.
color := Color blue.
self position: 250@100
```

Nous voulons que le morph déposé ait un nouveau comportement lorsqu'il est rejeté par le receveur ; cette fois-ci, il restera attaché au pointeur de la souris :

Méthode 11.26 – *Réagir lorsque le morph est rejeté lors du dépôt*

```
DroppedMorph»rejectDropMorphEvent: anEvent
| h |
h := anEvent hand.
WorldState
addDeferredUIMessage: [h grabMorph: self].
anEvent wasHandled: true
```

L'envoi du message `hand` à un événement répond la "main" (en anglais, `hand`), instance de `HandMorph` qui représente le pointeur de la souris et tout ce qu'il tient. Dans notre méthode, nous disons à l'écran Pharo, `World`, que la main (stockée dans la variable temporaire `h`) doit capturer le morph rejeté `self` grâce au message `grabMorph:..`. La méthode `wasHandled:` détermine si l'événement était capturé.

 Créer deux instances de `DroppedMorph` et faites un glisser-déposer pour chacune sur le receveur.

```
ReceiverMorph new openInWorld.  
(DroppedMorph new color: Color blue) openInWorld.  
(DroppedMorph new color: Color green) openInWorld.
```

Le morph vert (Color green) est rejeté et reste ainsi attaché au pointeur de la souris.

11.7 Le jeu du dé

Lançons-nous maintenant dans la création d'un jeu du dé complet. Nous voulons faire défiler toutes les faces d'un dé dans une boucle rapide suite à un premier clic de souris sur la surface de ce dé puis, lors d'un second clic, arrêter l'animation sur une face.



FIGURE 11.8 – Le dé dans Morphic.

Définissez un dé comme une sous-classe de BorderedMorph définissant un Morph avec un bord :appelez-le DieMorph (dé se dit die en anglais).

Classe 11.27 – Définir le dé DieMorph

```
BorderedMorph subclass: #DieMorph  
instanceVariableNames: 'faces dieValue isStopped'  
classVariableNames: ""  
poolDictionaries: ""  
category: 'PBE–Morphic'
```

La variable d'instance `faces` stocke le nombre de faces de notre dé ; nous nous autorisons à avoir des dés jusqu'à neuf faces ! `dieValue` contient la valeur de la face affichée en ce moment et `isStopped` est un booléen qui est true si et seulement si l'animation est à l'arrêt. Nous allons définir la *méthode de classe* `faces: n` dans le côté classe de `DieMorph` pour pouvoir créer un nouveau dé à `n` faces.

Méthode 11.28 – Créez un nouveau dé avec un nombre de faces déterminé
`DieMorph class»faces: aNumber`

```
↑ self new faces: aNumber
```

La méthode initialize est définie dans le côté instance de la classe ; souvenez-vous que new envoie initialize à toute instance nouvellement créée.

Méthode 11.29 – Initialiser les instances de DieMorph

```
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
```

Nous utilisons quelques méthodes de la classe BorderedMorph pour donner un aspect sympathique à notre dé : bordure épaisse avec un effet de relief, coins arrondis et dégradé de couleur sur la face visible. Nous définissons ensuite la méthode d'instance faces: pour affecter la variable d'instance — il s'agit d'une méthode d'accès de type mutateur — en vérifiant que le paramètre est bien valide :

Méthode 11.30 – Affecter le nombre correspondant à la face visible de dé

```
DieMorph»faces: aNumber
"Affecter le numéro de la face"
(aNumber isInteger
    and: [aNumber > 0]
    and: [aNumber <= 9])
ifTrue: [faces := aNumber]
```

Comprenez bien l'ordre dans lequel les messages sont envoyés lors de la création d'un dé. Si nous évaluons DieMorph faces: 9 :

1. la méthode de classe DieMorph class»faces: envoie new à DieMorph class ;
2. la méthode pour new (héritée par DieMorph class de Behavior) crée la nouvelle instance et lui envoie le message initialize ;
3. la méthode initialize de DieMorph affecte la valeur initiale 6 à faces ;
4. DieMorph class»new retourne à la méthode de classe DieMorph class»faces: qui envoie ensuite le message faces: 9 à la nouvelle instance ;
5. la méthode d'instance DieMorph»faces: s'exécute maintenant en affectant à la valeur 9 la variable d'instance faces.

Pour positionner les points noirs sur la face du dé, nous devons besoin de définir autant de méthodes qu'il y a de faces possibles :

Méthodes 11.31 – Neuf méthodes pour placer les points noirs sur la face visible du dé

```
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}
```

Ces méthodes définissent des collections de coordonnées de points pour chaque configuration de faces possible. Les coordonnées sont dans un carré de dimension 1×1 . Pour placer nos points, nous effectuons simplement un changement d'échelle.

Enfin, pour dessiner la face du dé, nous définissons la méthode `drawOn:` qui fera d'abord un envoi sur `super`, utilisant la méthode définie dans une classe-mère pour dessiner le fond de la face, et qui exploitera, dans un deuxième temps, les méthodes créées précédemment pour dessiner les points noirs.

Méthode 11.32 – Dessiner le dé

```
DieMorph»drawOn: aCanvas
super drawOn: aCanvas.
(self perform: ('face' , dieValue asString) asSymbol)
do: [:aPoint | self drawDotOn: aCanvas at: aPoint]
```

Les capacités réflexives de Smalltalk sont utilisées dans la dernière expression de cette méthode. Dessiner les points noirs d'une face revient à itérer sur la collection de coordonnées retournée par la méthode `faceX` (`X` est issu de la variable d'instance `dieValue` correspondant au numéro de la face en cours), en envoyant le message `drawDotOn:at:` pour chacune de ces coordonnées. Pour joindre la bonne méthode `faceX`, nous utilisons la méthode `perform:`

qui envoie le message construit à partir d'une chaîne de caractères ('face', `dieValue asString`) `asSymbol`. Cet usage de la méthode `perform:` est très fréquent.

Méthode 11.33 – Dessiner un simple point noir sur une face

```
DieMorph»drawDotOn: aCanvas at: aPoint
  aCanvas
    fillOval: (Rectangle
      center: self position + (self extent * aPoint)
      extent: self extent / 6)
    color: Color black
```

Puisque les coordonnées sont normées dans l'intervalle [0:1], elles sont mises à l'échelle des dimensions du dé avec `self extent * aPoint`.

 Créez une instance de dé dans un espace de travail :

```
(DieMorph faces: 6) openInWorld.
```

Pour pouvoir modifier la valeur de la face visible, nous devons créer un mutateur aussi pour `dieValue`. Grâce à elle, nous pourrions, par exemple, afficher la face à 4 points depuis une nouvelle méthode de la classe en y écrivant `self dieValue: 4`.

Méthode 11.34 – Affecter un nombre à la valeur courante du dé

```
DieMorph»dieValue: aNumber
  (aNumber isInteger
    and: [aNumber > 0]
    and: [aNumber <= faces])
  ifTrue:
    [dieValue := aNumber.
    self changed]
```

Nous allons utiliser le système d'animation pour faire défiler rapidement et aléatoirement (avec le message `atRandom`) toutes les faces du dé :

Méthodes 11.35 – Animer le dé

```
DieMorph»stepTime
  ↑ 100

DieMorph»step
  isStopped ifFalse: [self dieValue: (1 to: faces) atRandom]
```

Désormais, notre dé "roule" !

Pour démarrer ou arrêter l'animation par un clic de souris, nous utiliserons ce que nous avons préalablement appris sur les événements souris. Nous activons la réception des événements de la souris et nous décrivons notre gestion du clic dans la méthode `mouseDown:`.

Méthodes 11.36 – Gérer les clics de souris pour démarrer et arrêter l’animation

```
DieMorph»handlesMouseDown: anEvent
  ↑ true
```

```
DieMorph»mouseDown: anEvent
  anEvent redButtonPressed "click"
    ifTrue: [isStopped := isStopped not]
```

Maintenant notre dé “roule” ou se fige quand nous cliquons dessus.

11.8 Gros plan sur le canevas

La méthode drawOn: a un canevas, instance de Canvas, comme unique argument ; le canevas est l’espace dans lequel le morph se dessine. En utilisant les méthodes graphiques du canevas, vous êtes libre de donner l’apparence que vous voulez à votre morph. Si vous parcourez la hiérarchie d’héritage de la classe Canvas, vous constaterez plusieurs variantes. Par défaut, nous utilisons FormCanvas. Cette classe et sa classe-mère Canvas contiennent les méthodes graphiques essentielles pour dessiner des points, des lignes, des polygones, des rectangles, des ellipses, du texte et des images avec rotation et changement d’échelle.

Vous pouvez aussi utiliser d’autres types de canevas pour obtenir, par exemple, des méthodes supplémentaires ou encore, ajouter la transparence ou l’anti-crénelage (ou *anti-aliasing*³) aux morphs. Vous aurez besoin dans ces cas-là de canevas tels que AlphaBlendingCanvas ou BalloonCanvas. Pour obtenir un canevas différent dans la méthode drawOn: alors que son argument est une instance de FormCanvas, vous devrez court-circuiter le canevas courant par un autre.

❶ Redéfinissez drawOn: de la classe DieMorph pour utiliser un canevas semi-transparent :

Méthode 11.37 – Dessiner un dé semi-transparent

```
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]
```

C’est tout ce dont nous avons besoin !

3. Ce rendu est utilisé pour atténuer ou éliminer l’effet escalier des pixels.

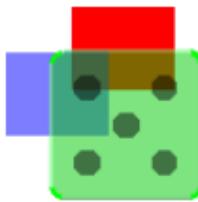


FIGURE 11.9 – Le dé semi-transparent.

11.9 Résumé du chapitre

Morphic comporte une librairie graphique grâce à laquelle les éléments de l'interface graphique peuvent être composés dynamiquement. Vous pouvez :

- convertir un objet en *morph* et l'afficher sur l'écran de Pharo, le *world*, en lui envoyant le message `asMorph openInWorld` ;
- faire apparaître le halo Morphic en [meta-cliquant](#) sur un morph et manipuler ce morph grâce aux poignées du halo. Ces poignées ont des ballons d'aide (ou *help balloons*) qui détaillent leur action ;
- composer des morphs en les emboîtant les uns dans les autres, soit par glisser-déposer, soit par envoi du message `addMorph:` ;
- dériver la classe d'un morph et redéfinir ses méthodes-clés telles que `initialize` et `drawOn:` ;
- contrôler la façon dont réagit un morph avec les événements issus de la souris et du clavier en redéfinissant les méthodes comme, par exemple, `handlesMouseDown:` et `handlesMouseOver:` ;
- animer un morph en définissant les méthodes `step` (ce que fait le morph) et `stepTime` (le nombre de millisecondes entre les pas) ;
- trouver différents morphs pré-définis pour l'interactivité de l'utilisateur comme `PopUpMenu` ou `UIManager` ;
- explorer les méthodes graphiques des différents canevas, instances de `Canvas` ou de sous-classes, pour exploiter leurs ressources pour le dessin des morphs.

Chapitre 12

Seaside par l'exemple

Seaside est un *framework* destiné à la construction d'applications web en Smalltalk. Il a été initialement développé par Avi Bryant en 2002 ; une fois maîtrisé, Seaside permet la création d'applications web aussi facilement que se fait l'écriture d'applications classiques.

Deux des applications basées sur Seaside les plus connues sont Squeak-Source¹ et Dabble DB². Seaside n'est pas commun dans le sens qu'il est réellement orienté objet : il n'y a aucune structure de type *template XHTML*, aucun flux de contrôle complexe par des pages web, ni même d'encodage de l'état dans les URLs. Vous envoyez simplement des messages aux objets. Quelle merveilleuse idée !

12.1 Pourquoi avons-nous besoin de Seaside ?

Les applications web modernes essayent d'intéragir avec l'utilisateur de la même façon que les applications de bureau : elles interrogent l'utilisateur et celui-ci leur répond, habituellement en remplissant un formulaire ou en cliquant sur un bouton. Mais le web fonctionne d'une autre manière : le navigateur de l'utilisateur fait des requêtes au serveur et le serveur répond avec une nouvelle page web. Dès lors les *frameworks* destinées au développement d'applications web doivent contourner une foule de problèmes, le principal étant la gestion du flux de contrôle "inversé". C'est pour cette raison que beaucoup d'applications web tentent d'interdire l'usage du bouton "retour arrière" (ou "back" en anglais, nommé *précedent(e)* sur certains navigateurs) du fait de la difficulté de garder une trace de l'état d'une session. Diffuser des flux de contrôle significatifs entre plusieurs pages web est sou-

1. <http://SqueakSource.com>

2. <http://DabbleDB.com>

vent lourd et la diffusion de plusieurs flux peut être difficile.

Seaside est un *framework* orienté composant qui allège le développement web de plusieurs façons. Tout d'abord, le flux de contrôle peut être exprimé naturellement en utilisant les envois de messages. Seaside garde une trace de la correspondance entre page web et un point précis dans l'exécution de l'application web. Donc le bouton "retour arrière" du navigateur web fonctionnement correctement.

Deuxièmement, la gestion de l'état est automatique. En tant que développeur, vous aurez le choix de permettre le *backtracking* (*c-à-d.* le chaînage arrière) de l'état ou non, ainsi la navigation en "retour arrière" annulera les effets de bord. Vous pouvez autrement utiliser le [support de transaction](#) inclus dans Seaside pour empêcher les utilisateurs d'annuler les effets de bord permanents lorsqu'ils utilisent le bouton "retour arrière". Vous n'avez pas à encoder l'information de l'état dans l'URL — Seaside s'en occupe pour vous.

Ensuite, les pages web sont générées depuis les composants imbriqués, chacun pouvant avoir son propre flux de contrôle indépendant. Il n'y a pas de *templates XHTML* — le code XHTML valide est généré de façon programmatique via un simple protocole Smalltalk. Seaside supporte les feuilles de style en cascade (CSS), ainsi le contenu et la mise en page sont clairement séparés.

Finalement, Seaside offre une interface web pratique pour le développement en facilitant la programmation itérative, le débogage interactif et la recompilation et l'extension des applications alors que le serveur reste en activité.

12.2 Démarrer avec Seaside

Télécharger le logiciel "Seaside One-Click Experience" depuis le site web³ est la façon la plus simple pour commencer avec Seaside. Il s'agit d'une version "clé en main" de Seaside 2.8 pour Mac OS X, Linux et Windows. Le site web de Seaside liste aussi de nombreux liens vers des ressources complémentaires dont de la documentation et des tutoriels (en anglais). Sachez, cependant, que Seaside a considérablement évolué durant ces dernières années et toutes ces ressources disponibles ne se rapportent pas nécessairement à la dernière version de Seaside.

Seaside inclut un serveur web : vous pouvez lancer ce serveur sur le port 8080 en évaluant `WAKom startOn: 8080` et le stopper en évaluant `WAKom stop`. Dans l'installation par défaut, le compte administrateur (en anglais, *administrator login*) est `admin` et le mot de passe (ou *password*) est `seaside`. Pour les changer, évaluez : `WADispatcherEditor initialize`. Ceci affichera une fenêtre de

3. <http://seaside.st>

dialogue pour la saisie d'un nouveau nom et d'un nouveau mot de passe.

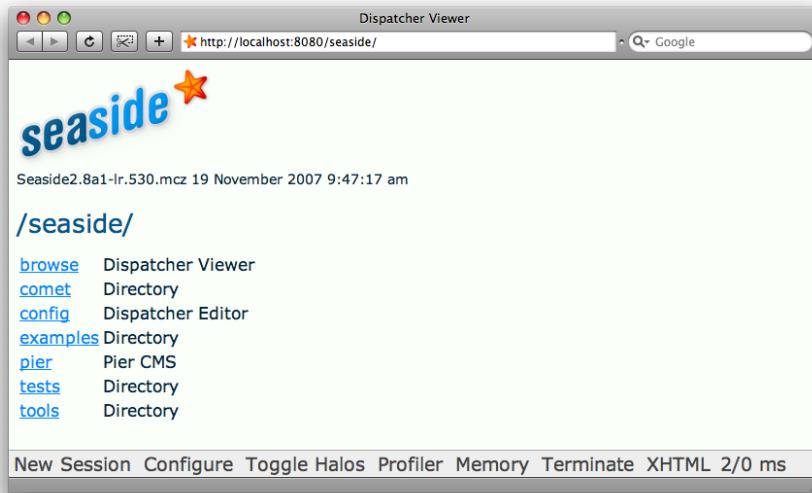


FIGURE 12.1 – La page de démarrage de Seaside.

- ❶ Démarrez le serveur Seaside et rendez-vous sur la page `http://localhost:8080/seaside/` avec votre navigateur favori.

Vous devriez voir une page web semblable à celle de la figure 12.1.

- ❷ Naviguez vers la page `examples>counter` (voir la figure 12.2).

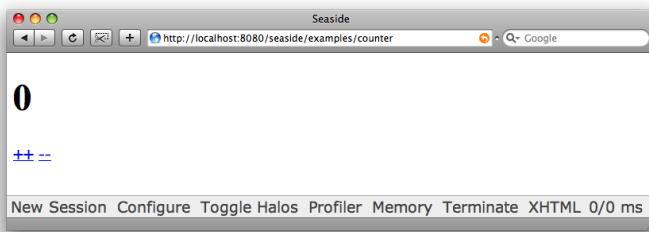


FIGURE 12.2 – La démo du compteur avec l'application “counter”.

Cette page est une petite application Seaside nommée counter : elle affiche

un compteur qui peut être incrémenté et déCREMENTé en cliquant sur les liens [++](#) et [--](#).

💡 Amusez-vous un peu avec l'application “counter” en cliquant sur ces liens. Utiliser le bouton “retour arrière” du navigateur pour revenir à l'état précédent et cliquer ensuite sur [++](#). Remarquez comment le compteur est correctement incrémenté depuis l'état actuellement affiché au lieu d'être incrémenté depuis l'état dans lequel le compteur était lorsque vous aviez commencé à utiliser le bouton “retour arrière”.

Notez la barre d'outils en bas de la page web dans la figure 12.1. Seaside supporte la notion de “sessions” pour suivre l'état de l'application pour différents utilisateurs. New Session démarre une nouvelle session sur l'application “counter”. Configure vous permet de configurer les paramètres de votre application via une interface web (pour fermer la vue Configure, cliquez sur le x dans le côté supérieur droit de la page). Toggle Halos propose d'explorer l'état de l'application lancée sur le serveur Seaside. Profiler et Memory offre une information détaillée à propos des performances à l'exécution de l'application. XHTML peut être utilisé pour valider la page web générée ; ce lien ne fonctionne que lorsque la page est accessible depuis Internet car il utilise le service de validation W3C.

Les applications Seaside sont construites à l'aide de “composants” interconnectables. Ces composants sont en réalité des objets Smalltalk ordinaires. Leur seule particularité est qu'ils doivent être des instances de classes héritées de la classe du framework Seaside appelée `WAComponent`. Nous pouvons explorer les composants et leurs classes dans l'image Pharo ou directement, depuis l'interface web, en utilisant les halos.



FIGURE 12.3 – Les halos dans Seaside.

 Sélectionnez **Toggle Halos**

. Vous devriez voir une page web comme celle de la figure 12.3. Dans le coin supérieur gauche, le texte WACounter nous indique le nom de la classe du composant Seaside qui implémente le comportement de cette page. À droite de ce texte, il y a trois icônes cliquables. Le premier, représentant un crayon, active un navigateur de classes Seaside sur la classe WACounter. Le second icône avec une loupe ouvre un inspecteur d'objets sur l'instance WACounter en cours d'activité. Le dernier, sous la forme de cercles colorés, affiche la feuille de style CSS de ce composant. Dans le coin supérieur droit, le **R** et le **S** vous laissent basculer entre la vue du code rendue et celle du code source. Essayez donc en cliquant sur ces différents liens. Remarquez que les liens **++** et **-** sont aussi actifs dans la vue du code source. Comparez d'ailleurs le formatage en couleurs du code source fourni par les halos Seaside avec la vue du code source non-formatée fournie par votre navigateur.

Le navigateur de classes Seaside et l'inspecteur d'objets peuvent être très pratiques lorsque le serveur tourne sur un autre ordinateur, surtout si celui-ci n'a pas d'écran ou qu'il se trouve dans un lieu distant. Néanmoins, lorsque vous commencez à développer une application Seaside, le serveur sera lancé localement et il sera plus facile d'utiliser les outils de développement standards disponibles dans l'image Pharo dans laquelle votre serveur est en activité.



FIGURE 12.4 – Suspendre l'application “counter”.

 Cliquez sur le lien *Object Inspector* représenté par l'icône de loupe du halo pour ouvrir un inspecteur d'objets sur l'objet-compteur Smalltalk et évaluez `self halt` (en cliquant sur **doit**). La page web s'arrêtera de charger. Allez maintenant dans votre image Seaside. Vous devriez voir une fenêtre **de pré-débogueur** affichant un objet WACounter exécutant un halt (voir la figure 12.4). Examinez cette exécution dans le débogueur en cliquant sur la première ligne de la pile, puis cliquez sur **Proceed**. Retourner dans votre navigateur web et notez que l'application “counter” fonctionne à nouveau.

Les composants Seaside peuvent être instanciés plusieurs fois et dans différents contextes.

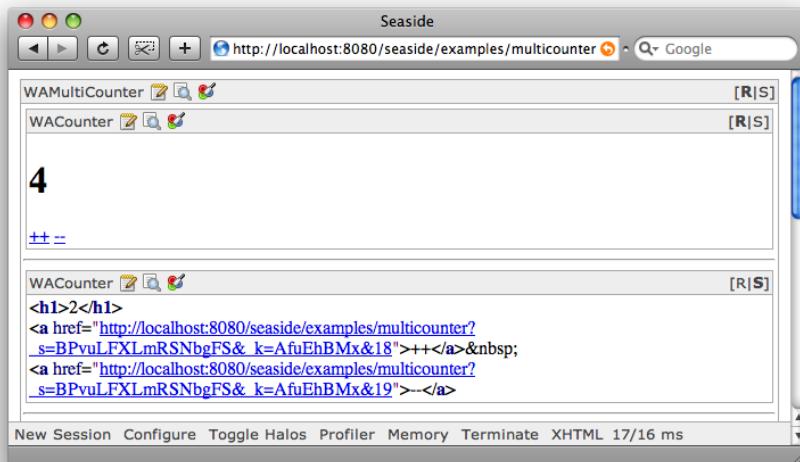


FIGURE 12.5 – Les sous-composants indépendants.

➊ Allez sur la page <http://localhost:8080/seaside/examples/multicounter> avec votre navigateur web. Vous verrez une application construite grâce à un certain nombre d'instances indépendantes du composant "counter". Incrémentez et décrémentez plusieurs de ces compteurs. Vérifiez que ces compteurs se comportent correctement même lorsque vous utilisez le bouton "retour arrière". Activez les halos pour voir que la structure de l'application est faite de composants imbriqués. Utilisez le navigateur de classes Seaside pour visualiser l'implémentation de WAMultiCounter. Vous devriez voir trois méthodes du côté classe (canBeRoot, description et initialize) ainsi que trois méthodes du côté instance (children, initialize et renderContentOn:). Remarquez qu'une application est simplement un composant qui peut être à la racine de la hiérarchie de composants ; cette possibilité de devenir racine, et donc application, est indiquée en définissant une méthode de classe canBeRoot (en anglais, "peut être racine") pour qu'elle réponde true.

Vous pouvez utiliser l'interface web Seaside, copier ou enlever une ou plusieurs applications (c-à-d. les composants racines). Essayez de changer la configuration comme suit.

➋ Allez sur <http://localhost:8080/seaside/config> avec votre navigateur. Saisissez le login et le mot de passe (admin et seaside par défaut). Sélectionnez Configure à côté de "examples". Sous l'en-tête "Add entry point" (c-à-d. ajouter un point d'entrée), saisissez le nouveau nom "counter2" pour le type Application et cliquez sur le bouton **Add** comme le montre la figure 12.6.

Sur l'écran suivant, choisissez WACounter comme composant racine Root Component puis, cliquez sur **Save** et enfin, **Close**. Nous avons maintenant un nouveau compteur installé à l'URL `http://localhost:8080/seaside/examples/counter2`. Utilisez la même interface de configuration pour enlever ce point d'entrée.

The screenshot shows two parts of the Seaside configuration interface:

- Top Panel (Settings):**
 - Default entry point:** A dropdown menu set to "(none)" with a "Submit" button.
 - Add entry point:**
 - Name:** `counter2`
 - Type:** `Application` (selected from a dropdown)
 - Add** button
- Bottom Panel (General Configuration):**

General	
Deployment Mode	false
Error Handler	WAWalkbackErrorHandler
Main Class	WARenderLoopMain
Redirect Continuation Class	WARedirectContinuation
Redirect Handler	WARedirectHandler
Render Continuation Class	WARenderContinuation
Root Component	WACounter (highlighted in blue)
Session Class	WASession
Session Expiry Seconds	600
Use Session Cookie	false

Arrows indicate the flow from the configuration settings to the resulting application structure:

- An arrow points from the "Root Component" dropdown in the configuration to the "WACounter" entry in the application directory listing.
- An arrow points from the "Save" button in the configuration to the "Configure" link for the "counter2" entry in the application directory.

Application Directory Listing:

```
/seaside/examples/
  ..
  counter   A very simple Seaside application   Configure Copy Remove
  counter2  A very simple Seaside application   Configure Copy Remove
  examplebrowser Browse through Seaside examples   Configure Copy Remove
  multicounter Multiple Seaside components on one page   Configure Copy Remove
```

FIGURE 12.6 – Configurer une nouvelle application.

Seaside fonctionne dans deux modes : soit le mode de développement dit *development* qui est le mode dans lequel nous venons de travailler, soit dans le mode de diffusion appelé *deployment* dans lequel la barre d'outils n'est pas accessible. Vous pouvez mettre Seaside dans le mode *deployment* en utilisant soit la page de configuration — pour cela naviguer jusqu'à l'entrée de l'application et cliquez sur le lien **Configure** — soit cliquez directement sur le bouton **Configure** dans la barre d'outils. Dans chacun de ces cas, mettez le mode *deployment* à *true*. Notez que ceci n'affecte que les nouvelles sessions. Vous pouvez aussi activer ce mode en évaluant `WAGlobalConfiguration setDeployment true`.

ploymentMode, et inversement, réactiver le mode *development* en évaluant WAGlobalConfiguration setDevelopmentMode.

En fait, la page web de configuration n'est qu'une autre application Seaside. Ainsi elle peut aussi être pilotée depuis la page de configuration. Si vous enlevez l'application "config", vous pouvez toujours la réinstaller en évaluant WADispatcherEditor initialize.

12.3 Les composants Seaside

Comme nous l'avons mentionné précédemment, les applications de Seaside sont bâties sur une agrégation de composants — en anglais *component*. Regardons comment Seaside fonctionne en programmant le composant "hello world".

Tout composant Seaside doit hériter directement ou indirectement de WAComponent comme le montre la figure 12.8.

 Définissez une sous-classe de WAComponent appelée HelloWorld.

Les composants doivent savoir comment faire leur propre rendu visuel. Ceci est fait habituellement en implémentant la méthode renderContentOn: qui prend comme argument une instance de WAHtmlCanvas ; ce dernier sait comment faire le rendu en XHTML.

 Implémentez la méthode suivante et mettez-la dans le protocole rendering :

```
WAHelloWorld»renderContentOn: html
    html text: 'hello world'
```

Nous devons dire maintenant à Seaside que ce composant est destiné à être autonome (donc une possible racine de l'arborescence de composants).

 Écrivez la méthode de classe suivante pour WAHelloWorld.

```
WAHelloWorld class»canBeRoot
    ↑ true
```

Nous avons fini !

 Pointez votre navigateur web <http://localhost:8080/seaside/config>, ajoutez une nouvelle entrée qui vous appellerez "hello" et choisissez WAHelloWorld comme composant racine.

Pointez maintenant votre navigateur à l'URL <http://localhost:8080/seaside/hello>. Vous y êtes ! Vous devriez voir une page web comme sur la figure 12.7.

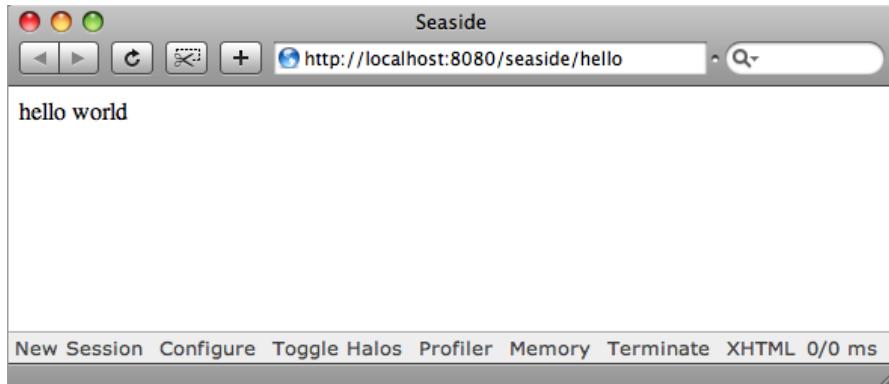


FIGURE 12.7 – “hello world” en Seaside.

Le *backtracking* d'état et l'application “counter”

L'application “counter” est seulement un peu plus compliquée que l'application “hello world”.

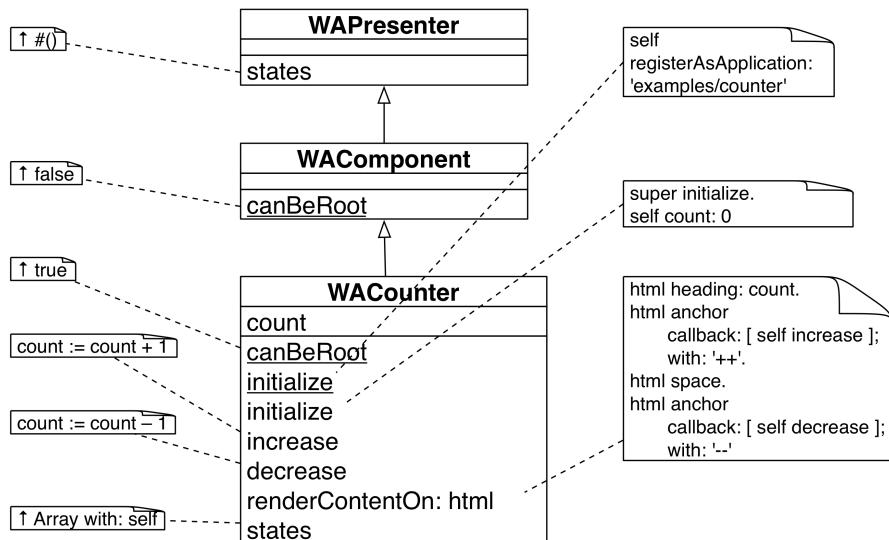


FIGURE 12.8 – La classe WACounter implémentant l'application “counter”. Les méthodes avec les noms soulignés sont des méthodes de classe ; les autres sont des méthodes d'instance.

La classe WACounter est une application autonome, ainsi WACounter class

doit répondre true au message `canBeRoot`. Elle doit aussi être enregistrée en tant qu'application ; sa méthode de classe `initialize` est utilisée pour cela (voir la figure 12.8).

`WACounter` définit deux méthodes, `increase` et `decrease`, qui seront déclenchés par les liens `++` et `-` sur la page web. Cette classe définit aussi une variable d'instance `count` pour enregistrer l'état du compteur. Cependant, nous voulons aussi que Seaside synchronise le compteur avec la page web : lorsque l'utilisateur clique sur le bouton "retour arrière" de son navigateur web, nous voulons que Seaside rappelle l'état de l'objet `WACounter`. Seaside inclut un mécanisme général pour le chaînage arrière ou *backtracking* mais chaque application doit signaler à Seaside quelle partie de son état doit être suivie.

Un composant permet le *backtracking* en implémentant la méthode d'instance `states` : `states` devrait retourner un tableau contenant tous les objets à suivre. Dans notre cas, l'objet `WACounter` s'ajoute lui-même à la table Seaside des objets à suivre pour le *backtracking* en retournant `Array with: self`.

Avertissement. Il y a un point subtil mais important à surveiller lorsque vous déclarez des objets pour le *backtracking*. Seaside suit l'état en faisant une *copie* de tous les objets déclarés dans le tableau `states`. Ceci se fait via l'objet `WASnapshot` ; `WASnapshot` est une sous-classe de `IdentityDictionary` qui enregistre les objets à être suivis en tant que clés et des *copies superficielles* (*c-à-d.* en `shallowCopy`) en tant que valeurs. *S'il y a backtracking de l'état d'une application vers une capture particulière de l'état, l'état de chaque objet entré dans ce dictionnaire est écrasé par la copie sauvegardée dans l'image du système.* À CORRIGER! MARTIAL: REVOIR CE PARAGRAPHE.

Voilà le point critique : Dans le cas de `WACounter`, vous pourriez vous dire que l'état à suivre est une valeur numérique — la valeur de la variable d'instance `count`. Cependant, la méthode `states` répondant `Array with: count` ne marche pas. Ceci est du au fait que l'objet `count` est un entier et un entier est immuable. Les méthodes `increase` et `decrease` ne changent pas l'état de l'objet 0 en 1 ou l'objet 3 en 2. Au lieu de ça, elles associent `count` à un autre entier : chaque fois que le compteur est incrémenté ou décrémenté, l'objet nommé `count` est remplacé par un autre. C'est pourquoi `WACounter»states` doit retourner `Array with: self`. Lorsque l'état d'un objet `WACounter` est remplacé par un état précédent, la valeur de chacune des variables d'instance dans l'objet est remplacée par une valeur précédente ; ceci remplace correctement la valeur actuelle de `count` par une valeur précédente.

12.4 Le rendu XHTML

Le but d'une application web est la création, ou le "rendu", de pages web. Comme nous l'avons dit dans la section 12.3, chaque composant Seaside est responsable de son propre rendu. En anglais, nous utilisons le terme *rendering* que nous retrouvons en tant que nom de protocole pour les méthodes de rendu. Commençons notre exploration du rendu par l'exemple du composant "counter".

Le rendu de l'application "counter"

Le rendu de l'application "counter" est relativement simple ; le code est visible sur la figure 12.8. La valeur actuelle du compteur est affichée comme une en-tête (ou *heading*) XHTML et les opérations d'incrémentation et de décrémentation sont implémentées sous la forme de lien HTML (ou *anchor*) avec des *callbacks* (*c-à-d.* des fonctions de rappel) sous la forme de *blocs* qui enverront soit *increase* soit *decrease* à l'objet "counter". Avant de voir plus en détails le protocole *rendering*, regardons un peu le cas de l'application "multicounter"

De la version "counter" à la version "multicounter"

La figure 12.9 montre `WAMultiCounter` qui est aussi une application de compteurs multiples. Elle est autonome ; elle surcharge donc la méthode de classe `canBeRoot` pour qu'elle réponde `true`. De plus, c'est un composant *composite*. Seaside a besoin d'une déclaration de ses composants enfants faite via la méthode `children` qui retourne un tableau de tous les composants que l'application contient. Le rendu de ce composant se fait en faisant le rendu de ces sous-composants séparés par une barre horizontale. En dehors des méthodes d'instance et de classe destinées à l'initialisation, il n'y a rien d'autre pour faire l'application "multicounter" !

Quelques mots encore à propos du rendu XHTML

Comme nous l'avons vu sur ces exemples, Seaside n'utilise pas de patrons ou *templates* pour générer les pages web. Au lieu de cela, il génère le code XHTML de manière programmatique. Tout composant Seaside devrait surcharger la méthode `renderContentOn:`. C'est ce message qui sera envoyé par Seaside pour chaque composant nécessitant d'être rendu. `renderContentOn:` a un argument qui est un canevas HTML (en anglais, *canvas*) sur lequel le composant devra être rendu. Par convention, le paramètre de canevas HTML est appelé `html`. Un canevas HTML est analogue au canevas graphique uti-

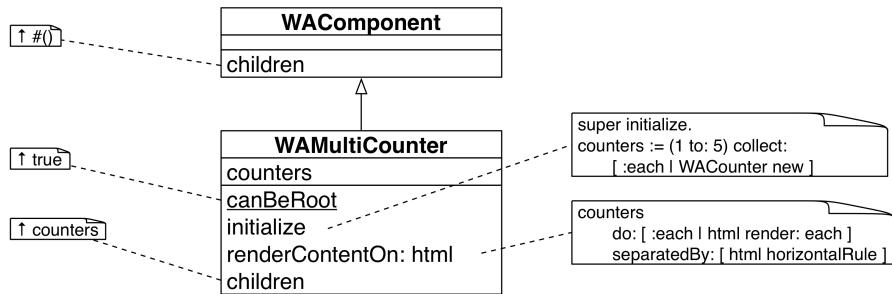


FIGURE 12.9 – L'application de compteurs multiples “multicounter” basée sur la classe WAMultiCounter.

lisé par Morphic (et la plupart des autres bibliothèques graphiques) pour **abstraire le graphisme des détails dépendants des périphériques**.

Voici quelques-unes des méthodes de rendu les plus simples :

```

html text: 'hello world'. "rendu du texte ordinaire"
html html: '&nbsp;'. "rendu d'un élément XHTML"
html render: 1. "rendu d'un object"

```

Le message `render: anyObject` (ou `anyObject` défini un objet quelconque) peut être envoyé à un canevas HTML pour faire le rendu de `anyObject`; ce message est normalement utilisé pour faire le rendu de sous-composants. Le message `renderContentOn:` sera alors envoyé à `anyObject`. C'est ce qu'il se produit avec l'application “multicounter” (voir la figure 12.9).

Utiliser les *brushes*

Un canevas offre un grand nombre de *brushes* (*c-à-d.* des pinceaux) utilisés pour le rendu d'un contenu sur le canevas. Il y a des *brushes* — au singulier, *brush* — pour toutes sortes d'éléments XHTML : des paragraphes, des tables, des listes... Pour voir l'ensemble des *brushes* et des méthodes utilitaires associées, vous devez naviguer avec votre Browser dans la classe `WACanvas` et ses sous-classes. L'argument de `renderContentOn:` est en fait une instance de la sous-classe `WARenderCanvas`.

Nous avons déjà vu le *brush* suivant dans les exemples “counter” et “multicounter” :

```

html horizontalRule.

```

Dans la figure 12.10, nous pouvons voir la sortie de beaucoup de *brushes*



FIGURE 12.10 – La démo des rendus : “Rendering Demo”.

simples inclus dans Seaside⁴. Le composant racine SeasideDemo fait simplement le rendu de ses sous-composants qui sont instances de SeasideHtmlDemo, SeasideFormDemo, SeasideEditCallDemo et SeasideDialogDemo, comme le montre la méthode 12.1.

4. Le code source de méthode 12.1 se trouve dans le paquetage PBE–SeasideDemo dans le projet <http://www.squeaksource.com/PharoByExample>. NdT : Les textes ont été traduits dans le présent ouvrage. Attendez-vous à ce que la version du code d'exemple que vous téléchargerez soit en anglais.

Méthode 12.1 – SeasideDemo»renderContentOn :

```
SeasideDemo»renderContentOn: html
    html heading: 'Rendering Demo'.
    html heading
        level: 2;
        with: 'Rendu de code HTML simple:'.
    html div
        class: 'subcomponent';
        with: htmlDemo.
    "rendu des composants restants ..."
```

Rappelez vous qu'un composant racine doit toujours déclarer ses enfants sinon Seaside refusera d'en faire le rendu.

```
SeasideDemo»children
↑ { htmlDemo . formDemo . editDemo . dialogDemo }
```

Remarquez qu'il y a deux différentes façons d'instancier le *brush* d'en-tête nommé *heading* : soit vous placez le texte directement en argument du message *heading:* envoyé au canevas, soit vous instanciez le *brush* via l'envoi de *heading* puis vous envoyez une cascade de messages au *brush* en question pour définir ses propriétés et en faire le rendu. Beaucoup des *brushes* disponibles peuvent être utilisés de ces deux manières.

Si vous envoyez à un *brush* une cascade de messages incluant *with:*, *with:* doit être le message *final*.

L'association du contenu au *brush* et le rendu de ce dernier est fait par *with:*.

Dans méthode 12.1, la première en-tête est de niveau (ou *level*) 1 puisque c'est la valeur par défaut. Nous mettons explicitement le niveau de la seconde en-tête à 2. Le sous-composant est rendu sous la forme de *div XHTML* avec "subcomponent" comme nom de classe CSS (pour en savoir plus sur les feuilles de style CSS, rendez-vous à la section 12.5). Notez aussi que l'argument du message à mots-clés *with:* n'a pas besoin d'être une chaîne de caractères littéral : ce peut être un autre composant ou même — comme dans l'exemple suivant — un bloc contenant des actions différées de rendu.

Le composant SeasideHtmlDemo fait la démonstration des *brushes* les plus basiques. Le code devrait parler de lui-même⁵.

```
SeasideHtmlDemo»renderContentOn: html
    self renderParagraphsOn: html.
    self renderListsAndTablesOn: html.
```

5. NdT : les *divs* et les *spans* sont des éléments HTMLXHTML et "link with callback" peut se traduire par "lien avec fonction de rappel".

```
self renderDivsAndSpansOn: html.  
self renderLinkWithCallbackOn: html
```

La division d'une longue méthode de rendu en plusieurs méthodes adjointes est une pratique commune : c'est ce que nous avons fait ici.

Ne mettez pas tout votre code de rendu dans une seule méthode. Séparez-le dans différentes méthodes adjointes dont le nom est de la forme `renderOn:`. Toutes les méthodes de rendu vont dans le protocole *rendering*. N'envoyez pas `renderContentOn:` depuis votre propre code : utilisez plutôt `render:`.

Observons le code suivant. La première méthode, `SeasideHtmlDemo»renderParagraphsOn:`, vous montre comment générer des paragraphes XHTML, du texte ordinaire ou en emphase (*c-à-d.* en italique) et des images. Remarquez qu'en Seaside les éléments simples sont rendus en spécifiant le texte qu'ils contiennent directement alors que les éléments complexes sont spécifiés dans des blocs. C'est une convention simple pour vous aider à structurer votre code de rendu.

```
SeasideHtmlDemo»renderParagraphsOn: html  
html paragraph: 'Un paragraphe en texte ordinaire.'  
html paragraph: [  
    html  
        text: 'Un paragraphe en texte ordinaire suivi par une ligne de séparation.';  
        break;  
        emphasis: 'Texte en emphase';  
        text: 'suivi d''une barre horizontale.';  
        horizontalRule;  
        text: 'Une URI d''image:'.  
    html image  
        url: self pharolImageUrl;  
        heigh: '40']
```

La méthode suivante, `SeasideHtmlDemo»renderListsAndTablesOn:`, vous montre comment générer des listes et des tables. Une table utilise deux niveaux de blocs pour afficher chacune de ses lignes et les cellules que ces dernières contiennent.

```
SeasideHtmlDemo»renderListsAndTablesOn: html  
html orderedList: [  
    html listItem: 'Un élément de liste ordonnée'.  
html unorderedList: [  
    html listItem: 'Un élément de liste non-ordonnée'.  
html table: [
```

```
html tableRow: [
    html tableData: 'Une table avec une cellule.']]
```

Nous pouvons voir sur l'exemple suivant comment nous pouvons spécifier les éléments *divs* et *spans* avec leurs attributs CSS *class* ou *id*. Bien sûr, les messages *class:* et *id:* peuvent être aussi envoyés à d'autres *brushes*, et pas seulement aux *divs* et aux *spans*. La méthode *SeasideDemoWidget»style* définit comment ces éléments XHTML devraient être affichés (voir la section 12.5).

```
SeasideHtmlDemo»renderDivsAndSpansOn: html
html div
    id: 'author';
    with: [
        html text: 'Texte brut dans un élément div avec l"id "author".'.
        html span
            class: 'highlight';
            with: 'Un élément span de classe "highlight".']
```

Finalement nous avons un simple exemple d'un lien, créé en liant un simple *callback* (ou fonction de rappel) à une "ancre" (en anglais, "anchor"). Cliquez sur le lien aura pour conséquence de basculer le texte entre "true" et "false" en faisant varier la variable d'instance booléenne *toggleValue*.

```
SeasideHtmlDemo»renderLinkWithCallbackOn: html
html paragraph: [
    html text: 'Un lien avec une action locale: '.
    html span with: [
        html anchor
            callback: [toggleValue := toggleValue not];
            with: 'change la valeur du booléen:'.
    html space.
    html span
        class: 'booléen';
        with: toggleValue ]]
```

Remarquez que les actions ne devraient apparaître que dans les *callbacks*. Le code exécuté durant le rendu ne devrait pas changer l'état de l'application !

Les formulaires

Les formulaires ou *forms* sont simplement rendus comme dans les autres exemples que nous avons vus précédemment. Voici le code pour le composant *SeasideFormDemo* sur la figure 12.10.

```
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' ]
```

En raison de sa nature complexe, le formulaire est rendu en utilisant un bloc. Notez que tous les changements d'état se produisent dans les *callbacks* et non dans les parties de code liées au rendu.

Le message `on:of:` est une particularité Seaside qui mérite une explication. Dans l'exemple, ce message est utilisé pour attacher un zone de saisie de texte à la variable `heading`. Les ancrages (*c-à-d.* les liens) et les boutons prennent en charge, eux aussi, ce message. Le premier argument est le nom d'une variable d'instance pour laquelle des méthodes d'accès ont été définies ; le second argument est l'objet auquel appartient cette variable d'instance. Les messages accesseur et mutateur avec la convention de noms usuelle (*c-à-d.* `heading` et `heading:` respectivement) doivent être compris par l'objet. Dans le cas présent d'une zone de saisie de texte, cela nous évite le problème d'avoir à définir un *callback* pour mettre à jour le champ de saisie et d'avoir à attacher le contenu par défaut de l'entrée à la valeur actuelle de la variable d'instance. En utilisant `on: #heading of: self`, la variable `heading` est mise à jour automatiquement à chaque fois que l'utilisateur change le texte de la zone de saisie.

Le même message est utilisé deux fois encore dans cet exemple pour mettre à jour la variable `color` en fonction de la sélection de la couleur dans le formulaire HTML et pour attacher l'état de la case à cocher (ou `checkbox`) à la variable `checked`. Vous pouvez trouver beaucoup d'autres exemples dans les tests fonctionnels de Seaside. Jetez un coup d'œil à la [catégorie Seaside-Tests-Functional](#) ou allez simplement sur la page <http://localhost:8080/seaside/tests/>

alltests avec votre navigateur web. Sélectionnez `WAInputTest` (cliquez sur le bouton `Restart` si vous avez désactivé le support JavaScript de votre navigateur web) pour voir la plupart des types d'éléments de formulaire.

N'oubliez pas que, si vous activez le bouton `Toggle Halos` de la barre d'outils, vous pouvez naviguer dans le code source des exemples directement via le navigateur de classes Seaside.

12.5 Les feuilles de style CSS

Les feuilles de style en cascade ou *Cascading Style Sheets*⁶ abrégé en CSS sont devenues une technique standard pour séparer le style du contenu des applications web. Seaside utilise les CSS pour éviter l'encombrement de votre code de rendu par la mise en page.

Vous pouvez mettre une feuille de style CSS pour vos composants web en définissant la méthode `style` qui devrait retourner les règles CSS de ce composant sous forme d'une chaîne de caractères. Les styles de tous les composants affichés sur un page web sont assemblés. Ainsi chaque composant peut avoir son propre style. Une meilleure approche serait de définir une classe abstraite pour votre application web définissant un style commun pour toutes ses sous-classes.

En réalité, il est préférable de définir les feuilles de style des applications déployées comme des fichiers externes. Ce faisant, l'apparence du composant est complètement séparée de sa fonctionnalité — regardez la classe `WAFileLibrary` qui permet de servir des fichiers statiques sans le besoin d'un autre serveur autonome dédié à ces fichiers.

Si vous n'êtes pas déjà familier avec les CSS, veuillez lire une brève introduction aux feuilles de style CSS.

En utilisant les CSS, vous définirez différentes classes aux éléments de texte et de paragraphe de vos pages web et vous déclarerez toutes les propriétés graphiques dans une feuille de style séparée, plutôt que d'écrire ces propriétés directement sous forme d'attributs de ces éléments. Les entités-paragraphes sont appelées *divs* et les entités-textes sont appelées *spans*. Vous devriez définir alors définir des noms symboliques pour ces classes, tels que "highlight" (*subbrillance* en français) pour le texte à mettre en surbrillance, et préciser dans votre feuille de style comment le texte en surbrillance doit être affiché. Une feuille de style comprend simplement un ensemble de règles qui décrivent le format des éléments XHTML donnés. Chaque règle se scinde en deux parties : un *sélecteur* qui précise sur quels éléments XHTML la règle s'applique et une *déclaration* qui liste un certain nombre d'attributs pour ce ou ces éléments.

6. <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 – La feuille de style commune SeasideDemoWidget.

la figure 12.11 illustre l'exemple d'une simple feuille de style pour l'application "Rendering Demo" vue plus tôt dans la figure 12.10. La première règle signale une préférence pour les fontes à utiliser pour le corps de la page web correspondant toujours à l'élément body. Les quelques règles suivantes donnent les propriétés des en-têtes de niveau 2 (h2) et celles des tables (table) et de leur cellule (td pour *table data*).

Les règles restantes ont des sélecteurs qui correspondent aux éléments XHTML qui ont le même nom d'attributs "class" ou "id". Les sélecteurs CSS pour les attributs de classe nommés "class" commencent par un point (".") et ceux pour les attributs *id* commencent par un dièse ("#"). La principale différence entre les classes et les attributs *id* est la suivante : plusieurs éléments peuvent avoir la même classe mais seul un élément peut avoir un attribut *id* donné (c-à-d. un identifiant). Ainsi, alors qu'une classe comme highlight

peut être utilisée plusieurs fois dans un page, un attribut *id* doit identifier un élément *unique* sur la page, comme un menu particulier par exemple, ou encore une date modifiée ou un auteur. Remarquez qu'un élément XHTML particulier peut avoir de multiples classes ; dans ce cas, tous les attributs d'affichage seront appliqués séquentiellement.

Des conditions au sélecteur peuvent être ajoutées. Ainsi le sélecteur `div subcomponent` correspondra seulement à un élément XHTML qui est à la fois un élément *div* et un élément de classe "subcomponent".

Il est aussi possible de préciser des éléments imbriqués, bien que cela soit rarement nécessaire. Par exemple, le sélecteur "paragraphe span" correspondra à un élément *span* inclus dans un paragraphe (élément *p*) mais exclura ceux inclus dans un élément *div*.

Il existe un grand nombre de livres et de sites web sur le sujet des CSS que vous pourrez consulter pour en apprendre plus. Pour une démonstration spectaculaire du pouvoir des CSS, nous vous recommandons de voir le site CSS Zen Garden⁷ qui présente comment un même contenu peut avoir un rendu totalement différent uniquement en changeant de feuille de style.

12.6 Gérer les flux de contrôle

Seaside facilite considérablement l'élaboration d'applications web ayant un flux de contrôle (en anglais, *control flow*) complexe. Deux mécanismes peuvent être utilisés :

1. un composant peut appeler — en anglais, *call* — un autre composant en envoyant `caller call: callee` où `caller` est le composant appelant et `callee` est le composant appelé. Le composant appelant est temporairement remplacé par le composant appelé jusqu'à ce que le composant appelé lui rende la main en envoyant `answer: (c-à-d. la réponse)`. L'appelant est généralement `self` mais ce peut être un autre composant visible ;
2. une modélisation des tâches (ou *workflow*) peut être définie sous forme de *task s*⁸. C'est un type particulier de composant, sous-classe de `WATask` (et non pas de `WAComponent`). Au lieu de définir `renderContentOn:`, ce composant ne définit aucun contenu de lui-même mais définit une méthode `go` qui envoie une série de messages `call:` pour activer un certain nombre de composants en retour.

7. <http://www.csszengarden.com/>

8. En français, *tâche*.

Call et answer

Call (c-à-d. l'appel) et *answer* (c-à-d. la réponse) sont utilisés pour réaliser des dialogues simples.

L'application “Rendering Demo” présente un exemple simple de *call:* et *answer:*. Le composant SeasideEditCallDemo affiche une zone de saisie de texte et un lien *edit*. Le *callback* pour ce lien appelle (en anglais, *call*) une nouvelle instance de SeasideEditAnswerDemo initialisée à la valeur du texte dans la zone de saisie. Le *callback* met aussi à jour cette zone de saisie au résultat qui est envoyé en réponse (en anglais, *answer*)—dans les codes suivants, nous avons souligné les messages *call:* et *answer:* pour plus de clarté.

```
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'
```

Le code ne fait absolument aucune référence à la nouvelle page web qui doit être créée, ce qui est particulièrement élégant. À l'exécution, une nouvelle page est créée dans laquelle le composant SeasideEditCallDemo est remplacé par un composant SeasideEditAnswerDemo ; le composant parent et les autres et les autres composants homologues restent inchangés.

Les messages *call:* et *answer:* ne doivent jamais être utilisés durant la phase de rendu. Ils peuvent être envoyés sans problème depuis un *callback* ou depuis la méthode *go* d'une *task*.

Le composant SeasideEditAnswerDemo est aussi remarquablement simple. Il se contente d'afficher un formulaire avec une zone de saisie. Le bouton *submit* (*brush* accessible par le message *submitButton*) est lié au *callback* qui répondra la valeur finale du texte dans la zone de saisie.

```
SeasideEditAnswerDemo»renderContentOn: html
html form: [
  html textInput
    on: #text of: self.
  html submitButton
    callback: [ self answer: self text ];
    text: 'ok'.
]
```

C'est tout !

Seaside prend soin du flux de contrôle et du rendu de tous les composants. Il est intéressant de voir que le bouton “retour arrière” du navigateur web fonctionnera bien (bien que les effets de bord ne soient pas rembobinés à moins d'avoir pris des dispositions complémentaires).

Les méthodes utilitaires

Puisque certains dialogues de type *call-answer* sont très communs, Seaside fournit des méthodes utilitaires pour vous éviter l'écriture des composants tels que `SeasideEditAnswerDemo`. Les dialogues générés sont montrés sur la figure 12.12. Nous pouvons voir que ces méthodes utilitaires sont utilisées dans la méthode `SeasideDialogDemo»renderContentOn:`:

Le message `request:` fait un appel à un composant qui vous laissera éditer une zone de saisie. Le composant répond la chaîne de caractères éditée dans cette zone de saisie. Un *label* optionnel et une valeur par défaut peuvent aussi être spécifiés.

```
SeasideDialogDemo»renderContentOn: html
    html anchor
        callback: [ self request: 'edit this' label: 'done' default: 'some text' ];
        with: 'self request:'.
    ...

```

Le message `inform:` appelle un composant qui affiche simplement l'argument-message et attend que l'utilisateur clique sur “ok”. Le composant appelé retourne simplement `self`.

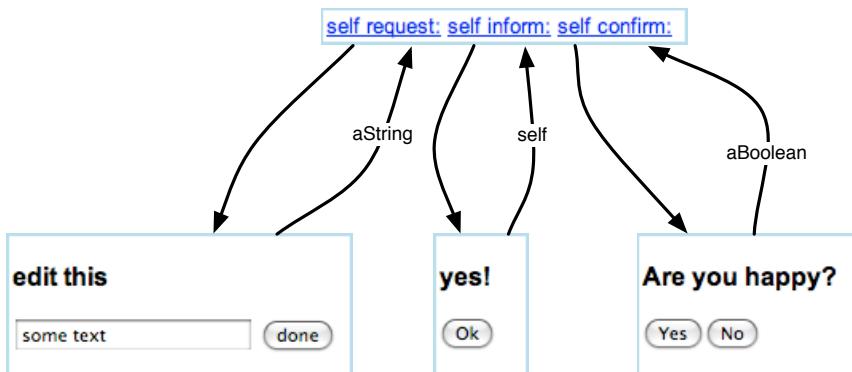


FIGURE 12.12 – Certains dialogues standards.

```
...
html space.
html anchor
callback: [ self inform: 'yes' ];
with: 'self inform'.
...

```

Le message `confirm:` pose une question et attend que l'utilisateur clique sur "Yes" ou "No". Le composant répond un booléen qui peut être utilisé pour faire d'autres actions.

```
...
html space.
html anchor
callback: [
    (self confirm: 'Are you happy?') "Êtes-vous content?"
    ifTrue: [ self inform: ':-)' ]
    ifFalse: [ self inform: ':-(' ]
];
with: 'self confirm'.

```

Quelques méthodes utilitaires, tels que `chooseFromCaption:`, sont définies dans le protocole *convenience* de la classe `WAComponent`.

Les tasks Seaside

Une task (ou *tâche*) est un composant qui est sous-classe de `WATask`. Elle n'effectue aucun rendu elle-même mais appelle simplement d'autres composants dans un flux de contrôle défini en implémentant la méthode `go`.

`WAConvenienceTest` est un simple exemple de l'utilisation d'une *task* définie dans la catégorie *Seaside-Tests-Functional*. Pour voir son effet, pointez votre navigateur web sur l'URL <http://localhost:8080/seaside/tests/alltests> et sélectionnez `WAConvenienceTest` et cliquez sur `Restart`.

```
WAConvenienceTest»go
[ self chooseCheese.
  self confirmCheese ] whileFalse.
self informCheese
```

Cette *task* appelle trois composants. Le premier, généré par la méthode utilitaire `chooseFromCaption:`, est une instance de `WAChoiceDialog` qui demande à l'utilisateur de choisir un fromage (*cheese*).

```
WAConvenienceTest»chooseCheese
cheese := self
chooseFrom: #('Greyerzer' 'Tilsiter' 'Sbrinz')
```

```
caption: 'What"s your favorite Cheese?'. "Quel est votre fromage préféré"
cheese isNil ifTrue: [ self chooseCheese ]
```

Le second est un dialogue WAYesOrNoDialog pour confirmer le choix (généré par la méthode utilitaire confirm:).

```
WAConvenienceTest»confirmCheese
"Ce fromage est-il votre préféré?"
↑self confirm: 'Is ', cheese, ' your favorite cheese?'
```

In fine, un dialogue WAFormDialog est appelé (via la méthode utilitaire inform:).

```
WAConvenienceTest»informCheese
"Votre fromage préféré est ..."
self inform: 'Your favorite cheese is ', cheese, ''
```

Vous pouvez voir ces dialogues générés sur la figure 12.13.

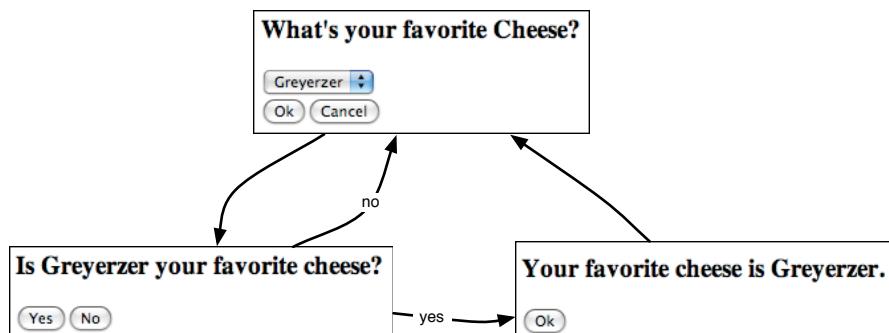


FIGURE 12.13 – Un simple exemple de task en action.

Les transactions

Nous avons vu dans la section 12.3 que Seaside peut garder une trace de la correspondance entre l'état des composants et les pages web en ayant des composants qui enregistrent leur état pour le *backtracking* : tout ce qu'un composant a besoin de faire, c'est d'implémenter la méthode states pour retourner un tableau de tous les objets dont l'état doit être suivi.

Cependant, certaines fois, nous ne voudrions pas avoir de *backtracking* de l'état : nous aimerais seulement *prévenir* l'utilisateur qu'il risque d'effacer accidentellement des effets qui devraient être permanents. Ce problème est souvent appelé “le problème du panier” (en anglais, “the shopping cart

problem"). Une fois que vous avez validé votre panier virtuel sur un site webmarchand et que vous avez payé pour les articles que vous avez acheté, il ne devrait pas être possible de revenir en arrière avec le bouton "retour arrière" de votre navigateur web et d'ajouter de nouveaux articles à votre panier !

Seaside vous permet de prévenir cela en définissant une *task* dans laquelle certaines actions sont regroupées en *transactions*. Vous pouvez utiliser le *backtracking* dans une *transaction* mais, une fois la *transaction* terminée, vous ne pouvez plus revenir sur celle-ci. Les pages correspondantes sont *invalidées* et toute tentative de revenir sur celles-ci entraînera la création d'une alerte par Seaside et redirigera l'utilisateur sur la page la plus récemment valide.

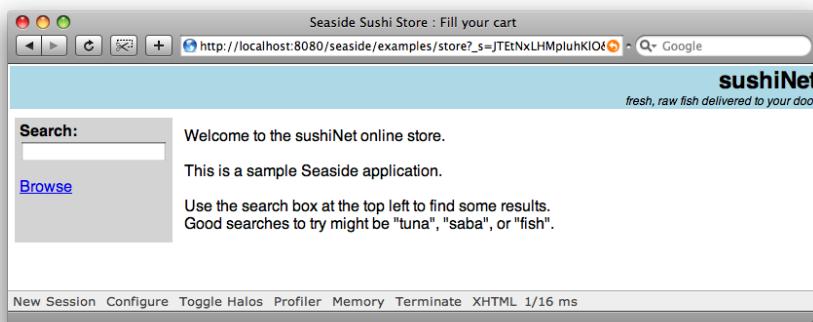


FIGURE 12.14 – L'application “Sushi Store”.

L'application “*Sushi Store*” est une application de démonstration illustrant de nombreuses particularités de Seaside dont les *transactions*. Cette application est incluse dans votre installation de Seaside. Dès lors vous pouvez l'essayer en vous rendant sur la page <http://localhost:8080/seaside/examples/store>⁹ avec votre navigateur web.

L'application “*Sushi Store*” (traduisible par le *restaurant de sushis en ligne*) présente la modélisation des tâches suivante :

1. Visiter la boutique.
2. Naviguer ou chercher un sushi.
3. Ajouter un sushi à votre panier (ou *shopping cart*).
4. Valider (*checkout*).

9. Si vous ne la trouvez pas dans votre image, vous pouvez charger une version de l'application “*Sushi Store*” sur SqueakSource dans le dépôt <http://www.squeaksource.com/SeasideExamples/> sous le nom de paquetage *Store*.

5. Vérifier votre ordre.
6. Entrer une adresse d'expédition.
7. Vérifier une adresse d'expédition.
8. Entrer les informations de paiement.
9. Votre poisson cru est en route !

Si vous activez les halos, vous verrez que le composant racine de l'application "Sushi Store" est un instance de `WAStore`. Ce composant ne fait rien d'autre que le rendu d'une barre de titre et de l'objet `task`; ce dernier est une variable d'instance de `WAStoreTask`.

```
WAStore»renderContentOn: html
  "... rendu de la barre de titre ..."
  html div id: 'body'; with: task
```

`WAStoreTask` retient cette séquence de modélisation de tâches. C'est important qu'à deux moments l'utilisateur ne puisse pas revenir en arrière et changer les informations déjà envoyées.

 Achetez avec la commande "Checkout" des sushis, confirmez en cliquant "Proceed with checkout" et utilisez ensuite le bouton "retour arrière" pour essayer de mettre plus de sushis dans votre panier (dans l'application, "Your cart"). Vous aurez le message "That page has expired" (c-à-d. "la page a expiré").

Seaside laisse le programmeur écrire si une certaine partie d'une séquence de tâches agit comme une *transaction* : une fois la *transaction* complète, l'utilisateur ne pourra plus revenir en arrière et l'annuler. Vous l'écrivez en envoyant le message `isolate:` à la `task` avec le bloc transactionnel comme argument. Nous pouvons le voir dans la séquence des tâches de l'application "Sushi Store" dans le code suivant :

```
WAStoreTask»go
| shipping billing creditCard |
cart := WAStoreCart new.
self isolate:
  [[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.
```

Nous voyons assez clairement que nous avons ici deux *transactions*. La première remplit le panier (en anglais, *fill cart*) et clôt la phase des achats (les méthodes adjointes, *fillCart* etc, prennent soin d'instancier et d'appeler les bons sous-composants). Une fois que vous avez confirmé le contenu du panier, vous ne pouvez plus revenir sans démarrer une nouvelle session. La seconde *transaction* concerne la saisie des informations d'envoi et de paiement. Vous pouvez naviguer en arrière et revenir dans la seconde *transaction* jusqu'à votre confirmation du paiement. Cependant, une fois les deux *transactions* concluent, toute tentative de navigation en arrière échouera.

Les *transactions* peuvent aussi être imbriquées. Une simple démonstration de ceci se trouve dans la classe `WANestedTransaction`. Le premier message `isolate:` prend un bloc comme argument ; celui-ci a un autre envoi de message `isolate:` imbriqué.

```
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'
```

 Allez sur <http://localhost:8080/seaside/tests/alltests>, sélectionnez `WATransactionTest` et cliquez sur `Restart`. Essayez de naviguer en arrière et en avant dans la transaction parent et enfant (child) en cliquant sur le bouton "retour arrière" et en cliquant ensuite sur le bouton `ok`. Remarquez que, dès qu'une transaction est complète, vous ne pouvez plus revenir dans la transaction, sans générer une erreur en cliquant sur `ok`.

12.7 Un tutoriel complet

Voyons comment nous pouvons construire une application Seaside complète de zéro¹⁰. Nous allons construire une calculatrice RPN¹¹ comme une application Seaside qui utilise une machine à pile simple comme modèle

10. L'exercice devrait vous prendre deux bonnes heures. Si vous préférez simplement regarder le code source final, vous pouvez le récupérer depuis le projet SqueakSource à l'adresse : <http://www.squeaksources.com/PharoByExample>. *PBE-SeasideRPN* est le paquetage à charger. Le tutoriel utilise des noms de classe légèrement différents ainsi vous pourrez comparer votre implémentation avec la notre.

11. La RPN pour "Reverse Polish Notation" ou notation polonaise inversée est une des trois écritures mathématiques ; elle est dite aussi *postfixe*.

(“pile” en anglais se disant “stack”). En outre, l’interface Seaside nous laissera choisir entre deux modes d’affichage — l’un nous montrant simplement la valeur actuelle au sommet de la pile et l’autre nous montrant l’état complet de la pile. Vous pouvez voir la calculatrice munie de ses deux options d’affichage sur la figure 12.15.

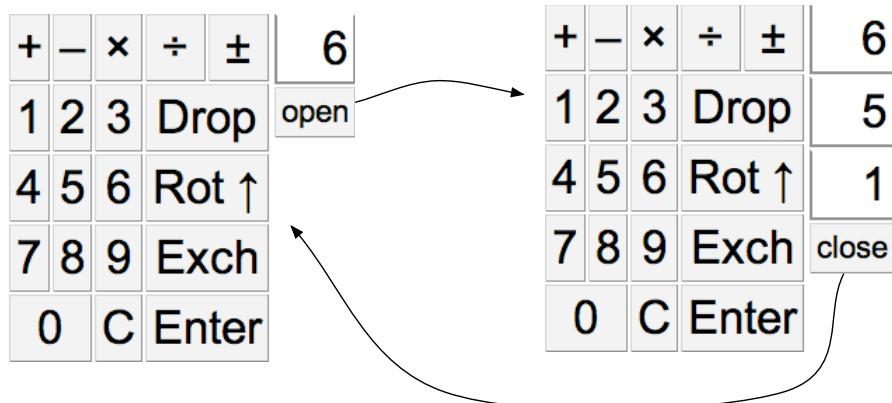


FIGURE 12.15 – La calculatrice RPN et sa machine à pile.

Nous commençons par implémenter la machine à piles et ses tests.

❶ Définissez une nouvelle classe `MyStackMachine` *dans une nouvelle catégorie de votre choix et* avec une variable d’instance `contents` initialisée comme une nouvelle collection ordonnée, instance de `OrderedCollection`.

```
MyStackMachine»initialize
    super initialize.
    contents := OrderedCollection new.
```

La machine à pile devrait fournir des opérateurs `push:` et `pop` pour ajouter et enlever des valeurs respectivement ainsi que des commandes pour voir le sommet de la pile que nous appellerons `top` et pour faire plusieurs opérations arithmétiques telles que l’addition (`add`), la soustraction (`subtract`), la multiplication (`multiply`) et la division (`divide`) des valeurs au sommet de la pile.

❷ Écrivons des tests pour les opérations de la pile et implémentons ensuite ces opérations. Voici un exemple de test (avec la classe `MyStackMachineTest`, instance de `TestCase`, pour le cas de la division `div` en ayant pris soin d’initialiser la variable d’instance `stack` dans `setUp`) :

```
MyStackMachineTest»testDiv
stack
  push: 3;
  push: 4;
  div.
self assert: stack size = 1.
self assert: stack top = (4/3).
```

Vous devriez penser à utiliser des méthodes utilitaires pour les opérations arithmétiques pour vérifier qu'il y a toujours deux nombres sur la pile avant de faire quoi que ce soit et lever une erreur si ce prérequis n'est pas atteint¹². Si vous faites ainsi, la plupart de vos méthodes tiendront sur une ou deux lignes.

Vous pourriez aussi considérer l'implémentation d'une méthode MyStackMachine»printOn: pour faciliter le débogage de votre implémentation de la machine à pile avec l'aide de l'Inspector (une petite astuce : déléguer simplement l'impression printOn: à la variable d'instance contents).

 *Complétez la classe MyStackMachine en écrivant les opérateurs dup qui duplique la valeur du sommet de la pile et la met dans la pile, exch (abrégé de exchange) qui échange les deux valeurs du sommet de la pile et rotUp qui fait une permutation circulaire du contenu entier de la pile — la valeur en top se retrouvera ainsi en bas de la pile.*

Nous avons maintenant une implémentation simple d'une machine à pile. Nous pouvons commencer la programmation de notre calculatrice RPN Seaside.

Nous allons créer 5 classes :

- MyRPNWidget — c'est une classe que nous voulons abstraite et qui définit la feuille de style CSS commune pour l'application et d'autres comportements communs pour les composants de la calculatrice RPN. C'est une sous-classe de WAComponent et super-classe directe des quatre classes suivantes ;
- MyCalculator — c'est le composant racine. Il devrait enregistrer l'application dans Seaside (côté classe), instancier et faire le rendu de ses sous-composants et il devrait enfin déclarer un état pour le *backtracking* ;
- MyKeypad — ce composant affiche les boutons (*key*) que nous utiliserons pour interagir avec la calculatrice ;
- MyDisplay — ce composant affiche le sommet de la pile et fournit un bouton pour appeler un autre composant pour afficher la vue détaillée (*c-à-d.* pour gérer un *call*) ;

12. C'est une bonne idée d'utiliser Object»assert: dans MyStackMachine pour écrire les préconditions requises pour une opération. Cette méthode levera un AssertionFailure si l'utilisateur essaie d'utiliser la machine à pile dans un état invalide.

- MyDisplayStack — ce composant affiche la vue détaillée de la pile et offre un bouton pour répondre en retour (c-à-d. pour gérer un *answer*). C'est une sous-classe de MyDisplay.

 *Définissez la classe MyRPNWidget dans la catégorie MyCalculator. Définissez le style commun pour l'application.*

Voici une feuille de style CSS minimale pour l'application. Vous pourrez à loisir la rendre plus chic si vous voulez.

```
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; }'
```

 *Définissez la classe MyCalculator de façon à être un composant racine et enregistrez le en tant qu'application Seaside (autrement dit, implémentez canBeRoot et la méthode de classe initialize). Commencez à implémenter la méthode MyCalculator»renderContentOn: pour faire le rendu de quelque chose de simple comme, par exemple, afficher le nom de l'application et vérifiez que l'application tourne correctement dans un navigateur web.*

MyCalculator est responsable de l'instanciation des classes MyStackMachine, MyKeypad et MyDisplay.

 *Définissez MyKeypad et MyDisplay comme sous-classes de MyRPNWidget. Tous ces trois composants auront besoin d'accéder à une instance commune de la machine à pile; définissez donc la variable d'instance stackMachine et une méthode d'initialisation setMyStackMachine: dans la classe mère MyRPNWidget. Ajoutez les variables d'instance keypad et display à la classe MyCalculator et initialisez-les dans MyCalculator»initialize (sans oublier d'envoyer super initialize).*

 *Passez l'instance partagée de la machine à pile à l'objet keypad et à l'objet*

display dans la même méthode MyCalculator»initialize. Implémentez MyCalculator »renderContentOn: de façon à faire le rendu des instances keypad et display. Pour afficher correctement les sous-composants, vous devez coder la méthode MyCalculator»children de manière à retourner un tableau avec keypad et display. Implémentez une méthode de rendu quelconque, par exemple pour afficher un texte ordinaire, pour le keypad et le display et vérifiez que l'application calculatrice affiche ces deux sous-composants.

Maintenant nous allons changer l'implémentation du display pour afficher la valeur du sommet de la pile.

 Utilisez une table HTML avec la classe "keypad" contenant une ligne avec une seule cellule de classe "stackcell" dans MyDisplay»renderContentOn:. Changez maintenant la méthode de rendu de keypad pour que le nombre 0 soit mis sur la pile dans le cas où celle-ci est vide (définissez et utilisez MyKeypad»ensureMyStackMachineNotEmpty). Faites en sorte que keypad affiche une table vide de classe "keypad". Le calculateur devrait afficher une seule cellule contenant la valeur 0. Si vous activez les halos, vous devriez voir quelque chose comme suit :

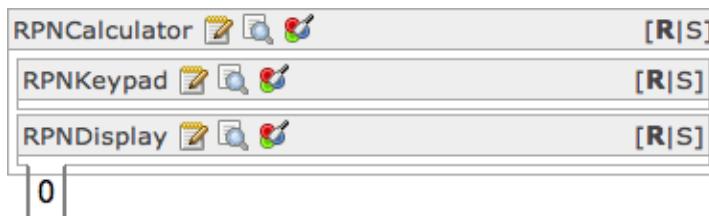


FIGURE 12.16 – Affichage du sommet de la pile.

Implémentons désormais une interface pour interagir avec la pile.

 Définissez tout d'abord les méthodes adjointes facilitant l'écriture de l'interface :

```
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
```

Nous utiliserons ces deux méthodes pour définir les boutons du keypad avec les *callbacks* appropriés. Certains boutons peuvent s'étaler sur plusieurs colonnes (avec `colSpan:`) mais, par défaut, ils occupent une seule colonne.

 Utiliser ces deux méthodes pour écrire le keypad comme suit (une astuce : commencez par faire en sorte que les chiffres et du bouton "Enter" fonctionnent ([en laissant la méthode `setClearMode`](#)) avant de vous attarder aux opérateurs arithmétiques) :

```
MyKeypad»renderContentOn: html
self ensureStackMachineNotEmpty.
html table
  class: 'keypad';
  with: [
    html TableRow: [
      self renderStackButton: '+' callback: [self stackOp: #add] on: html.
      self renderStackButton: '-' callback: [self stackOp: #min] on: html.
      self renderStackButton: '*' callback: [self stackOp: #mul] on: html.
      self renderStackButton: '/' callback: [self stackOp: #div] on: html.
      self renderStackButton: '+' 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].
    " et ainsi de suite ... "
    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]]
```

Vérifiez que le keypad s'affiche proprement. Si vous essayez de cliquer sur les boutons, vous verrez que la calculatrice ne marche pas encore...

 Implémentez `MyKeypad»type:` pour mettre à jour le sommet de la pile en annexant les chiffres saisis. Vous aurez besoin de convertir la valeur du sommet de la pile en chaîne de caractères, la mettre à jour en la concaténant avec l'argument de `type:` et la convertir enfin en entier, en faisant de la sorte :

```
MyKeypad»type: aString
stackMachine push: (stackMachine pop asString, aString) asNumber.
```

Lorsque vous cliquez sur les boutons-chiffre, l'affichage devrait être mis à jour (soyez donc sûr que la méthode MyStackMachine»pop retourne la valeur sortie de la pile, sinon ça ne marchera pas !).

 Now we must implement MyKeypad»stackOp: Something like this will do the trick :

```
MyKeypad»stackOp: op
[ stackMachine perform: op ] on: AssertionFailure do: [ ].
```

Nous ne sommes pas sûrs du succès de toutes les opérations. Par exemple, une addition peut échouer si nous n'avons pas deux nombres sur la pile. Pour l'instant, nous pouvons simplement ignorer de telles erreurs. Si nous nous sentons plus ambitieux plus tard, nous pourrons ajouter un retour informatif à l'utilisateur dans un bloc de gestion d'erreur.

 La première version de la calculatrice devrait fonctionner maintenant. Essayez d'entrer des nombres en cliquant sur les boutons-chiffre, puis cliquez sur Enter pour dupliquer la valeur actuelle, et cliquez enfin sur + pour faire l'addition de ces deux valeurs.

Vous remarquerez que la saisie des chiffres ne se passent pas comme prévu. En fait, la calculatrice devrait savoir si vous saisissez un nouveau nombre ou que vous complétez une nombre existant.

 Adaptez la méthode MyKeypad»type: pour se comporter différemment suivant le mode actuel de saisie. Introduisez une variable d'instance mode pouvant prendre trois valeurs #typing (lorsque vous saisissez), #push (après que vous ayez effectué une opération de calcul et que la saisie devrait forcer l'entrée de la valeur sur la pile) ou #clear (après que vous ayez cliqué sur le bouton Enter et que le sommet de la pile devrait se mettre dans l'état initial avant la prochaine saisie). La nouvelle méthode type: pourrait ressembler à ceci :

À CORRIGER! MARTIAL: POUR TODO-CB, TYPE : DEVRAIT UTILISER "STACKMACHINE DUP." AU LIEU DE "STACKMACHINE PUSH : STACKMACHINE TOP." ; J'AI DÉJÀ CHANGÉ EN PRÉVISION...

```
MyKeypad»type: aString
self inPushMode ifTrue: [
    stackMachine dup.
    self stackClearTop].
self inClearMode ifTrue: [ self stackClearTop ].
stackMachine push: (stackMachine pop asString, aString) asNumber.
```

La saisie devrait mieux fonctionner maintenant mais le fait de ne pouvoir voir la pile intégralement est frustrant.

❶ Définissez la classe `MyDisplayStack` comme une sous-classe de `MyDisplay`. Ajoutez un bouton dans la méthode de rendu de `MyDisplay` qui appellera une nouvelle instance de `MyDisplayStack`. Vous aurez besoin d'un lien HTML (ou anchor) ressemblant à ceci :

```
html anchor
callback: [ self call: (MyDisplayStack new setStackMachine: stackMachine)];
with: 'open'
```

Le `callback` entraînera le remplacement temporaire de l'actuelle instance de `MyDisplay` en une nouvelle instance de la classe `MyDisplayStack` dont le travail consiste à afficher la pile entière. Lorsque ce composant signale que le travail est fini (en envoyant `self answer`), l'instance originel de `MyDisplay` reviendra dans le [flux](#).

❷ Définissez la méthode de rendu de la classe `MyDisplayStack` pour qu'elle affiche toutes les valeurs sur la pile (vous aurez besoin de définir soit un accesseur `contents` pour atteindre le contenu de la machine à pile, soit une méthode `MyStackMachine»do:` pour itérer sur les valeurs de la pile). L'interface de la pile devra aussi proposer un bouton "close" dont le callback effectuera simplement en `self answer`.

```
html anchor
callback: [ self answer];
with: 'close'
```

Vous devriez maintenant être capable d'ouvrir (avec `open`) et de fermer (avec `close`) l'interface `display` de la pile durant l'utilisation de la calculatrice.

Il y a cependant une chose que nous avons oubliée. Essayez d'effectuer des opérations sur la pile. Utilisez maintenant le bouton "retour arrière" de votre navigateur web et essayez encore d'effectuer des opérations sur la pile (par ex., ouvrez la pile, saisissez `1` une fois et `Enter` deux fois puis `+`). La pile devrait afficher "2" et "1". Cliquez maintenant sur la bouton "retour arrière". La pile montre encore trois fois "1". Mais si vous cliquez sur `+`, la pile affichera "3" au lieu de "2". Le `backtracking` ne marche pas encore).

❸ Codez la méthode `MyCalculator»states` pour retourner un tableau contenant le contenu `contents` de la machine de pile. Vérifiez que le `backtracking` fonctionne correctement désormais !

Détendez-vous et prenez un rafraîchissant : vous l'avez bien mérité !

12.8 Un bref coup d'œil sur la technologie AJAX

AJAX (Asynchronous JavaScript and XML) est une technique pour créer des applications web plus interactives en exploitant les fonctionnalités offertes par JavaScript du côté client.

Deux bibliothèques JavaScript populaires sont Prototype¹³ et script.aculo.us¹⁴.

script.aculo.us étend la librairie Prototype en ajoutant des fonctionnalités pour l'animation et le glissé-déposé (*drag-and-drop*). Ces deux bibliothèques sont incluses dans Seaside via la paquetage *Scriptaculous*.

Toutes les images prêtes à l'emploi ont ce paquetage déjà chargé. La dernière version est disponible sur <http://www.squeaksource.com/Seaside>. Une démo en ligne est visible à l'adresse <http://scriptaculous.seasidehosting.st>. Si vous avez une image actuellement lancée, pointez tout simplement votre navigateur web sur la page <http://localhost:8080/seaside/tests/scriptaculous>.

Les extensions script.aculo.us suivent la même approche que Seaside lui-même — configuez des objets Smalltalk pour modéliser votre application et le code JavaScript nécessaire sera généré pour vous.

Jetons un coup d'œil sur un simple exemple pour voir comment le support JavaScript côté client peut rendre la réactivité de notre calculatrice RPN plus naturelle. Actuellement tout clic sur un chiffre entraîne une requête pour rafraîchir la page. Nous aimerais plutôt gérer l'édition de l'affichage côté client en mettant à jour l'affichage de la partie *display* de la page existante.

 Pour pouvoir communiquer depuis JavaScript avec des éléments précis de l'interface, nous devons tout d'abord donner à ces éléments un attribut *id unique*. Changez la méthode de rendu de la calculatrice¹⁵ comme suit :

```
MyCalculator»renderContentOn: html
    html div id: 'keypad'; with: keypad.
    html div id: 'display'; with: display.
```

 Pour pouvoir refaire le rendu du *display* lorsqu'un bouton du *keypad* est pressé, le composant *keypad* a besoin de connaître le composant *display*. Ajoutez une variable d'instance *display* à la classe *MyKeypad* et une méthode d'initialisation *MyKeypad»setDisplay:* et utilisez-la dans la méthode *MyCalculator»initialize*. Nous sommes maintenant capable d'adoindre du code JavaScript

13. <http://www.prototypejs.org>.

14. <http://script.aculo.us>.

15. Si vous n'avez pas implémenté le tutoriel vous-même, vous pouvez [télécharger l'exemple complet PBE-SeasideRPN depuis](#) <http://www.squeaksource.com/PharoByExample> et appliquer les modifications suggérées aux classes en RPN* au lieu des classes en My*.

aux boutons en mettant à jour MyKeypad»renderStackButton:callback:colSpan:on: comme suit :

```
MyKeypad»renderStackButton: text callback: aBlock colSpan: anInteger on: html
html tableData
  class: 'key';
  colSpan: anInteger;
  with: [
    html anchor
      callback: aBlock;
      onClick: "gère les événements JavaScript"
        (html updater
          id: 'display';
          callback: [ :r |
            aBlock value.
            r render: display];
          return: false);
    with: [ html html: text ]]
```

`onClick:` indique un gestionnaire d'événements JavaScript . `html updater` renvoie un *updater* c-à-d. une instance de SUUpdater, un objet Smalltalk représentant l'objet JavaScript Ajax.Updater (<http://www.prototypejs.org/api/ajax/updater>). Cet objet fait une requête AJAX et met à jour, par le texte en réponse, le contenu d'un conteneur. Le message `id:` dit à l'*updater* quel élément DOM XHTML mettre à jour ; ici, il s'agit du contenu de l'élément `div` d'attribut `id` "display". Un bloc est passé en argument à `callback:` pour se déclencher quand l'utilisateur cliquera sur le bouton. L'argument de bloc `r` est une nouvelle interface de rendu ou *render* qui peut être utilisée pour le rendu du composant `display` (remarquez que même si le code HTML est toujours accessible, il n'est plus valide au moment où ce `callback` est évalué). Avant de faire le rendu du composant `display`, nous évaluons `aBlock` pour effectuer l'action désirée.

`return: false` interdit au moteur JavaScript de déclencher le `callback` d'origine du lien, ce qui engendrerait un rafraîchissement complet. Nous pouvons aussi retirer l'ancre d'origine `callback:`, mais en la laissant, nous sommes sûrs que la calculatrice marchera même si le JavaScript est désactivé.

 *Essayez la calculatrice à nouveau et remarquez que le rafraîchissement complet de la page se produit toujours lorsque vous cliquez sur un chiffre du keypad (autrement dit, l'URL de la page web change à chaque clic).*

Bien que nous ayons bien implémenté le comportement du côté client, nous ne l'avons pas encore activé. Nous devons donc permettre la gestion des événements JavaScript.

 *Cliquez sur le lien Configure dans la barre d'outils de la calculatrice.*

Sélectionnez “Add Library :” SULibrary (pour configurer l’ajout de la bibliothèque SULibrary) et cliquez sur **Add** puis **Close**.

Vous pouvez aussi ajouter la bibliothèque de manière programmatique (plutôt que manuellement) lorsque vous enregistrez l’application :

```
MyCalculator class>initialize
```

```
(self registerAsApplication: 'rpn')
    addLibrary: SULibrary
```

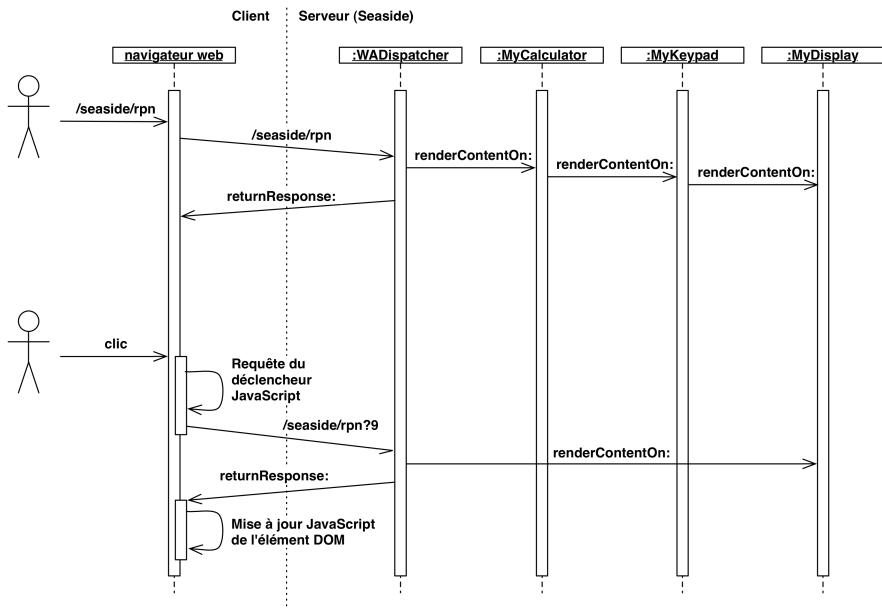


FIGURE 12.17 – Diagramme de séquences simplifié des interactions AJAX dans notre application Seaside.

💡 *Essayez l’application revisitée. Notez que la réponse est beaucoup plus naturelle. En particulier, aucune nouvelle URL n’est générée lors du clic.*

Vous devez vous demander : “oui mais, comment ça marche ?”. La figure 12.17 montre comment les deux versions—avec et sans AJAX—de l’application RPN (avec et sans AJAX) fonctionnent. AJAX court-circuite simplement le rendu de façon à mettre à jour le composant display *uniquelement*. JavaScript est responsable à la fois du déclenchement de la requête et de la mise à jour de l’élément DOM correspondant. Regardons le code source généré, principalement le code JavaScript.

```
new Ajax.Updater(
```

```
'display',
'http://localhost/seaside/RPN+Calculator',
{'evalScripts': true,
 'parameters': ['_s=zcdfqfonqwbeYzkza', '_k=jMORHtqr','9'].join('&'))};
return false
```

Pour des exemples plus avancés, nous vous invitons à visiter la page <http://localhost:8080/seaside/tests/scriptaculous> avec votre navigateur web.

Astuces. En cas de soucis du côté serveur, servez-vous du débogueur Smalltalk. Pour parer aux problèmes côté client, utiliser FireFox (<http://www.mozilla.com>) et FireBug, son débogueur JavaScript (<http://www.getfirebug.com/>) en plugin.

12.9 Résumé du chapitre

Nous avons vu que :

- La façon la plus simple de commencer avec Seaside est de télécharger le programme “Seaside One-Click Experience” sur <http://seaside.st>;
- Lancer ou arrêter le serveur se fait en évaluant WAKom startOn: 8080 ou WAKom stop respectivement ;
- Changer le *login* et mot de passe de l’administrateur peut se faire en évaluant WADispatcherEditor initialize ;
- **Toggle Halos** permet de visualiser directement le code source de l’application, les objets à l’exécution, les feuilles de style CSS et le code XHTML ;
- Envoyer WAGlobalConfiguration setDeploymentMode masque la barre d’outils.
- Les applications web Seaside sont composées de composants, chacun étant une sous-classe de **WAComponent** ;
- Seul un composant racine (*Root Component*) peut être enregistré comme application. Il devrait implémenter la méthode de classe canBeRoot. Il est possible d’enregistrer le composant comme application dans la méthode de classe initialize en envoyant self registerAsApplication: *chemin de l’application*. **Si vous surchargez description, il est possible de retourner une nom descriptif pour l’application qui sera affiché dans l’éditeur de configuration** ;
- Pour gérer le chaînage arrière ou *backtracking*, un composant devait disposer d’une méthode states renvoyant un tableau des objets dont l’état sera restauré quand l’utilisateur clique sur le bouton “retour arrière” de son navigateur web ;
- Le rendu d’un composant se fait via la méthode renderContentOn:. L’argument de cette méthode est un canevas destiné au rendu XHTML

- (généralement appelé html) ;
- Un composant peut faire le rendu d'un sous-composant en envoyant self render: *sous-component* ;
 - Le code XHTML est généré de manière programmatique en envoyant des messages à des *brushes*. Un *brush* est obtenu en envoyant un message, par exemple paragraph ou div, au canevas HTML ;
 - Si vous envoyez des messages en cascade à un *brush* qui contient le message with:, ce with: devra être le dernier message envoyé. Le message with: envoie le contenu et le fait le rendu du résultat ;
 - Les actions devrait apparaître uniquement dans des *callbacks* (ou fonctions de rappel). Vous ne devez jamais changer l'état de l'application durant la phase de rendu ;
 - Vous pouvez attacher plusieurs éléments graphiques de formulaire (ou *widgets*) et autre ancrés (ou liens) à des variables d'instance munies de méthodes d'accès en envoyant le message on: *variable d'instance of: objet au brush* ;
 - Vous pouvez définir la feuille de style CSS pour une hiérarchie de composants en définissant la méthode style de sorte à ce qu'elle retourne une chaîne de caractères contenant la feuille de style (pour les applications officiellement déployées, il est commun de se référer à une feuille de style externe se trouvant à une URL statique) ;
 - Les flux de contrôle peuvent être programmés en envoyant x call: y où le composant x sera remplacé par y jusqu'à ce qu'y réponde en envoyant le message answer: avec un résultat dans un *callback*. Le receveur de call: est habituellement self mais peut être de façon générale n'importe quel composant visible ;
 - Il existe un flux de contrôle nommé task — instance d'une sous-classe de WATask. Il devrait implémenter la méthode go pour appeler avec le message call: une série de composants dans une séquence de tâches ;
 - Vous pouvez vous simplifier le travail de création d'interactions simples en utilisant les méthodes utilitaires de WAComponent telles que request, inform:, confirm: et chooseFromCaption: ;
 - Pour interdire à l'utilisateur de se servir du bouton "retour arrière" de son navigateur web pour accéder un état d'exécution passé de l'application web, vous pouvez isoler des parties de la séquence des tâches dans des *transactions* en les incluant dans un bloc-argument du message isolate:.

Troisième partie

Pharo avancé

Chapitre 13

Classes et métaclasses

Comme nous l'avons vu dans le chapitre 5, en Smalltalk tout est objet, et tout objet est une instance d'une classe. Les classes ne sont pas des cas particuliers : les classes sont des objets, et les objets représentant les classes sont des instances d'autres classes. Ce modèle objet résume l'essence de la programmation orientée objet : il est petit, simple, élégant et uniforme. Cependant, les implications de cette uniformité peuvent prêter à confusion pour les débutants. L'objectif de ce chapitre est de montrer qu'il n'y a rien de compliqué, de "magique" ou de spécial ici : juste des règles simples appliquées uniformément. En suivant ces règles, vous pourrez toujours comprendre le code, quelle que soit la situation.

13.1 Les règles pour les classes et les métaclasses

Le modèle objet de Smalltalk est basé sur un nombre limité de concepts appliqués uniformément. Les concepteurs de Smalltalk ont appliqué le principe du "rasoir d'Occam" : toute considération conduisant à un modèle plus complexe que nécessaire a été abandonnée. Rappelons ici les règles du modèle objet qui ont été présentées dans le chapitre 5.

Règle 1. Tout est objet.

Règle 2. Tout objet est instance d'une classe.

Règle 3. Toute classe a une super-classe.

Règle 4. Tout se passe par envoi de messages.

Règle 5. La recherche de méthodes suit la chaîne d'héritage.

Comme nous l'avons mentionné en introduction de ce chapitre, une conséquence de la Règle 1 est que les *classes sont des objets aussi*, dans ce

cas la Règle 2 dit que les classes sont obligatoirement des instances de classes. La classe d'une classe est appelée une *méta-classe*.

Une métaclass est automatiquement créée pour chaque nouvelle classe. La plupart du temps, vous n'avez pas besoin de vous soucier ou de penser aux métaclasses. Cependant, chaque fois que vous utilisez le Browser pour naviguer du "côté classe" d'une classe, il est utile de se rappeler que vous êtes en train de naviguer dans une classe différente. Une classe et sa métaclass sont deux classes inséparables, même si la première est une instance de la seconde. Pour expliquer correctement les classes et les métaclasses, nous devons étendre les règles du chapitre 5 en ajoutant les règles suivantes :

Règle 6. Toute classe est une instance d'une métaclass.

Règle 7. La hiérarchie des métaclasses est parallèle à celle des classes.

Règle 8. Toute métaclass hérite de Class et de Behavior.

Règle 9. Toute métaclass est une instance de Metaclass.

Règle 10. La métaclass de Metaclass est une instance de Metaclass.

Ensemble, ces 10 règles complètent le modèle objet de Smalltalk. Nous allons tout d'abord revoir les 5 règles issues du chapitre 5 à travers un exemple simple. Ensuite, nous examinerons ces nouvelles règles à travers le même exemple.

13.2 Retour sur le modèle objet de Smalltalk

Puisque tout est objet, la couleur bleue est aussi un objet en Smalltalk.

Color blue → Color blue

Tout objet est une instance d'une classe. La classe de la couleur bleue est la classe Color :

Color blue class → Color

Toutefois, si l'on fixe la valeur *alpha* d'une couleur, nous obtenons une instance d'une classe différente, nommée TranslucentColor :

(Color blue alpha: 0.4) class → TranslucentColor

Nous pouvons créer un morph et fixer sa couleur à cette couleur translucide :

EllipseMorph new color: (Color blue alpha: 0.4); openInWorld

Vous pouvez voir l'effet produit dans la la figure 13.1.

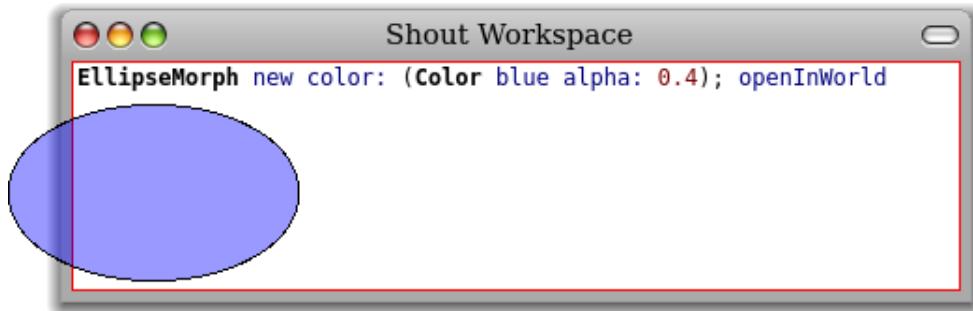


FIGURE 13.1 – Une ellipse translucide.

D'après la Règle 3, toute classe possède une super-classe. La super-classe de TranslucentColor est Color et la super-classe de Color est Object :

TranslucentColor superclass	→	Color
Color superclass	→	Object

Comme tout se produit par envoi de messages (Règle 4), nous pouvons en déduire que blue est un message à destination de Color ; class et alpha: sont des messages à destination de la couleur bleue ; openInWorld est un message à destination d'une ellipse morph et superclass est un message à destination de TranslucentColor et Color. Dans chaque cas, le receveur est un objet puisque tout est objet bien que certains de ces objets soient aussi des classes.

La recherche de méthodes suit la chaîne d'héritage (Règle 5), donc quand nous envoyons le message class au résultat de Color blue alpha: 0.4, le message est traité quand la méthode correspondante est trouvée dans la classe Object, comme illustré par la figure 13.2.

Cette figure résume l'essence de la relation *est un(e)*. Notre objet bleu translucide *est une instance de TranslucentColor*, mais nous pouvons aussi dire qu'il *est une Color* et qu'il *est un Object*, puisqu'il répond aux messages définis dans toutes ces classes. En fait, il y a un message, *isKindOf:*, qui peut être envoyé à n'importe quel objet pour déterminer s'il est en relation "*est un*" avec une classe donnée :

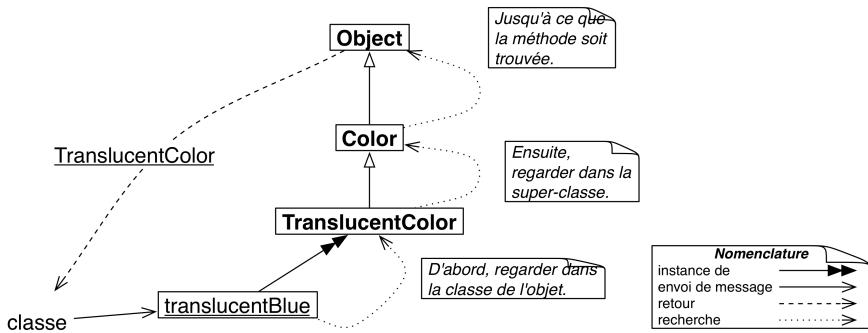


FIGURE 13.2 – Envoyer un message à une couleur translucide.

```

translucentBlue := Color blue alpha: 0.4.
translucentBlue isKindOf: TranslucentColor      → true
translucentBlue isKindOf: Color                 → true
translucentBlue isKindOf: Object                → true

```

13.3 Toute classe est une instance d'une métaclassee

Comme nous l'avons mentionné dans la section 13.1, les classes dont les instances sont aussi des classes sont appelées des métaclasses.

Les métaclasses sont implicites. Les métaclasses sont automatiquement créées quand une classe est définie. On dit qu'elles sont *implicites* car en tant que programmeur, vous n'avez jamais à vous en soucier. Une métaclassee implicite est créée pour chaque classe que vous créez donc chaque métaclassee n'a qu'une seule instance.

Alors que les classes ordinaires sont nommées par des variables globales, les métaclasses sont anonymes. Cependant, nous pouvons toujours les référencer à travers la classe qui est leur instance. Par exemple, la classe de **Color** est **Color class** et la classe de **Object** est **Object class** :

```

Color class   → Color class
Object class → Object class

```

La figure 13.3 montre que chaque classe est une instance de sa métaclassee (anonyme).

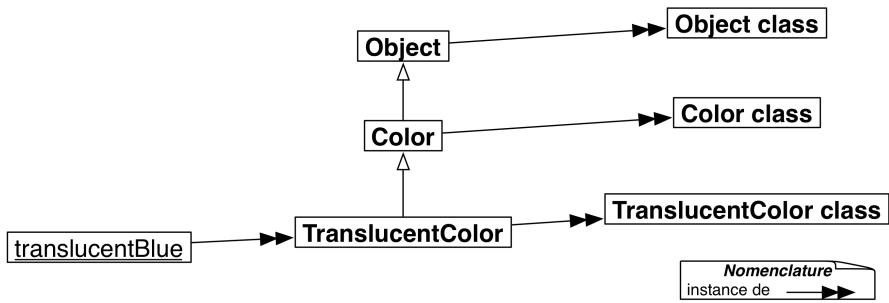


FIGURE 13.3 – Les métaclasses de TranslucentColor et ses super-classes.

Le fait que les classes soient aussi des objets facilite leur interrogation par envoi de messages. Voyons cela :

```

Color subclasses           → {TranslucentColor}
TranslucentColor subclasses → #()
TranslucentColor allSuperclasses → an OrderedCollection(Color Object
                                         ProtoObject)
TranslucentColor instVarNames → #('alpha')
TranslucentColor allInstVarNames → #('rgb' 'cachedDepth' 'cachedBitPattern' '
                                         alpha')
TranslucentColor selectors → an IdentitySet(#pixelValueForDepth:
                                         #pixelWord32
#convertToCurrentVersion:refStream: #isTransparent #scaledPixelValue32
                                         #bitPatternForDepth: #storeArrayValuesOn: #setRgB:alpha: #alpha #isOpaque
                                         #pixelWordForDepth: #isTranslucentColor #hash #isTranslucent #alpha: #storeOn:
                                         #asNontranslucentColor #privateAlpha #balancedPatternForDepth:)

```

13.4 La hiérarchie des métaclasses est parallèle à celle des classes

La Règle 7 dit que la super-classe d'une métaclassne ne peut pas être une classe arbitraire : elle est contrainte à être la métaclassne de la super-classe de l'unique instance de cette métaclassne.

TranslucentColor class superclass → Color class
TranslucentColor superclass class → Color class

C'est ce que nous voulons dire par le fait que la hiérarchie des métaclasses est parallèle à la hiérarchie des classes ; la figure 13.4 montre comment cela fonctionne pour la hiérarchie de `TranslucentColor`.

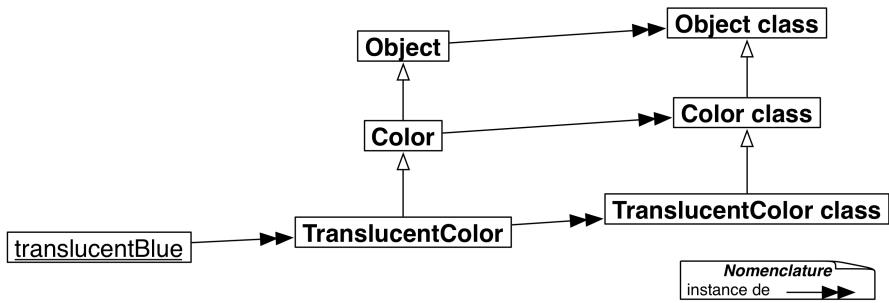


FIGURE 13.4 – La hiérarchie des métaclasses est parallèle à la hiérarchie des classes.

TranslucentColor class	→	TranslucentColor class
TranslucentColor class superclass	→	Color class
TranslucentColor class superclass superclass	→	Object class

L'uniformité entre les classes et les objets. Il est intéressant de revenir en arrière un moment et de réaliser qu'il n'y a pas de différence entre envoyer un message à un objet et à une classe. Dans les deux cas, la recherche de la méthode correspondante commence dans la classe du receveur et chemine le long de la chaîne d'héritage.

Ainsi, les messages envoyés à des classes doivent suivre la chaîne d'héritage des métaclasses. Considérons, par exemple, la méthode `blue` qui est implémentée du côté classe de `Color`. Si nous envoyons le message `blue` à `TranslucentColor`, alors il sera traité de la même façon que les autres messages. La recherche commence dans `TranslucentColor class` et continue dans la hiérarchie des métaclasses jusqu'à trouver dans `Color class` (voir la figure 13.5).

TranslucentColor blue	→	Color blue
-----------------------	---	------------

Notons que l'on obtient comme résultat un `Color blue` ordinaire, et non pas un translucide — il n'y a pas de magie !

Nous voyons donc qu'il y a une recherche de méthode uniforme en Smalltalk. Les classes sont juste des objets et se comportent comme tous les autres objets. Les classes ont le pouvoir de créer de nouvelles instances uniquement parce qu'elles répondent au message `new` et que la méthode pour `new` sait créer de nouvelles instances. Normalement, les objets qui ne sont pas des classes ne comprennent pas ce message, mais si vous avez une bonne raison

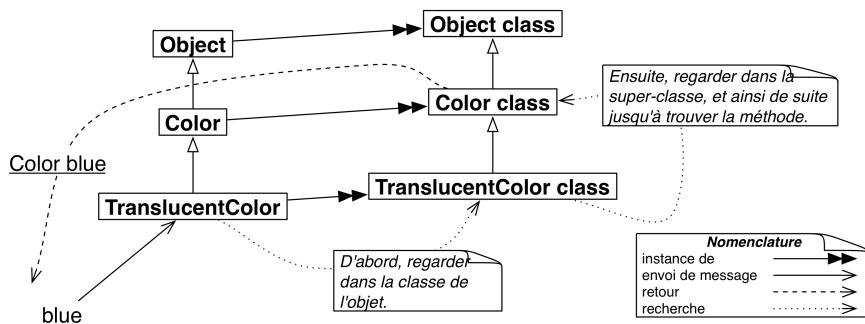


FIGURE 13.5 – Le traitement des messages pour les classes est le même que pour les objets ordinaires.

pour faire cela, il n'y a rien qui vous empêche d'ajouter une méthode new à une classe qui n'est pas une métaclass.

Comme les classes sont des objets, nous pouvons aussi les inspecter.

Inspectez Color blue et Color.

Notons que vous inspectez, dans un cas, une instance de Color et dans l'autre cas, la classe Color elle-même. Cela peut prêter à confusion parce que la barre de titre de l'inspecteur contient le nom de la classe de l'objet en cours d'inspection. L'inspecteur sur Color vous permet de voir entre autre la super-classe, les variables d'instances, le dictionnaire des méthodes de la classe Color, comme indiqué dans la figure 13.6.

13.5 Toute métaclass hérite de Class et de Behavior

Toute métaclass est une classe, donc hérite de Class. À son tour, Class hérite de ses super-classes, ClassDescription et Behavior. En Smalltalk, puisque tout est un objet, ces classes héritent finalement toutes de Object. Nous pouvons voir le schéma complet dans la figure 13.7.

Où est défini new? Pour comprendre l'importance du fait que les métaclasses héritent de Class et de Behavior, demandons-nous où est défini new

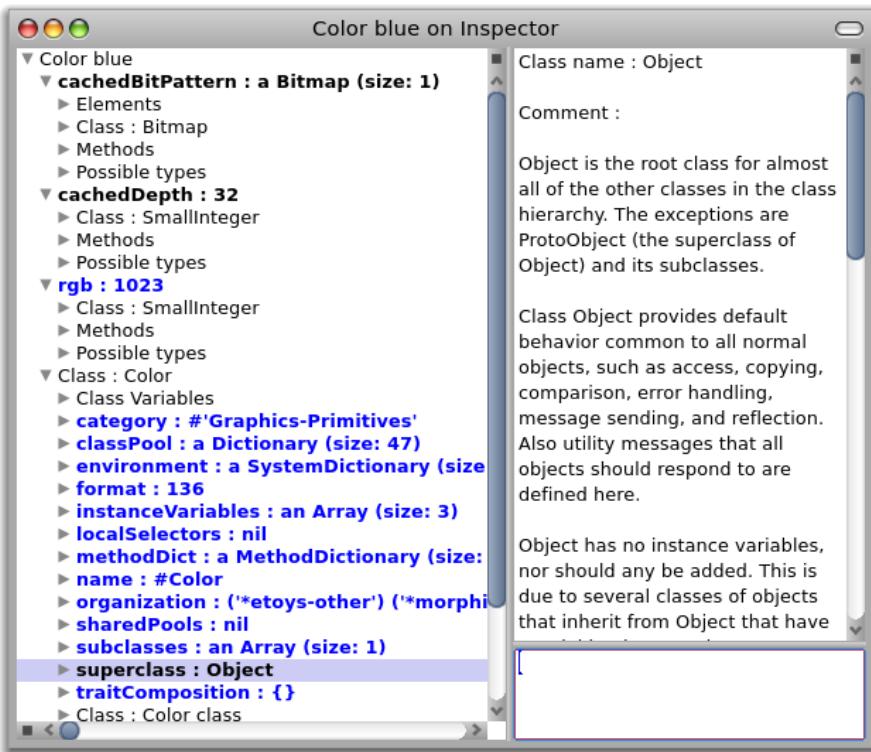


FIGURE 13.6 – Les classes sont aussi des objets.

et comment cette définition est trouvée. Quand le message `new` est envoyé à une classe, il est recherché dans sa chaîne de métaclasses et finalement dans ses super-classes `Class`, `ClassDescription` et `Behavior` comme montré dans la figure 13.8.

La question “Où est défini `new`?” est cruciale. La méthode `new` est définie en premier dans la classe `Behavior` et peut être redéfinie dans ses sous-classes, ce qui inclut toutes les métaclasses des classes que nous avons définies, si cela est nécessaire. Maintenant, quand un message `new` est envoyé à une classe, il est recherché, comme d’habitude, dans la métaclass de cette classe, en continuant le long de la chaîne de super-classes jusqu’à la classe `Behavior` si aucune redéfinition n’a été rencontrée sur le chemin.

Notons que le résultat de l’envoi de message `TranslucentColor new` est une instance de `TranslucentColor` et non de `Behavior`, même si la méthode est trouvée dans la classe `Behavior` ! `new` retourne toujours une instance de `self`, la classe qui a reçu le message, même si cela est implémenté dans une autre classe.

`TranslucentColor new class` → `TranslucentColor "et non pas Behavior"`

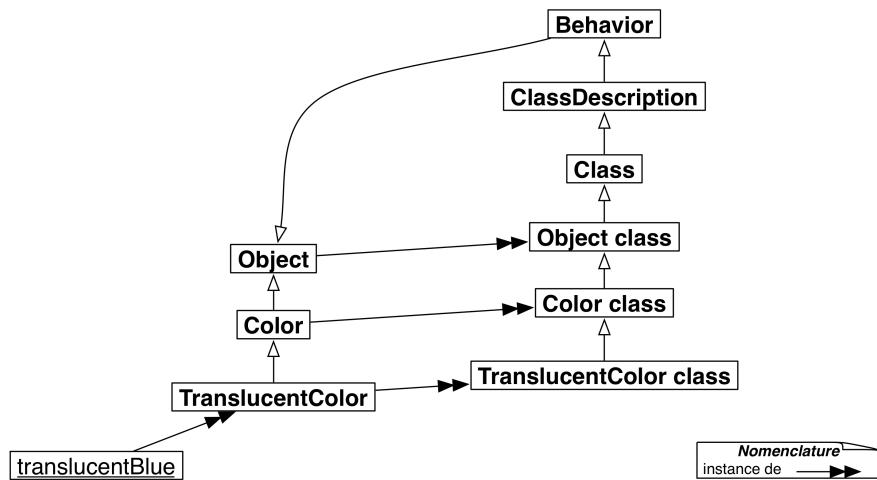


FIGURE 13.7 – Les métaclasses héritent de Class et de Behavior.

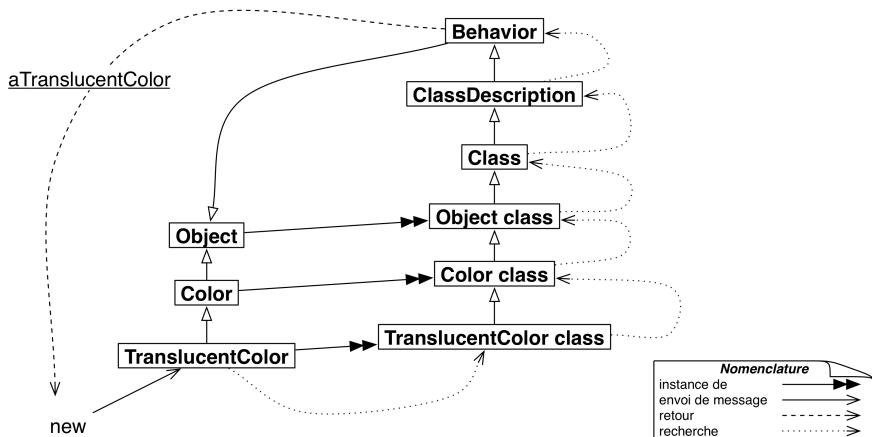


FIGURE 13.8 – `new` est un message ordinaire recherché dans la chaîne des métaclasses.

Une erreur courante est de rechercher `new` dans la super-classe de la classe du receveur. La même chose se produit pour `new`: le message standard pour créer un objet d'une taille donnée. Par exemple, `Array new: 4` crée un tableau de 4 éléments. Vous ne trouverez pas la définition de cette méthode dans `Array` ni dans aucune de ses super-classes. À la place, vous devriez regarder dans `Array class` et ses super-classes puisque c'est là que la recherche commence.

Les responsabilités de Behavior, ClassDescription et Class. Behavior fournit l'état minimum et nécessaire à des objets possédant des instances : cela inclut un lien super-classe, un dictionnaire de méthodes et une description des instances (*c-à-d.* représentation et nombre). Behavior hérite de Object, donc elle, ainsi que toutes ses sous-classes peuvent se comporter comme des objets.

Behavior est aussi l'interface basique pour le compilateur. Elle fournit des méthodes pour créer un dictionnaire de méthodes, compiler des méthodes, créer des instances (*c-à-d.* new, basicNew, new:, et basicNew:), manipuler la hiérarchie de classes (*c-à-d.* superclass :, addSubclass:), accéder aux méthodes (*c-à-d.* selectors, allSelectors, compiledMethodAt:), accéder aux instances et aux variables (*c-à-d.* allInstances, instVarNames ...), accéder à la hiérarchie de classes (*c-à-d.* superclass, subclasses) et interroger (*c-à-d.* hasMethods, includesSelector, canUnderstand:, inheritsFrom:, isVariable).

ClassDescription est une classe abstraite qui fournit des facilités utilisées par ses deux sous-classes directes, Class et Metaclass. ClassDescription ajoute des facilités fournies à la base par Behavior : des variables d'instances nommées, la catégorisation des méthodes dans des protocoles, la notion de nom (abstrait), la maintenance de *change sets*, la journalisation des changements et la plupart des mécanismes requis pour l'exportation de *change sets*.

Class représente le comportement commun de toutes les classes. Elle fournit un nom de classe, des méthodes de compilation, des méthodes de stockage et des variables d'instance. Elle fournit aussi une représentation concrète pour les noms des variables de classe et des variables de pool (addClassVarName:, addSharedPool:, initialize). Class sait comment créer des instances donc toutes les métaclasses doivent finalement hériter de Class.

13.6 Toute métaclasse est une instance de Metaclass

Les métaclasses sont aussi des objets ; elles sont des instances de la classe Metaclass comme montré dans la figure 13.9. Les instances de la classe Metaclass sont les métaclasses anonymes ; chacune ayant exactement une unique instance qui est une classe.

Metaclass représente le comportement commun des métaclasses. Elle fournit des méthodes pour la création d'instances (sub\–class\–Of:) permettant de créer des instances initialisées de l'unique instance Metaclass pour l'initialisation des variables de classe, la compilation de méthodes et l'obtention d'informations à propos des classes (liens d'héritage, variables d'instance, etc.).

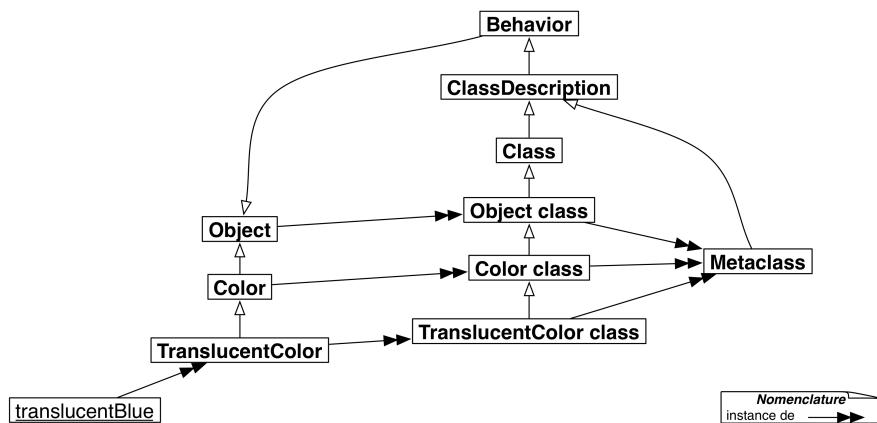


FIGURE 13.9 – Toute métaclass est une instance de Metaclass.

13.7 La métaclass de Metaclass est une instance de Metaclass

La dernière question à laquelle il faut répondre est : quelle est la classe de Metaclass class ? La réponse est simple : il s’agit d’une métaclass, donc forcément une instance de Metaclass, exactement comme toutes les autres métaclasses dans le système (voir la figure 13.10).

La figure montre que toutes les métaclasses sont des instances de Metaclass, ce qui inclut aussi la métaclass de Metaclass. Si vous comparez les figures 13.9 et 13.10, vous verrez comment la hiérarchie des métaclasses reflète parfaitement la hiérarchie des classes, tout le long du chemin jusqu’à Object class.

Les exemples suivants montrent comment il est possible d’interroger la hiérarchie de classes afin de démontrer que la figure 13.10 est correcte. En réalité, vous verrez que nous avons dit un pieux mensonge — Object class superclass → ProtoObject class, et non Class. En Pharo, il faut aller une super-classe plus haut dans la hiérarchie pour atteindre Class.

Exemple 13.1 – La hiérarchie des classes

TranslucentColor superclass	→	Color
Color superclass	→	Object

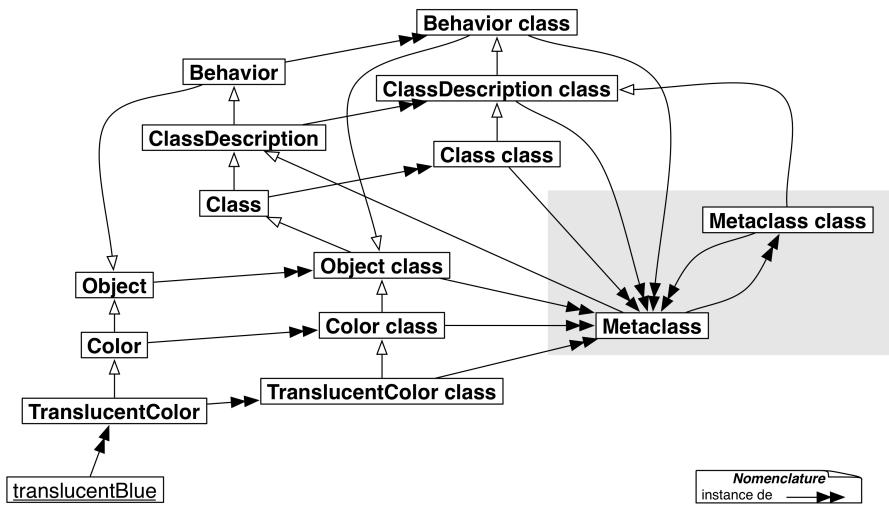


FIGURE 13.10 – Toutes les métaclasses sont des instances de la classe Metaclass, même la métaclass de Metaclass.

Exemple 13.2 – La hiérarchie parallèle des métaclasses

TranslucentColor class superclass	→ Color class
Color class superclass	→ Object class
Object class superclass superclass <i>class</i> "	→ Class "Attention : saute ProtoObject
Class superclass	→ ClassDescription
ClassDescription superclass	→ Behavior
Behavior superclass	→ Object

Exemple 13.3 – Les instances de Metaclass

TranslucentColor class class → Metaclass
Color class class → Metaclass
Object class class → Metaclass
Behavior class class → Metaclass

Exemple 13.4 – Metaclass class est une Metaclass

Metaclass class class → Metaclass
Metaclass superclass → ClassDescription

13.8 Résumé du chapitre

Maintenant, vous devriez mieux comprendre la façon dont les classes sont organisées et l'impact de l'uniformité du modèle objet. Si vous vous

perdez ou que vous vous embrouillez, vous devez toujours vous rappeler que l'envoi de messages est la clé : cherchez alors la méthode dans la classe du receveur. Cela fonctionne pour *tous* les receveurs. Si une méthode n'est pas trouvée dans la classe du receveur, elle est recherchée dans ses super-classes.

1. Toute classe est une instance d'une métaclass. Les métaclasses sont implicites. Une métaclass est créée automatiquement à chaque fois que vous créez une classe ; cette dernière étant sa seule instance.
2. La hiérarchie des métaclasses est parallèle à celle des classes. La recherche de méthodes pour les classes est analogue à la recherche de méthodes pour les objets ordinaires et suit la chaîne des super-classes entre métaclasses.
3. Toute métaclass hérite de Class et de Behavior. Toute classe *est une* Class. Puisque les métaclasses sont aussi des classes, elles doivent hériter de Class. Behavior fournit un comportement commun à toutes les entités ayant des instances.
4. Toute métaclass est une instance de Metaclass. ClassDescription fournit tout ce qui est commun à Class et à Metaclass.
5. La métaclass de Metaclass est une instance de Metaclass. La relation *instance-de* forme une boucle fermée, donc Metaclass class class → Metaclass.

Chapitre 14

La réflexivité

Smalltalk est un langage de programmation réflexif. Brièvement, cela signifie que les programmes ont la possibilité d'agir sur leur propre exécution et structure. Plus techniquement, cela signifie que les *méta-objets* du système en cours d'exécution peuvent être *réifiés* sous forme d'objets ordinaires qui peuvent alors recevoir des requêtes et être inspecter. Les méta-objets dans Smalltalk sont les classes, métaclasses, dictionnaires de méthodes, méthodes compilées, pile d'exécution, etc ... Cette forme de réflexivité est également appelée *introspection* et est disponible dans de nombreux langages de programmation.

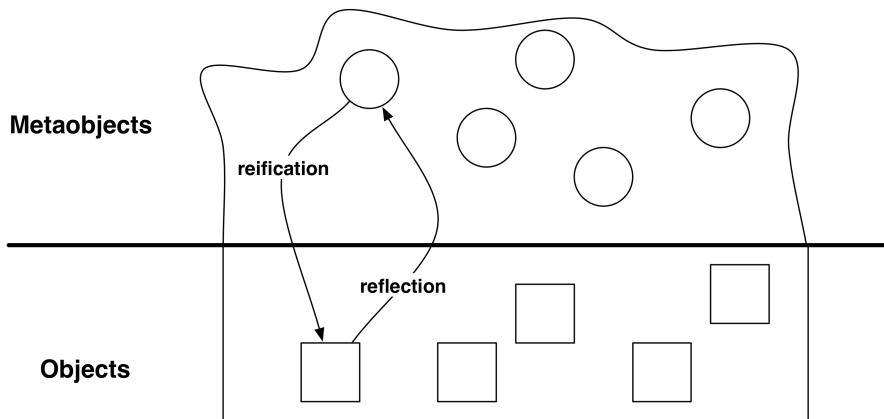


FIGURE 14.1 – Réification et réflexivité.

Inversement il est possible en Smalltalk de modifier les méta-objets réifiés et que leurs modifications soient prises en compte lors de l'exécution du système (voir la figure 14.1). Ceci est appelé *intercession* et est principalement utilisé dans les langages de programmation dynamique et de manière

plus limitée dans les langages statiques.

Un programme qui manipule d'autres programmes (ou même lui-même) est un *méta-programme*. Pour qu'un langage de programmation soit réflexif, il faut qu'il supporte à la fois l'introspection et l'intercession.

L'introspection est la capacité afin *d'examiner* les structures de données qui définissent le programme comme les objets, classes, méthodes ou pile d'exécution. L'intercession est la capacité de modifier ces structures, en d'autre terme de changer la sémantique du langage et le comportement d'un programme depuis le programme lui-même. La *réflexivité structurelle* s'intéresse à l'exploration et à la modification des structures lors du l'exécution du système, alors que la *réflexivité de comportement* concerne l'interprétation de ces structures.

Dans ce chapitre, nous allons principalement nous intéresser à la réflexivité structurelle. Nous illustrerons avec plusieurs exemples pratiques comment Smalltalk supporte l'introspection et la métaprogrammation.

14.1 Introspection

En utilisant un inspecteur, il est possible d'examiner un objet, de changer les valeurs de ces variables d'instances et même de lui envoyer un message.

➊ Évaluer le code qui suit dans un workspace :

```
w := Workspace new.  
w openLabel: 'My Workspace'.  
w inspect
```

Ceci va ouvrir un deuxième workspace et un inspecteur. L'inspecteur montre l'état interne de ce nouveau workspace : la liste de ces variables d'instances dans la partie gauche (*dependents*, *contents*, *bindings...*) et la valeur de la variable d'instance sélectionnée dans la partie droite. La variable d'instance *contents* représente ce que le workspace affiche dans sa zone de texte. Ainsi si vous la sélectionnez, la partie droite montrera une chaîne de caractères vide.

➋ Maintenant tapez 'hello' à la place de la chaîne de caractères vide, puis ensuite faire accept.

La valeur de la variable *contents* change, mais la fenêtre du workspace n'en sera pas notifié, c'est-à-dire qu'elle ne ré-affiche pas son contenu. Afin d'activer le rafraîchissement de la fenêtre, évaluer *self contentsChanged* dans la partie inférieure de l'inspecteur.

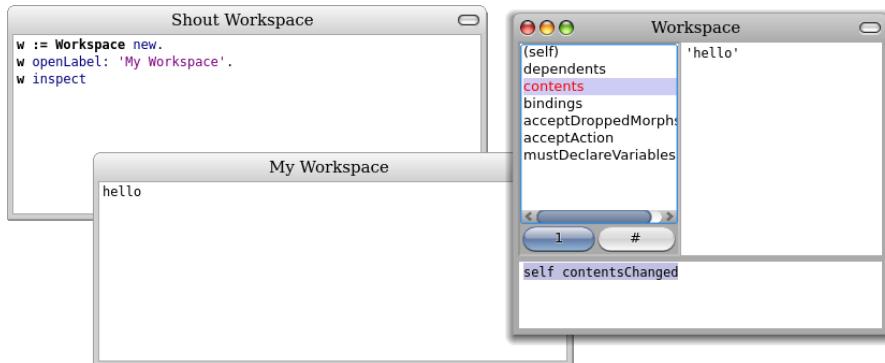


FIGURE 14.2 – Inspecter un Workspace.

Accéder aux variables d’instances

Comment fonctionne l’inspecteur ? En Smalltalk, toutes les variables d’instances sont protégées. En théorie, il est impossible d’y accéder depuis un autre objet si la classe ne définit pas d’accesseur. En pratique, l’inspecteur peut accéder aux variables sans avoir besoin d’accesseurs, parce qu’il utilise les capacités réflexives de Smalltalk. En Smalltalk, les classes définissent les variables d’instances soient par nom ou au moyen d’indices numériques. L’inspecteur utilisent des méthodes définies dans la classe Object afin d’y accéder : `instVarAt : index` et `instVarNamed : aString` peuvent être utilisé pour avoir respectivement la valeur de la variable d’instance à la position `index` ou identifié par `aString` ; pour associer de nouvelles valeurs à ces variables d’instances, on utilise `instVarAt:put:` et `instVarNamed:put:`.

Par exemple, vous pouvez changer la valeur de la variable d’instance `contents` de `w` précédemment définie en évaluant :

```
w instVarNamed: 'contents' put: 'howdy'; contentsChanged
```

Caveat : Bien que ces méthodes sont utiles pour construire les outils de l’environnement de développement, les utiliser dans le cadre d’une application conventionnelle est une mauvaise idée : ces méthodes réflexives rompent l’encapsulation des objets et peuvent rendre votre code plus difficile à comprendre et à maintenir.

`instVarAt:` et `instVarAt:put:` sont des méthodes primitives, c’est-à-dire qu’elles sont implémentées comme des opérations primitives de la machine virtuelle Pharo. Si vous consultez le code de ces méthodes, la syntaxe spéciale d’un pragma `<primitive: N>` où `N` est un entier.

```
Object>instVarAt: index
"Primitive. Answer a fixed variable in an object. ..."
<primitive : 73>
"Access beyond fixed variables."
↑self basicAt: index - self class instSize
```

Généralement le code après l'invocation de la primitive n'est pas exécuté. Il est exécuté seulement si la primitive échoue. Dans le cas qui nous intéresse, si on essaie d'accéder à une variable qui n'existe pas, alors le code qui suit la primitive sera essayé. Ceci permet aussi d'utiliser le débogueur sur des méthodes primitives. Bien qu'il est possible de modifier le code des méthodes primitives, ceci est risqué pour la stabilité de votre image Pharo.

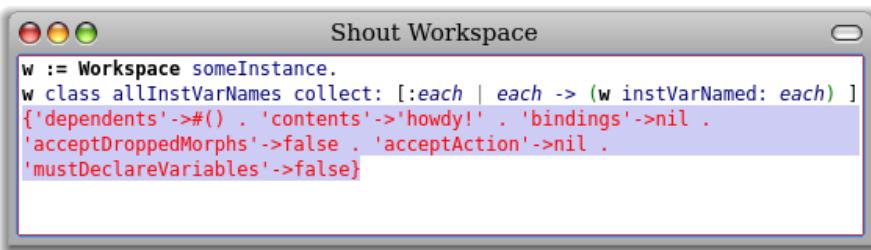


FIGURE 14.3 – Afficher toutes les variables d’instance d’un Workspace.

la figure ?? montre comment afficher les valeurs des variables d’instances d’une instance (w) de la classe Workspace. La méthode allInstVarNames retourne l’ensemble des noms de variables d’instances d’une classe donnée.

De la même façon, il est possible de collecter les instances qui ont certaines propriétés. Par exemple, pour avoir toutes les instances de la classe SketchMorph dont la variable d’instance owner est initialisée avec un morph de type world (c-à-d. les morphs qui sont affichés à chaque instant à l’écran), essayer cette expression :

```
SketchMorph allInstances select: [:c | (c instVarNamed: 'owner') isWorldMorph]
```

Parcourir les variables d’instances

Considérons le message instanceVariableValues, qui retourne une collection de toutes les valeurs des variables d’instances définies dans la classe, en excluant les variables d’instances héritées. Par exemple :

```
(1@2) instanceVariableValues → an OrderedCollection(1 2)
```

La méthode est implémentée dans Object de la manière qui suit :

```
Object»instanceVariableValues
```

"Answer a collection whose elements are the values of those instance variables of the receiver which were added by the receiver's class."

```
| c |
c := OrderedCollection new.
self class superclass instSize + 1
    to: self class instSize
    do: [ :i | c add: (self instVarAt: i)].
^ c
```

Cette méthode parcourt par indice les variables d'instances que la classe définit, à partir du dernier index utilisé par les superclasses. (La méthode `instSize` retourne le nombre des variables d'instances nommées que la classe définit.)

Faire des requêtes sur les classes et interfaces

Les outils de développement Pharo (navigateur de code, débogueur, inspecteur, ...) utilisent tous les mécanismes réflexifs que nous avons vu jusqu'à présent.

Voici quelques messages supplémentaires qui peuvent être utile afin de construire des outils de développement :

`isKindOf : aClass` retourne vrai si le receveur est une instance de `aClass` ou d'une de ces superclasses. Par exemple :

```
1.5 class      → Float
1.5 isKindOf: Number → true
1.5 isKindOf: Integer → false
```

`respondsTo : aSymbol` retourne vrai si le receveur a une méthode dont le sélecteur est `aSymbol`. Par exemple :

```
1.5 respondsTo: #floor → true "car Number implémente floor"
1.5 floor → 1
Exception respondsTo: #, → true "exception classes can be grouped"
```

Caveat : Bien que tous ces outils soient particulièrement utiles pour construire des outils de développements, ils ne sont pas appropriés pour une application classique. Demander à un objet sa classe ou bien l'interroger pour connaître les messages qu'il comprend, sont un signe indiquant une mauvaise conception objet, puisque généralement cela signifie une violation du principe d'encapsulation. Les outils de développements ne sont pas considérés comme des applications comme les autres, puisque leur domaine d'applications Development tools, however, are not normal applications, since their domain is that of software itself. As such these tools have a right to dig deep into the internal details of code.

Métriques de code

Voyons maintenant comment nous pouvons utiliser les mécanismes d'introspection de Smalltalk pour rapidement pouvoir construire des métriques de code. Les métriques de code mesurent certains aspects comme la profondeur de l'arbre d'héritage, le nombre de sous-classes directes ou indirectes, le nombre de méthodes ou de variables d'instances de chaque classe ou enfin le nombre local de méthodes ou de variables d'instances. Voici quelques résultats de métriques pour la classe Morph, qui est la superclasse de tous les objets graphiques de Pharo, révélant qu'il s'agit d'une classe d'une taille conséquente et qu'elle est la racine d'une hiérarchie importante. Peut-être qu'elle nécessiterait d'être refactoriser !

Morph allSuperclasses size.	→	2 "profondeur d'héritage"
Morph allSelectors size.	→	1378 "nombre de méthodes"
Morph allInstVarNames size.	→	6 "nombre de variables d'instances"
Morph selectors size.	→	998 "nombre de nouvelles méthodes"
Morph instVarNames size.	→	6 "nombre de nouvelles variables"
Morph subclasses size.	→	45 "sous-classes directes"
Morph allSubclasses size.	→	326 "total de sous-classes"
Morph linesOfCode.	→	5968 "nombre total de lignes de code!"

Une des métriques les plus intéressantes dans le domaine de la programmation par objet est le nombre de méthodes qui étendent les méthodes héritées de la superclasse. Ceci nous informe de la relation entre une classe et ses superclasses. Dans les prochaines sections, nous verrons comment exploiter notre connaissance des mécanismes d'exécution pour répondre à de telles questions.

14.2 Parcourir le code

En Smalltalk, tout est objet. Les classes sont en particulier des objets qui fournissent des mécanismes utiles afin de parcourir leurs instances. La plupart des messages que nous allons voir maintenant sont implémentés dans la classe Behavior. Ils sont donc compris de toutes les classes.

Comme nous avons vu précédemment, nous pouvons obtenir une instance particulière d'une classe donnée en lui envoyant le message #someInstance.

```
Point someInstance → 0@0
```

Vous pouvez également rassembler toutes les instances avec #allInstances ou déterminer le nombre d'instances en mémoire avec #instanceCount.

ByteString allInstances	→	#('collection' 'position' ...)
ByteString instanceCount	→	104565
String allSubInstances size	→	101675

Ces caractéristiques peuvent être très utiles lors du débogage d'une application, car il est possible de demander à une classe d'énumérer les méthodes possédant des propriétés spécifiques.

- whichSelectorsAccess: retourne la liste de tous les sélecteurs de méthodes qui lisent ou écrivent dans une variable dont le nom est passé en argument
- whichSelectorsStoreInto: retourne les sélecteurs des méthodes qui modifient la valeur d'une variable d'instance
- whichSelectorsReferTo: retourne les sélecteurs des méthodes qui envoient un certain message
- crossReference associe chaque message avec l'ensemble des méthodes qui l'envoient.

Point whichSelectorsAccess: 'x'	→	an IdentitySet(#\\'#= #scaleBy: ...)
Point whichSelectorsStoreInto: 'x'	→	an IdentitySet(#setX:setY: ...)
Point whichSelectorsReferTo: #+	→	an IdentitySet(#rotateBy:about: ...)
Point crossReference	→	an Array(an Array('*' an IdentitySet(#rotateBy:about: ...)) an Array('+'' an IdentitySet(#rotateBy:about: ...)) ...)

Les messages qui suivent prennent en compte l'héritage :

- whichClassIncludesSelector: retourne la super-classe qui implémente le message concerné
- unreferencedInstanceVariables retourne la liste des variables d'instances qui ne sont ni utilisées dans la classe du receveur, ni dans aucune de ses sous-classes

```
Rectangle whichClassIncludesSelector: #inspect    → Object
Rectangle unreferencedInstanceVariables      → #()
```

`SystemNavigation` is a facade that supports various useful methods for querying and browsing the source code of the system. `SystemNavigation default` returns an instance you can use to navigate the system. Par exemple :

```
SystemNavigation default allClassesImplementing: #yourself → {Object}
```

Les messages suivants devraient être également compréhensibles par eux-même :

```
SystemNavigation default allSentMessages size      → 24930
SystemNavigation default allUnsentMessages size   → 6431
SystemNavigation default allUnimplementedCalls size → 270
```

Notons que les messages implémentés mais non envoyés ne sont pas nécessairement inutiles, car ils peuvent être envoyés implicitement (*par ex.*, en utilisant `perform`). Les messages envoyés mais non implémentés sont plus problématiques, car les méthodes envoyant ces messages vont échoués à l'exécution. Ceci peut être le signe d'une implémentation non finie, d'une API obsolète ou bien de librairies manquantes.

`SystemNavigation default allCallsOn: #Point` retourne tous les messages envoyés explicitement à `Point` comme receveur du message.

All these features are integrated in the programming environment of Pharo, in particular in the code browsers. As you are surely already aware, there are convenient keyboard shortcuts for browsing all `implementors` (`CMD-m`) and `senders` (`CMD-n`) of a given message. What is perhaps not so well known is that there are many such pre-packaged queries implemented as methods of the `SystemNavigation` class in the *browsing* protocol. For example, you can programmatically browse all implementors of the message `ifTrue:` by evaluating :

```
SystemNavigation default browseAllImplementorsOf: #ifTrue:
```

Particularly useful are the methods `browseAllSelect:` and `browseMethodsWithSourceString ::`. Here are two different ways to browse all methods in the system that perform super sends (the first way is rather brute force ; the second way is better and eliminates some false positives) :

```
SystemNavigation default browseMethodsWithSourceString: 'super'.
SystemNavigation default browseAllSelect: [:method | method sendsToSuper ].
```

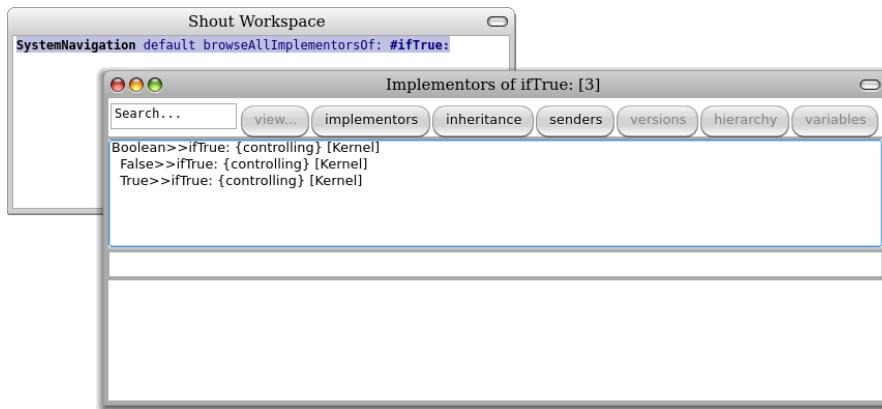


FIGURE 14.4 – Parcourir toutes les implémentations de #ifTrue:.

14.3 Classes, dictionnaires de méthodes et méthodes

Comme les classes sont des objets, il est possible de les inspecter ou de les explorer de la même manière que les objets.

Évaluer Point explore.

Dans la figure 14.5, l'explorer montre la structure de la classe Point. You can see that the class stores its methods in a dictionary, indexing them by their selector. The selector `#*` points to the decompiled bytecode of `Point»*`.

Examinons la relation entre classes et méthodes. Dans la figure 14.6 nous voyons que classes et métaclasses ont en commun la superclasse Behavior. C'est où la méthode new est définie parmi d'autres méthodes clés pour les classes. Chaque classe a un dictionnaire de méthode, qui associe chaque sélecteur de méthodes à des méthodes compilées. Chaque méthode compilée connaît la classe dans laquelle elle est installée. Dans la figure 14.5, qu'elle est conservée au moyen d'une association dans literal5.

On peut exploiter les relations qui établissent classes et méthodes, pour effectuer des requêtes sur le système. For example, to discover which methods are newly introduced in a given class, *c-à-d.* do not override superclass methods, we can navigate from the class to the method dictionary as follows :

```
[:aClass| aClass methodDict keys select: [:aMethod |
  (aClass superclass canUnderstand: aMethod) not ]] value: SmallInteger
  → an IdentitySet(#threeDigitName #printStringBase:nDigits: ...)
```

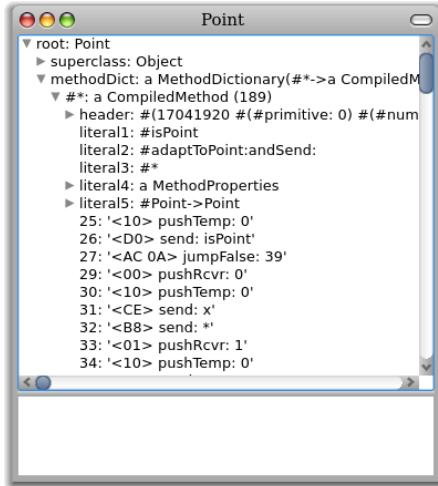


FIGURE 14.5 – Explorer class Point and the bytecode of its #* method.

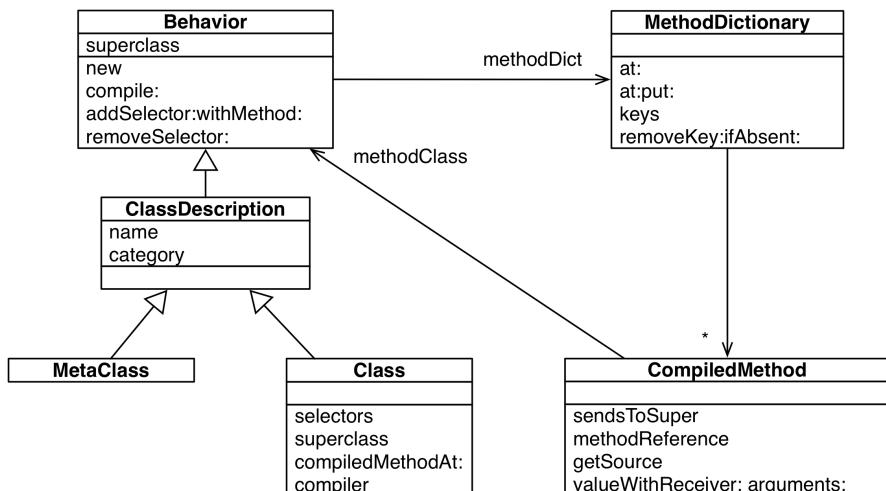


FIGURE 14.6 – Classes, dictionnaires de méthodes et méthodes compilées

A compiled method does not simply store the bytecode of a method. It is also an object that provides numerous useful methods for querying the system. One such method is `isAbstract` (which tells if the method sends `subclassNameResponsibility`). We can use it to identify all the abstract methods of an abstract class

```
[:aClass| aClass methodDict keys select: [:aMethod |
  (aClass>>aMethod) isAbstract ]] value: Number
  → an IdentitySet(#storeOn:base: #printOn:base: #+ #- #* #/ ...)
```

Note that this code sends the `>>` message to a class to obtain the compiled method for a given selector.

To browse the super-sends within a given hierarchy, for example within the Collections hierarchy, we can pose a more sophisticated query :

```
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'
```

Note how we navigate from classes to method dictionaries to compiled methods to identify the methods we are interested in. A `MethodReference` is a lightweight proxy for a compiled method that is used by many tools. There is a convenience method `CompiledMethod»methodReference` to return the method reference for a compiled method.

```
(Object>>#=) methodReference methodSymbol → #=
```

14.4 Environnements de navigation du code

Although `SystemNavigation` offers some useful ways to programmatically query and browse system code, there is a better way. The Refactoring Browser, which is integrated into Pharo, provides both interactive and programmatic ways to pose complex queries.

Suppose we are interested to discover which methods in the `Collection` hierarchy send a message to `super` which is different from the method's selector. This is normally considered to be a bad code smell, since such a super-send should normally be replaced by a self-send. (Think about it — you only need `super` to extend a method you are overriding ; all other inherited methods can be accessed by sending to `self` !)

The refactoring browser provides us with an elegant way to restrict our query to just the classes and methods we are interested in.

 Open a browser on the class `Collection`. cliquer avec le bouton d'action on the class name and select refactoring scope>subclasses with. This will open a new Browser Environment on just the Collection hierarchy. Within

this restricted scope select refactoring scope>super-sends to open a new environment with all methods that perform super-sends within the Collection hierarchy. Now cliquer on any method and select refactor>code critics. Navigate to Lint checks>Possible bugs>Sends different super message and cliquer avec le bouton d'action to select browse .

In la figure 14.7 we can see that 19 such methods have been found within the Collection hierarchy, including Collection>printNameOn:, which sends super printOn::.

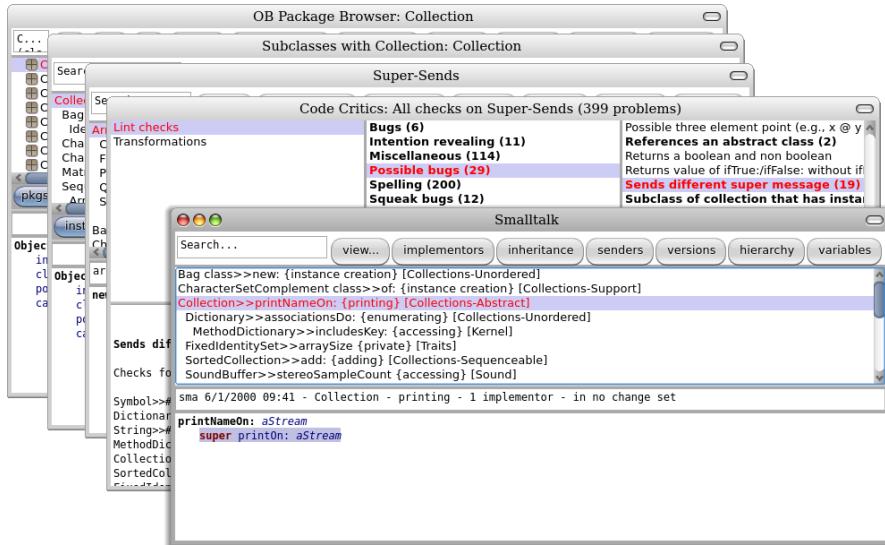


FIGURE 14.7 – Finding methods that send a different super message.

Browser environments can also be created programmatically. Here, for example, we create a new BrowserEnvironment for Collection and its subclasses, select the super-sending methods, and open the resulting environment.

```
((BrowserEnvironment new forClasses: (Collection withAllSubclasses))
  selectMethods: [:method | method sendsToSuper])
  label: 'Collection methods sending super';
  open.
```

Note how this is considerably more compact than the earlier, equivalent example using SystemNavigation.

Finally, we can find just those methods that send a different super message programmatically as follows :

```
((BrowserEnvironment new forClasses: (Collection withAllSubclasses))
    selectMethods: [:method |
        method sendsToSuper
        and: [((method parseTree superMessages includes: method selector) not)]]
    label: 'Collection methods sending different super';
    open)
```

Here we ask each compiled method for its (Refactoring Browser) parse tree, in order to find out whether the super messages differ from the method's selector. Have a look at the *querying* protocol of the class RBProgramNode to see some the things we can ask of parse trees.

14.5 Accéder au contexte d'exécution

We have seen how Smalltalk's reflective capabilities let us query and explore objects, classes and methods. But what about the run-time environment?

Contextes des méthodes

In fact, the run-time context of an executing method is in the virtual machine — it is not in the image at all! On the other hand, the debugger obviously has access to this information, and we can happily explore the run-time context, just like any other object. How is this possible?

Actually, there is nothing magical about the debugger. The secret is the pseudo-variable `thisContext`, which we have encountered only in passing before. Whenever `thisContext` is referred to in a running method, the entire run-time context of that method is reified and made available to the image as a series of chained `MethodContext` objects.

We can easily experiment with this mechanism ourselves.

 Change the definition of `Integer»factorial` by inserting the underlined expression as shown below :

```
Integer»factorial
    "Answer the factorial of the receiver."
    self = 0 ifTrue: [thisContext explore. self halt. ↑ 1].
    self > 0 ifTrue: [↑ self * (self - 1) factorial].
    self error: 'Not valid for negative integers'
```

 Now evaluate `3 factorial` in a workspace. You should obtain both a debugger window and an explorer, as shown in la figure 14.8.

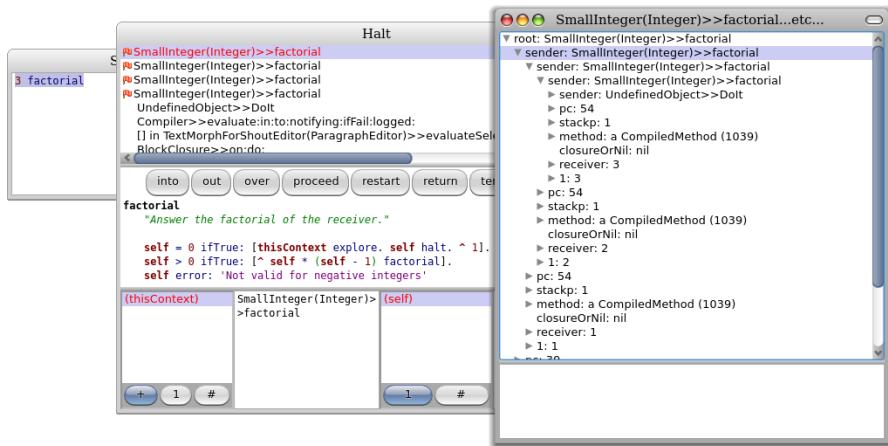


FIGURE 14.8 – Exploring thisContext.

Welcome to the poor-man's debugger ! If you now browse the class of the explored object (*c-à-d* by evaluating self browse in the bottom pane of the explorer) you will discover that it is an instance of the class MethodContext, as is each sender in the chain.

thisContext is not intended to be used for day-to-day programming, but it is essential for implementing tools like debuggers, and for accessing information about the call stack. You can evaluate the following expression to discover which methods make use of thisContext :

```
SystemNavigation default browseMethodsWithSourceString: 'thisContext'
```

As it turns out, one of the most common applications is to discover the sender of a message. Here is a typical application :

Object»subclassResponsibility

"This message sets up a framework for the behavior of the class' subclasses. Announce that the subclass should have implemented this message."

```
self error: 'My subclass should have overridden ', thisContext sender selector
printString
```

By convention, methods in Smalltalk that send self subclassResponsibility are considered to be abstract. But how does Object»subclassResponsibility provide a useful error message indicating which abstract method has been invoked ? Very simply, by asking thisContext for the sender.

Points d'arrêts intelligents

The Smalltalk way to set a breakpoint is to evaluate `self halt` at an interesting point in a method. This will cause `thisContext` to be reified, and a debugger window will open at the breakpoint. Unfortunately this poses problems for methods that are intensively used in the system.

Suppose, for instance, that we want to explore the execution of `OrderedCollection»add:`. Setting a breakpoint in this method is problematic.

 Take a fresh image and set the following breakpoint :

```
OrderedCollection»add: newObject
  self halt.
  ↑self addLast: newObject
```

Notice how your image immediately freezes! We do not even get a debugger window. The problem is clear once we understand that (i) `OrderedCollection»add:` is used by many parts of the system, so the breakpoint is triggered very soon after we accept the change, but (ii) *the debugger itself* sends `add:` to an instance of `OrderedCollection`, preventing the debugger from opening! What we need is a way to *conditionally halt* only if we are in a context of interest. This is exactly what `Object»haltIf:` offers.

Suppose now that we only want to halt if `add:` is sent from, say, the context of `OrderedCollectionTest»testAdd:`.

 Fire up a fresh image again, and set the following breakpoint :

```
OrderedCollection»add: newObject
  self haltIf : #testAdd.
  ↑self addLast: newObject
```

This time the image does not freeze. Try running the `OrderedCollectionTest`. (You can find it in the *CollectionsTests-Sequenceable* category.)

How does this work? Let's have a look at `Object»haltIf:`:

```
Object»haltIf: condition
| ctxt |
condition isSymbol ifTrue: [
  "only halt if a method with selector symbol is in callchain"
  ctxt := thisContext.
  [ctxt sender isNil] whileFalse: [
    ctxt := ctxt sender.
    (ctxt selector = condition) ifTrue: [Halt signal]. ].
  ↑self.
].
...
...
```

Starting from `thisContext`, `haltIf:` goes up through the execution stack, checking if the name of the calling method is the same as the one passed as parameter. If this is the case, then it raises an exception which, by default, summons the debugger.

It is also possible to supply a boolean or a boolean block as an argument to `haltIf:`, but these cases are straightforward and do not make use of `thisContext`

14.6 Intercepter les messages non compris

So far we have used the reflective features of Smalltalk mainly to query and explore objects, classes, methods and the run-time stack. Now we will look at how to use our knowledge of the Smalltalk system structure to intercept messages and modify behaviour at run-time.

When an object receives a message, it first looks in the method dictionary of its class for a corresponding method to respond to the message. If no such method exists, it will continue looking up the class hierarchy, until it reaches `Object`. If still no method is found for that message, the object will *send itself* the message `doesNotUnderstand:` with the message selector as its argument. The process then starts all over again, until `Object»doesNotUnderstand:` is found, and the debugger is launched.

But what if `doesNotUnderstand:` is overridden by one of the subclasses of `Object` in the lookup path ? As it turns out, this is a convenient way of realizing certain kinds of very dynamic behaviour. An object that does not understand a message can, by overriding `doesNotUnderstand:`, fall back to an alternative strategy for responding to that message.

Two very common applications of this technique are (1) to implement lightweight proxies for objects, and (2) to dynamically compile or load missing code.

Lightweight proxies

In the first case, we introduce a “minimal object” to act as a proxy for an existing object. Since the proxy will implement virtually no methods of its own, any message sent to it will be trapped by `doesNotUnderstand:`. By implementing this message, the proxy can then take special action before delegating the message to the real subject it is the proxy for.

Let us have a look at how this may be implemented¹.

We define a `LoggingProxy` as follows :

1. You can also load `PBE-Reflection` from <http://www.squeaksource.com/PharoByExample/>

```
ProtoObject subclass: #LoggingProxy
  instanceVariableNames: 'subject invocationCount'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE-Reflection'
```

Note that we subclass ProtoObject rather than Object because we do not want our proxy to inherit over 400 methods (!) from Object.

```
Object methodDict size → 408
```

Our proxy has two instance variables : the subject it is a proxy for, and a count of the number of messages it has intercepted. We initialize the two instance variables and we provide an accessor for the message count. Initially the subject variable points to the proxy object itself.

```
LoggingProxy»initialize
  invocationCount := 0.
  subject := self.
```

```
LoggingProxy»invocationCount
  ↑ invocationCount
```

We simply intercept all messages not understood, print them to the Transcript, update the message count, and forward the message to the real subject.

```
LoggingProxy»doesNotUnderstand: aMessage
  Transcript show: 'performing ', aMessage printString; cr.
  invocationCount := invocationCount + 1.
  ↑ aMessage sendTo: subject
```

Here comes a bit of magic. We create a new Point object and a new LoggingProxy object, and then we tell the proxy to become: the point object :

```
point := 1@2.
LoggingProxy new become : point.
```

This has the effect of swapping all references in the image to the point to now refer to the proxy, and vice versa. Most importantly, the proxy's subject instance variable will now refer to the point !

```
point invocationCount → 0
point + (3@4) → 4@6
point invocationCount → 1
```

This works nicely in most cases, but there are some shortcomings :

```
point class → LoggingProxy
```

Curiously, the method `class` is not even implemented in `ProtoObject` but in `Object`, which `LoggingProxy` does not inherit from ! The answer to this riddle is that class is never sent as a message but is directly answered by the virtual machine.²

Even if we can ignore such special message sends, there is another fundamental problem which cannot be overcome by this approach : self-sends cannot be intercepted :

```
point := 1@2.
LoggingProxy new become: point.
point invocationCount → 0
point rect: (3@4) → 1@2 corner: 3@4
point invocationCount → 1
```

Our proxy has been cheated out of two self-sends in the `rect:` method :

```
Point»rect: aPoint
  ↑ Rectangle origin: (self min: aPoint) corner: (self max: aPoint)
```

Although messages can be intercepted by proxies using this technique, one should be aware of the inherent limitations of using a proxy. In la section 14.7 we will see another, more general approach for intercepting messages.

Générer des méthodes manquantes

The other most common application of intercepting not understood messages is to dynamically load or generate the missing methods. Consider a very large library of classes with many methods. Instead of loading the entire library, we could load a stub for each class in the library. The stubs know where to find the source code of all their methods. The stubs simply trap all messages not understood, and dynamically load the missing methods on-demand. At some point, this behaviour can be deactivated, and the loaded code can be saved as the minimal necessary subset for the client application.

Let us look at a simple variant of this technique where we have a class that automatically adds accessors for its instance variables on-demand :

2. yourself is also never truly sent. Other messages that may be directly interpreted by the VM, depending on the receiver, include : + - < > <= >= ~ = * / == @ bitShift: // bitAnd: bitOr: at: at:put: size next nextPut: atEnd blockCopy: value value: do: new new: x y. Selectors that are never sent, because they are inlined by the compiler and transformed to comparison and jump bytecodes : 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: Attempts to send these messages to non-boolean objects can be intercepted and execution can be resumed with a valid boolean value by overriding `mustBeBoolean` in the receiver or by catching the `NonBooleanReceiver` exception.

```

DynamicAccessors»doesNotUnderstand: aMessage
| messageName |
messageName := aMessage selector asString.
(self class instVarNames includes: messageName)
ifTrue: [
    self class compile: messageName, String cr, '↑ ', messageName.
    ↑ aMessage sendTo: self].
↑ super doesNotUnderstand: aMessage

```

Any message not understood is trapped here. If an instance variable with the same name as the message sent exists, then we ask our class to compile an accessor for that instance variables and we re-send the message.

Suppose the class DynamicAccessors has an (uninitialized) instance variable *x* but no pre-defined accessor. Then the following will generate the accessor dynamically and retrieve the value :

```

myDA := DynamicAccessors new.
myDA x → nil

```

Let us step through what happens the first time the message *x* is sent to our object (see la figure 14.9).

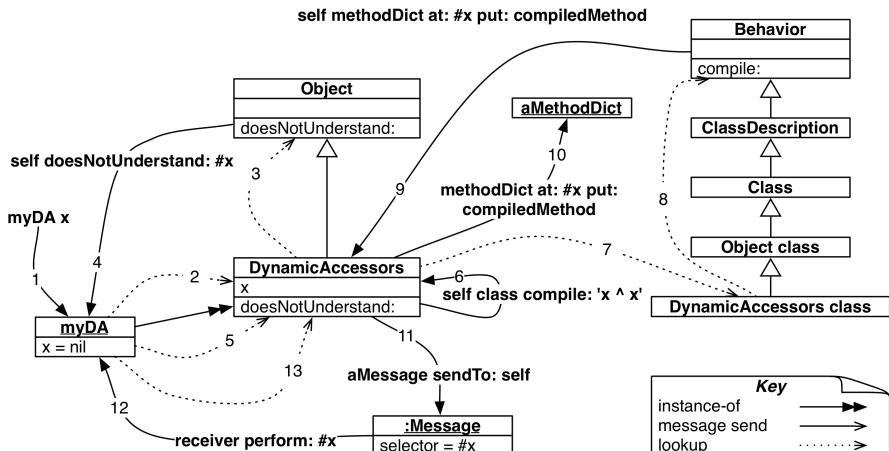


FIGURE 14.9 – Dynamically creating accessors.

(1) We send *x* to *myDA*, (2) the message is looked up in the class, and (3) not found in the class hierarchy. (4) This causes `self doesNotUnderstand: #x` to be sent back to the object, (5) triggering a new lookup. This time `doesNotUnderstand: #x` is found immediately in *DynamicAccessors*, (6) which asks its class to compile the string '*x* ↑ *x*'. The `compile` method is looked up (7), and (8) finally found in *Behavior*, which (9-10) adds the new compiled method

to the method dictionary of DynamicAccessors. Finally, (11-13) the message is resent, and this time it is found.

The same technique can be used to generate setters for instance variables, or other kinds of boilerplate code, such as visiting methods for a Visitor.

Note the use of Object»perform: in step (13) which can be used to send messages that are composed at run-time :

5 perform: #factorial	→	120
6 perform: ('fac', 'atorial') asSymbol	→	720
4 perform: #max: withArguments: (Array with: 6)	→	6

14.7 Objects as method wrappers

We have already seen that compiled methods are ordinary objects in Smalltalk, and they support a number of methods that allow the programmer to query the run-time system. What is perhaps a bit more surprising, is that *any object* can play the role of a compiled method. All it has to do is respond to the method `run:with:in:` and a few other important messages.

 Define an empty class Demo. Evaluate Demo new answer42 and notice how the usual "Message Not Understood" error is raised.

Now we will install a plain Smalltalk object in the method dictionary of our Demo class.

 Evaluate Demo methodDict at : #answer42 put : ObjectsAsMethodsExample new. Now try again to print the result of Demo new answer42. This time we get the answer 42.

If we take look at the class ObjectsAsMethodsExample we will find the following methods :

```
answer42
↑42

run: oldSelector with: arguments in: aReceiver
↑self perform: oldSelector withArguments: arguments
```

When our Demo instance receives the message `answer42`, method lookup proceeds as usual, however the virtual machine will detect that in place of a compiled method, an ordinary Smalltalk object is trying to play this role. The VM will then send this object a new message `run:with:in:` with the original method selector, arguments and receiver as arguments. Since `ObjectsAsMethodsExample` implements this method, it intercepts the message and delegates it to itself.

We can now remove the fake method as follows :

```
Demo methodDict removeKey: #answer42 ifAbsent: []
```

If we take a closer look at `ObjectsAsMethodsExample`, we will see that its superclass also implements the methods `flushcache`, `methodClass:` and `selector:`, but they are all empty. These messages may be sent to a compiled methods, so they need to be implemented by an object pretending to be a compiled method. (`flushcache` is the most important method to be implemented ; others may be required depending on whether the method is installed using `Behavior»addSelector:withMethod:` or directly using `MethodDictionary»at:put:..`)

Using methods wrappers to perform test coverage

Method wrappers are a well-known technique for intercepting messages³. In the original implementation⁴, a method wrapper is an instance of a subclass of `CompiledMethod`. When installed, a method wrapper can perform special actions before or after invoking the original method. When uninstalled, the original method is returned to its rightful position in the method dictionary.

In Pharo, method wrappers can be implemented more easily by implementing `run:with:in:` instead of by subclassing `CompiledMethod`. In fact, there exists a lightweight implementation of objects as method wrappers⁵, but it is not part of standard Pharo at the time of this writing.

Nevertheless, the Pharo Test Runner uses precisely this technique to evaluate test coverage. Let's have a quick look at how it works.

The entry point for test coverage is the method `TestRunner»runCoverage` :

```
TestRunner»runCoverage
| packages methods |
... "identify methods to check for coverage"
self collectCoverageFor: methods
```

The method `TestRunner»collectCoverageFor:` clearly illustrates the coverage checking algorithm :

```
TestRunner»collectCoverageFor: methods
| wrappers suite |
wrappers := methods collect: [ :each | TestCoverage on: each ].
suite := self
reset;
```

3. John Brant *et al.*, Wrappers to the Rescue. dans Proceedings European Conference on Object Oriented Programming (ECOOP'98). Volume 1445, Springer-Verlag 1998.

4. <http://www.squeaksource.com/MethodWrappers.html>

5. <http://www.squeaksource.com/ObjectsAsMethodsWrap.html>

```

suiteAll.
[ wrappers do: [ :each | each install ].
[ self runSuite: suite ] ensure: [ wrappers do: [ :each | each uninstall ] ]
    valueUnpreemptively.
wrappers := wrappers reject: [ :each | each hasRun ].
wrappers isEmptyy
    ifTrue:
        [ UIManager default inform: 'Congratulations. Your tests cover all code under
analysis.' ]
    ifFalse: ...

```

A wrapper is created for each method to be checked, and each wrapper is installed. The tests are run, and all wrappers are uninstalled. Finally the user obtains feedback concerning the methods that have not been covered.

How does the wrapper itself work ? The TestCoverage wrapper has three instance variables, hasRun, reference and method. They are initialized as follows :

```

TestCoverage class»on: aMethodReference
    ↑ self new initializeOn: aMethodReference

TestCoverage»initializeOn: aMethodReference
    hasRun := false.
    reference := aMethodReference.
    method := reference compiledMethod

```

The install and uninstall methods simply update the method dictionary in the obvious way :

```

TestCoverage»install
    reference actualClass methodDictionary
        at: reference methodSymbol
        put: self

TestCoverage»uninstall
    reference actualClass methodDictionary
        at: reference methodSymbol
        put: method

```

and the run:with:in: method simply updates the hasRun variable, uninstalls the wrapper (since coverage has been verified), and resends the message to the original method

```

run: aSelector with: anArray in: aReceiver
    self mark; uninstall.
    ↑ aReceiver withArgs: anArray executeMethod: method

mark

```

```
hasRun := true
```

(Have a look at `ProtoObject»withArgs:executeMethod:` to see how a method displaced from its method dictionary can be invoked.)

That's all there is to it !

Method wrappers can be used to perform any kind of suitable behaviour before or after the normal operation of a method. Typical applications are instrumentation (collecting statistics about the calling patterns of methods), checking optional pre- and post-conditions, and memoization (optionally caching computed values of methods).

14.8 Pragmas

Un *pragma* est une annotation qui donne des information sur un programme, mais qui n'est pas directement impliqué dans l'exécution de ce programme. Les pragmas n'ont pas d'effet direct lors du déroulement d'une méthode annotée. Les pragmas sont très utiles notamment pour :

- Donner de l'information au compilateur : les pragmas peuvent être utilisés par le compilateur pour qu'une méthode appelle une fonction primitive. Cette fonction doit être définie par la machine virtuelle ou au moyen d'un greffon externe.
- Donner de l'information à l'exécution.

Les pragmas s'utilisent uniquement lors de la déclaration des méthodes d'un programme. Une méthode peut déclarer un ou plusieurs pragmas qui sont écrits avant toutes expressions Smalltalk. En réalité, un pragma défini une sorte de message statique avec des arguments qui sont des littéraux.

Nous avons déjà parlé brièvement des pragmas lorsque nous avons brièvement introduit la notion de primitives précédemment dans ce chapitre. Une primitive n'est rien moins qu'une déclaration de pragma. Consider <primitive: 73> as contained in `instVarAt:`. The pragma's selector is `primitive: and its arguments is an immediate literal value, 73.`

Le compilateur Smalltalk est probablement l'un des utilisateurs les plus important des pragmas. SUnit is another tool that makes use of annotations. SUnit is able to estimate the coverage of an application from a test unit. One may want to exclude some methods from the coverage. This is the case of the documentation method in `SplitJointTest` class :

```
SplitJointTest class»documentation
```

```
<ignoreForCoverage>
"self showDocumentation"
```

↑ This package provides function.... "

By simply annotating a method with the pragma `<ignoreForCoverage>` one can control the scope of the coverage.

As instances of the class `Pragma`, pragmas are first class objects. A compiled method answers to the message `pragmas`. This method returns an array of pragmas.

```
(SplitJoinTest class >> #showDocumentation) pragmas
    → an Array(<ignoreForCoverage>)
(Float>>#+) pragmas → an Array(<primitive: 41>)
```

Methods defining a particular query may be retrieved from a class. The class side of `SplitJoinTest` contains some methods annotated with `<ignoreForCoverage>`:

```
Pragma allNamed: #ignoreForCoverage in: SplitJoinTest class → an Array(<
    ignoreForCoverage> <ignoreForCoverage> <ignoreForCoverage>)
```

A variant of `allNamed:in:` may be found on the class side of `Pragma`.

A pragma knows in which method it is defined (using `method`), the name of the method (`selector`), the class that contains the method (`methodClass`), its number of arguments (`numArgs`), about the literals the pragma has for arguments (`hasLiteral:` and `hasLiteralSuchThat:`).

14.9 Résumé du chapitre

Reflection refers to the ability to query, examine and even modify the metaobjects of the run-time system as ordinary objects.

- The Inspector uses `instVarAt:` and related methods to query and modify “private” instance variables of objects.
- Send `Behavior>allInstances` to query instances of a class.
- The messages `class`, `isKindOf:`, `respondsTo:` etc are useful for gathering metrics or building development tools, but they should be avoided in regular applications : they violate the encapsulation of objects and make your code harder to understand and maintain.
- `SystemNavigation` is a utility class holding many useful queries for navigation and browsing the class hierarchy. For example, use `SystemNavigation default browseMethodsWithSourceString: 'pharo'.` to find and browse all methods with a given source string. (Slow, but thorough !)
- Every Smalltalk class points to an instance of `MethodDictionary` which maps selectors to instances of `CompiledMethod`. A compiled method knows its class, closing the loop.
- `MethodReference` is a lightweight proxy for a compiled method, providing additional convenience methods, and used by many Smalltalk tools.

- `BrowserEnvironment`, part of the Refactoring Browser infrastructure, offers a more refined interface than `SystemNavigation` for querying the system, since the result of a query can be used as the scope of a new query. Both GUI and programmatic interfaces are available.
- `thisContext` is a pseudo-variable that reifies the run-time stack of the virtual machine. It is mainly used by the debugger to dynamically construct an interactive view of the stack. It is also especially useful for dynamically determining the sender of a message.
- Intelligent breakpoints can be set using `haltIf:`, taking a method selector as its argument. `haltIf:` halts only if the named method occurs as a sender in the run-time stack.
- A common way to intercept messages sent to a given target is to use a “minimal object” as a proxy for that target. The proxy implements as few methods as possible, and traps all message sends by implementing `doesNotUnderstand:`. It can then perform some additional action and then forward the message to the original target.
- Send `become:` to swap the references of two objects, such as a proxy and its target.
- Beware, some messages, like `class` and `yourself` are never really sent, but are interpreted by the VM. Others, like `+`, `-` and `ifTrue:` may be directly interpreted or inlined by the VM depending on the receiver.
- Another typical use for overriding `doesNotUnderstand:` is to lazily load or compile missing methods.
- `doesNotUnderstand:` cannot trap self-sends.
- A more rigorous way to intercept messages is to use an object as a method wrapper. Such an object is installed in a method dictionary in place of a compiled method. It should implement `run:with:in:` which is sent by the VM when it detects an ordinary object instead of a compiled method in the method dictionary. This technique is used by the SUnit Test Runner to collect coverage data.

Quatrième partie

Annexes

Annexe A

Foire Aux Questions

A.1 Prémisses

FAQ 1 *Où puis-je trouver la dernière version de Pharo ?*

Réponse <http://pharo-project.org>

FAQ 2 *Quelle image de Pharo devrais-je utiliser avec ce livre ?*

Réponse «<<< .mine Vous pouvez utiliser n'importe quelle image Pharo mais nous vous recommandons d'utiliser l'image préparée sur le site web de Pharo Par l'Exemple : <http://PharoByExample.org/fr>. Vous risquez d'avoir des comportements surprenants lors de la saisie ===== Vous pouvez utiliser n'importe quelle image de Pharo mais nous vous recommandons d'utiliser l'image préparée sur le site web de Pharo par l'exemple : <http://PharoByExample.org/fr>. Vous risqueriez d'avoir des comportements surprenants lors de la saisie >>>> .r34584 des exercices en utilisant une autre image.

A.2 Collections

FAQ 3 *Comment puis-je trier une OrderedCollection ?*

Réponse Envoyez le message suivant asSortedCollection.

```
#(7 2 6 1) asSortedCollection —→ a SortedCollection(1 2 6 7)
```

FAQ 4 Comment puis-je convertir une collection de caractères en une chaîne de caractères String ?

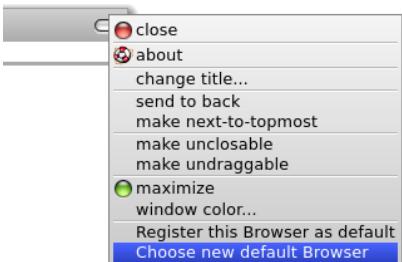
Réponse

```
String streamContents: [:str | str nextPutAll: 'hello' asSet] → 'hleo'
```

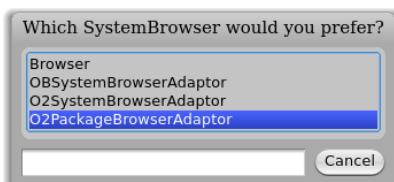
A.3 Naviguer dans le système

FAQ 5 Le navigateur de classes ne ressemble pas à celui décrit dans le livre. Que se passe-t-il ?

Réponse Vous utilisez probablement une image disposant d'une version différente d'OmniBrowser (abrégé en OB) installé comme Browser par défaut. Dans ce livre, nous présumons que le navigateur Omnibrowser Package Browser (navigateur par paquetages) est installé par défaut. Vous pouvez changer cela en cliquant sur la bulle grise (le bouton si vous voulez...) à droite de la barre de titre du navigateur, puis en sélectionnant dans le menu du Browser "Choose new default Browser" (en français, *choisissez le nouveau Browser par défaut*). Dans la liste des navigateurs proposés, cliquez sur O2PackageBrowserAdaptor. Le prochain navigateur de classes que vous ouvrirez sera le Package Browser.



(a) Choisir un nouveau Browser.



(b) Sélectionnez l'OB Package Browser

FIGURE A.1 – Changer le navigateur de classes par défaut.

FAQ 6 Comment puis-je chercher une classe ?

Réponse CMD-b (pour *browse c-à-d.* parcourir à l'aide du navigateur de classes) sur le nom de la classe ou CMD-f dans le panneau des catégories du Browser.

FAQ 7 Comment puis-je trouver/naviguer dans tous les envois à super ?

Réponse La deuxième solution est la plus rapide :

```
SystemNavigation default browseMethodsWithSourceString: 'super'.
SystemNavigation default browseAllSelect: [:method | method sendsToSuper ].
```

FAQ 8 Comment puis-je naviguer au travers de tous les envois de messages à super dans une hiérarchie ?

Réponse

```
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 Comment puis-je découvrir quelles sont les nouvelles méthodes implémentées dans une classe ?

Réponse Dans le cas présent nous demandons quelles sont les nouvelles méthodes introduites par True :

```
newMethods:= [:aClass| aClass methodDict keys select:
    [:aMethod | (aClass superclass canUnderstand: aMethod) not ]].
newMethods value: True   —> an IdentitySet(#asBit)
```

FAQ 10 Comment puis-je trouver les méthodes d'une classe qui sont abstraites ?

Réponse

```
abstractMethods:=
    [:aClass | aClass methodDict keys select:
        [:aMethod | (aClass>>aMethod) isAbstract ]].
abstractMethods value: Collection   —> an IdentitySet(#remove:ifAbsent: #add: #do:)
```

FAQ 11 Comment puis-je créer une vue de l'arbre syntaxique abstrait ou AST d'une expression ?

Réponse Charger le paquetage AST depuis <http://squeaksource.com>. Ensuite évaluer :

```
(RBParser parseExpression: '3+4') explore
```

(D'autre part *explorer le.*)

FAQ 12 *Comment puis-je trouver tout les traits dans le système ?*

Réponse

```
Smalltalk allTraits
```

FAQ 13 *Comment puis-je trouver quelles classes utilisent les traits ?*

Réponse

```
Smalltalk allClasses select: [:each | each hasTraitComposition ]
```

A.4 Utilisation de Monticello et de SqueakSource

FAQ 14 *Comment puis-je charger un projet du SqueakSource ?*

Réponse

1. Trouvez le projet que vous souhaitez sur <http://squeaksource.com>
2. Copiez le code d'enregistrement
3. Sélectionnez `open > Monticello browser`
4. Sélectionnez `+Repository > HTTP`
5. Collez et acceptez le code d'enregistrement ; entrez votre mot de passe
6. Sélectionnez le nouveau dépôt et ouvrez-le avec le bouton `Open`
7. Sélectionnez et chargez la version la plus récente

FAQ 15 *Comment puis-je créer un projet SqueakSource ?*

Réponse

1. Allez à <http://squeaksource.com>
2. Enregistrez-vous comme un nouveau membre
3. Enregistrez un projet (nom = catégorie)
4. Copiez le code d'enregistrement
5. `open > Monticello browser`
6. `+Package` pour ajouter une catégorie
7. Sélectionnez le package
8. `+Repository > HTTP`
9. Collez et acceptez le code d'enregistrement ; entrez votre mot de passe
10. `Save` pour enregistrer la première version

FAQ 16 *Comment puis-je étendre Number avec la méthode Number»chf tel que Monticello la reconnaissent comme étant une partie de mon projet Money ?*

Réponse Mettez-la dans une catégorie de méthodes nommée `*Money`. Monticello réunit toutes les méthodes dont les noms de catégories ont la forme `*package` et les insére dans votre package.

A.5 Outils

FAQ 17 *Comment puis-je ouvrir de manière pragmatique le SUnit TestRunner ?*

Réponse Évaluez `TestRunner open`.

FAQ 18 *Où puis-je trouver le Refactoring Browser ?*

Réponse Chargez le paquetage AST puis le moteur de refactorisation sur le site <http://squeaksource.com> :

<http://www.squeaksource.com/AST>

<http://www.squeaksource.com/RefactoringEngine>

FAQ 19 *Comment puis-je enregistrer le navigateur comme navigateur par défaut ?*

Réponse Cliquez sur l'icône (une bulle grise) du menu situé à droite dans la barre de titre de la fenêtre du Browser. Choisissez `Register this Browser as default` pour enregistrer le navigateur courant comme navigateur par défaut ou bien, sélectionnez `Choose new default Browser` pour obtenir un menu flottant d'où vous pourrez faire votre choix parmi les différentes classes de Browser.

A.6 Expressions régulières et analyse grammaticale

FAQ 20 *Où est la documentation pour le paquetage RegEx ?*

Réponse Regardez dans le protocole DOCUMENTATION de RxParser class situé dans la catégorie VB-Regex .

FAQ 21 *Y a-t'il des outils pour l'écriture d'un outil d'analyse grammaticale ?*

Réponse Utilisez SmaCC — le compilateur de compilateur (ou générateur de compilateur)¹ Smalltalk. Vous devrez installer au moins SmaCC-lr.13. Chargez-le depuis <http://www.squeaksource.com/SmaccDevelopment.html>. Il y a un bon tutoriel en ligne à l'adresse : <http://www.refactory.com/Software/SmaCC/Tutorial.html>

FAQ 22 *Quels paquetages devrais-je charger depuis SqueakSource SmaccDevelopment pour écrire un analyseur grammatical ?*

Réponse Chargez la dernière version de SmaccDev — le lanceur de programme est déjà actif. (Attention : SmaccDevelopment est destiné à la version 3.8 de Squeak)

1. En anglais, Compiler-Compiler.

Bibliographie

- Sherman R. Alpert, Kyle Brown et 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. dans 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 et David Robson:** Smalltalk 80 : the Language and its Implementation. Reading, Mass.: Addison Wesley, mai 1983, ISBN 0-201-13688-0
- Dan Ingalls et al.:** Back to the Future : The Story of Squeak, a Practical Smalltalk Written in Itself. dans Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97). ACM Press, novembre 1997 ⟨URL: [http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html

Wilf LaLonde et 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. dans **Robert Martin, Dirk Riehle et Frank Buschmann \\(éd.\\):** Pattern Languages of Program Design 3. Addison Wesley, 1998, 5–18\]\(http://stephane.ducasse.free.fr/FreeBooks/ByExample/\)](http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html)

