

# Squeak par l'Exemple

Andrew Black   Stéphane Ducasse

Oscar Nierstrasz   Damien Pollet

avec l'aide de Damien Cassou et Marcus Denker

*Version du 2007-11-24*

Ce livre est disponible en libre téléchargement depuis [scg.unibe.ch/SBE](http://scg.unibe.ch/SBE).

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

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

*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) : [creativecommons.org/licenses/by-sa/3.0/legalcode](http://creativecommons.org/licenses/by-sa/3.0/legalcode)

# Table des matières

## I Comment démarrer

<b>1</b>	<b>Une visite de Squeak</b>	<b>3</b>
1.1	Premiers pas. . . . .	3
1.2	Le menu World. . . . .	8
1.3	Enregistrer, quitter et redémarrer une session Squeak.. .	10
1.4	Les Workspaces et les fenêtres Transcripts . . . . .	12
1.5	Keyboard shortcuts . . . . .	14
1.6	SqueakMap . . . . .	17
1.7	The System Browser . . . . .	19
1.8	Finding classes . . . . .	20
1.9	Finding methods . . . . .	23
1.10	Defining a new method. . . . .	25
1.11	Chapter summary. . . . .	30
<b>2</b>	<b>Une première application</b>	<b>31</b>
2.1	Le jeu de Quinto . . . . .	31
2.2	Créer une nouvelle catégorie de classe . . . . .	32
2.3	Définir la classe SBECcell . . . . .	33

2.4	Ajouter des méthodes à la classe . . . . .	35
2.5	Inspecter un objet . . . . .	38
2.6	Définir la classe SBEGame . . . . .	39
2.7	Organiser les méthodes en protocoles . . . . .	43
2.8	Essayons notre code . . . . .	48
2.9	Sauvegarder et partager le code Smalltalk . . . . .	51
2.10	Résumé du chapitre . . . . .	56
<b>3</b>	<b>Un résumé de la syntaxe</b>	<b>59</b>
3.1	Les éléments syntaxiques . . . . .	60
3.2	Les pseudo-variables . . . . .	63
3.3	Envois de messages . . . . .	64
3.4	Syntaxe relative aux méthodes . . . . .	66
3.5	La syntaxe des blocs . . . . .	67
3.6	Un résumé des Tests et des Itérations . . . . .	69
3.7	Primitives et Pragmas . . . . .	71
3.8	Résumé du chapitre . . . . .	72
<b>4</b>	<b>Comprendre la syntaxe des messages</b>	<b>75</b>
4.1	Identifier les messages . . . . .	75
4.2	Trois sortes de messages . . . . .	78
4.3	Composition de messages . . . . .	81
4.4	Quelques astuces pour identifier les messages à mots-clés . . . . .	88
4.5	Séquences d'expression . . . . .	91
4.6	Cascades de messages . . . . .	91
4.7	Résumé du chapitre . . . . .	92

## II Développer avec Squeak

<b>5</b>	<b>Le modèle objet de Smalltalk</b>	<b>95</b>
5.1	Les règles du modèle . . . . .	95
5.2	Tout est un objet . . . . .	96
5.3	Tout objet est instance de classe . . . . .	97
5.4	Toute classe a une super-classe . . . . .	107
5.5	Tout se passe par envoi de messages . . . . .	112
5.6	La recherche de méthode suit la chaîne d'héritage . . . . .	114
5.7	Les variables partagées . . . . .	122
5.8	Résumé du chapitre . . . . .	129
<b>6</b>	<b>L'environnement de programmation de Squeak</b>	<b>131</b>
6.1	Une vue générale . . . . .	132
6.2	Le System Browser . . . . .	133
6.3	Monticello . . . . .	151
6.4	L'inspecteur Inspector et l'explorateur Explorer . . . . .	161
6.5	Debugger, le débogueur . . . . .	165
6.6	Le navigateur de processus . . . . .	176
6.7	Trouver les méthodes . . . . .	178
6.8	Change set et son gestionnaire Change Sorter . . . . .	179
6.9	Le navigateur de fichiers File List Browser . . . . .	183
6.10	En Smalltalk, pas de perte de codes . . . . .	185
6.11	Résumé du chapitre . . . . .	187
<b>7</b>	<b>SUnit</b>	<b>191</b>
7.1	Introduction . . . . .	191
7.2	Pourquoi le test est important . . . . .	193
7.3	De quoi est fait un bon test ? . . . . .	194
7.4	SUnit par l'exemple . . . . .	195
7.5	Les recettes pour SUnit . . . . .	200

7.6	L'environnement SUnit . . . . .	.202
7.7	Caractéristiques avancées de SUnit . . . . .	.205
7.8	La mise en œuvre de SUnit . . . . .	.207
7.9	Quelques conseils sur le test . . . . .	.211
7.10	Résumé du chapitre . . . . .	.212
<b>8</b>	<b>Les classes de base</b>	<b>215</b>
8.1	Object . . . . .	.215
8.2	Numbers . . . . .	.226
8.3	Characters . . . . .	.231
8.4	Strings . . . . .	.232
8.5	Booleans . . . . .	.234
8.6	Chapter summary. . . . .	.235
<b>9</b>	<b>Les collections</b>	<b>237</b>
9.1	Introduction . . . . .	.237
9.2	Les variétés de collections . . . . .	.238
9.3	Les implémentations des collections . . . . .	.243
9.4	Exemples de classes importantes . . . . .	.244
9.5	Les collections itératrices ou iterators . . . . .	.258
9.6	Astuces pour tirer profit des collections . . . . .	.263
9.7	Résumé du chapitre . . . . .	.265
<b>10</b>	<b>Stream : les flux de données</b>	<b>267</b>
10.1	Deux séquences d'éléments . . . . .	.267
10.2	Streams contre Collections. . . . .	.269
10.3	Utiliser les streams avec les collections . . . . .	.270
10.4	Utiliser les streams pour accéder aux fichiers . . . . .	.280
10.5	Résumé du chapitre . . . . .	.283

### III Squeak avancé

<b>11</b>	<b>Classes et méta-classes</b>	<b>287</b>
11.1	Les règles pour les classes et les méta-classes . . . .	.287
11.2	Retour sur le modèle objet de Smalltalk . . . . .	.289
11.3	Toute classe est une instance d'une méta-classe . . . .	.291
11.4	La hiérarchie des méta-classes est parallèle à celle des classes . . . . .	.292
11.5	Toute méta-classe hérite de <code>Class</code> et de <code>Behavior</code> . . .	.295
11.6	Toute méta-classe est une instance de <code>Metaclass</code> . . .	.298
11.7	La méta-classe de <code>Metaclass</code> est une instance de <code>Metaclass</code>	.299
11.8	Résumé du chapitre . . . . .	.301

### IV Annexes

<b>A</b>	<b>Foire Aux Questions</b>	<b>305</b>
A.1	Prémices . . . . .	.305
A.2	Collections . . . . .	.305
A.3	Naviguer dans le système . . . . .	.306
A.4	Utilisation de Monticello et de SqueakSource . . . .	.308
A.5	Outils . . . . .	.309
A.6	Expressions régulières et analyse grammaticale . . .	.310





Première partie


# **Comment démarrer**



# Chapitre 1

## Une visite de Squeak

Nous vous proposons dans ce chapitre une visite de Squeak afin de vous familiariser avec son environnement. De nombreux aspects seront abordés, il est conseillé d'avoir une machine prête pour suivre ce chapitre.

Cet icône  signalera dans le texte les étapes où vous devrez essayer quelque chose vous-même. Vous apprendrez à démarrer Squeak, 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

Squeak est librement disponible depuis le site principal de Squeak : [www.squeak.org](http://www.squeak.org). Vous devez y télécharger 3 archives (pour 4 fichiers principaux qui constituent une installation courante de Squeak)

1. La *machine virtuelle* (VM) est la seule partie de l'environnement qui est particulière à chaque système d'exploitation. Des machines virtuelles pré-compilées sont disponibles pour la plupart des systèmes (Linux, OS/X, Win32). Dans la figure 1.1 vous remarquerez par exemple la machine virtuelle pour le Macintosh :



FIG. 1.1 – Téléchargement de Squeak.

*Squeak 3.8.15beta1U.app.*

2. Le fichier *source* contient le code source du système Squeak qui change peu. Dans la la figure 1.1 il correspond à *SqueakV39.sources*. Le fichier source *SqueakV39.sources* est destiné aux versions 3.9 ou supérieures de Squeak. Pour des versions antérieures vous devez utiliser un fichier source (par exemple *SqueakV39.sources*) de même version que Squeak (de 3.0 à 3.8).
3. Le fichier *image* est un cliché d'un système 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 ainsi que les classes et les méthodes puisque ce sont aussi des objets. Le second avec l'extention *.changes* contient les changements apportés au code source, ils y sont journalisés.

### Téléchargement et installation de Squeak.

Dans ce livre et pour tous les exemples nous avons utilisé *Squeak-dev*, cette image est disponible sur <http://damien.cassou.free.fr/squeak-dev>.

Elle contient une large collection d'outils de développement et permet d'installer très facilement des paquets complémentaires.

Si vous avez déjà une autre version de Squeak qui fonctionne sur votre machine, la plupart des exemples d'introduction de ce livre fonctionneront et il n'est pas nécessaire de mettre à jour Squeak.

Mais vous constaterez peut-être des différences dans l'apparence ou le comportement que nous décrirons.


De toute façon, si vous téléchargez Squeak pour la première fois, vous devriez utiliser l'image *Squeak-dev*.

Pendant que vous travaillez avec Squeak les fichiers *.image* et *.changes* sont modifiés, vous devez vous assurer qu'ils sont accessibles en écriture. Conservez toujours ces deux fichiers ensemble, c'est-à-dire dans le même dossier. Et surtout, ne tentez pas de les modifier avec un éditeur de texte, Squeak les utilise pour stocker vos objets de travail et vos changements dans le code source. Sauvegardez vos images téléchargées et vos fichiers *changes*, vous pourrez ainsi toujours démarrer à partir d'une image propre et recharger votre code.


Les fichiers *sources* et l'exécutable de la VM peuvent être en lecture seul, il est donc possible de les partager pour 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 et distinct. Vous pouvez adapter l'installation de Squeak à vos habitudes de travail et votre système d'exploitation.

**Lancement.** Pour lancer Squeak, selon votre système : glissez un le fichier *.image* sur l'icône de l'exécutable de la machine virtuelle, sinon double-cliquez sur le fichier *.image*, ou depuis une ligne de commande tapez le nom de l'exécutable de la machine virtuelle suivi par le 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 travailler par glisser-déposer sur la VM ou d'utiliser la ligne de commande)

Une fois lancé, Squeak vous présente une large fenêtre qui contient des espaces de travail (voir la figure ??). Notez qu'il n'y a pas de barre de menu, à la place Squeak utilise des menus contextuels.

 Lancez Squeak. Vous pouvez fermer des *Workspace* (espaces de travail) en cliquant sur l'icône X au coin supérieur gauche des fenêtres ou les replier en cliquant sur le symbole O au coin supérieur droit.

**Première interaction** Une bon point de départ est le contenu du menu World (Monde) présenté dans la figure 1.2 (a).

 Cliquez à l'aide de la souris dans l'arrière plan de la fenêtre principale pour afficher le menu World, puis sélectionnez `open ... ▸ workspace` pour

créer un nouveau Workspace.

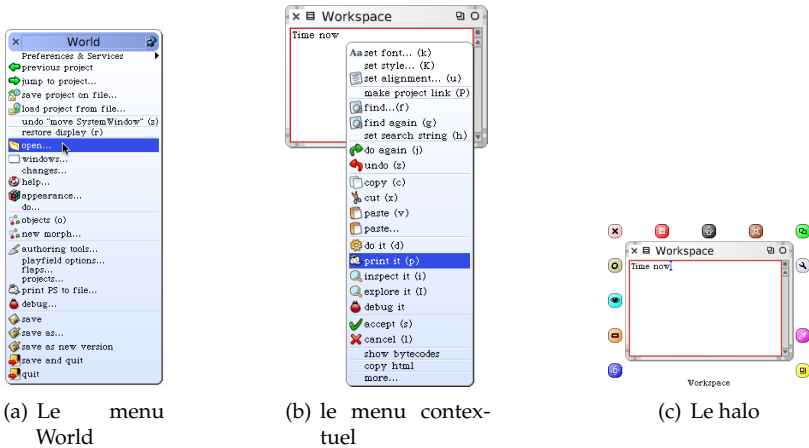


FIG. 1.2 – Le menu World (affiché avec le bouton rouge de la souris), un menu contextuel (bouton jaune de la souris), et un halo halo (bouton bleu).

Squeak 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 clavier en complément de la souris pour simuler les boutons manquants. Une souris à deux boutons fonctionne bien avec Squeak, mais si la votre 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 Squeak n'en sera que plus agréable.


Squeak évite les termes “clic gauche” ou “clic droit” car leurs effets peuvent varier selon les systèmes ou les réglages utilisateur. Squeak désigne les boutons avec des couleurs. Le bouton avec lequel vous obtenez le menu World est intitulé *bouton rouge*, il est également employé pour sélection-





FIG. 1.3 – La souris de l'auteur. La molette correspond au bouton bleu.

ner du texte, des choix de menus ou déplacer des fenêtres. Lors de vos premiers pas avec Squeak il vous sera sûrement utile d'utiliser cette référence de couleurs avec votre souris, comme montré sur la la figure 1.3.

Le *bouton jaune* est l'autre bouton le plus employé dans Squeak, vous l'utiliserez pour afficher les menus contextuels qui présentent des options selon le contexte ou plus précisément selon l'endroit et les objets sur lesquels vous cliquez. Voir la la figure 1.2 (b).

 Entrez *Time now* dans le *workspace*. Puis cliquez avec le bouton jaune dans le *workspace*. Et dans le menu qui apparaît sélectionnez `print it`.


Enfin, le *bouton bleu* est utilisé pour activer le menu “halo”. Ce menu est présenté sous la forme d'une collection de poignées autour de l'objet actif de l'écran et qui permettent d'en changer la taille, de le faire pivoter. Voir la figure 1.2 (c). En survolant lentement les poignées avec le pointeur de votre souris, une bulle d'aide en affichera un descriptif.

 Cliquez avec le bouton bleu sur le *Workspace*. Et déplacez la poignée  (située à proximité du coin inférieur gauche) pour faire pivoter le *Workspace*.

Nous recommandons aux personnes gauchères de configurer leur souris et d'affecter le bouton rouge à la gauche de leur souris, le bouton jaune à droite et d'utiliser la molette de défilement (si elle est disponible) comme bouton bleu. Avec une souris sans molette il est possible d'invoquer le menu halo en maintenant `alt`, `ctrl` ou `option` pendant que vous cliquez sur le bouton rouge. Si vous utilisez un Macintosh avec une souris à un bouton, vous pouvez simuler le second bouton en maintenant la touche `⌘` enfoncée et en cliquant. Si vous prévoyez d'utiliser Squeak souvent, nous vous recommandons d'investir dans un modèle à deux boutons.

Vous pouvez configurer votre souris selon vos souhaits en utilisant les préférences de votre système ou le pilote de votre dispositif de pointage. Squeak vous propose des réglages pour adapter votre souris et les touches spéciales de votre clavier. Vous trouverez le *Preference Browser* dans l'option `open` du menu `World`. Dans le *Preference Browser*, la catégorie `general` contient une option `swapMouseButtons` qui

permuter les boutons jaune et bleu (voir `swapMouseButtons`). Le `keyboard` a une catégorie pour dupliquer les touches de commandes et rendre équivalent une pression sur alt à une pression sur ctrl.

 Ouvrez le *Preference Browser* en cliquant avec le bouton rouge dans l'arrière plan de la fenêtre de Squeak et cherchez l'option `swapMouseButtons` en utilisant la zone de recherche.

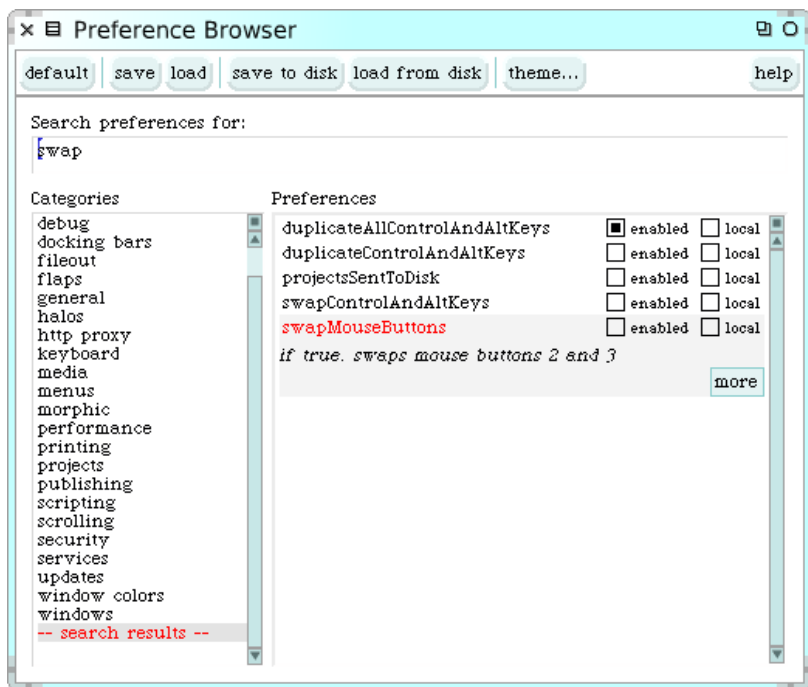



FIG. 1.4 – Le Preference Browser.

## 1.2 Le menu World

 Cliquez avec le bouton rouge dans l'arrière plan de Squeak.



Le menu **World** apparaît à nouveau.

La plupart des menus de Squeak ne sont pas modaux comme une fenêtre de dialogue pour la sauvegarde d'un fichier que vous devez soit compléter, soit annuler, sans pouvoir travailler en même temps dans l'application. Avec Squeak vous pouvez maintenir ces menus sur l'écran en cliquant sur l'icône en forme d'épingle au coin supérieur droit. Essayez ! Vous remarquerez que les menus apparaissent quand vous cliquez mais ne disparaissent pas quand vous relâchez votre bouton, ils restent visibles jusqu'à que vous ayez fait une sélection ou cliqué en dehors du menu. Et tous les menus affichés à l'écran peuvent se déplacer en glissant leur barre de titre, comme n'importe quelle fenêtre.

Le menu World vous offre un moyen d'accéder à la plupart des outils de Squeak.

 Regardez attentivement le menu `world▷open ...`.

Vous verrez les principaux outils de Squeak, et surtout le System Browser (l'un des nombreux navigateurs de classes) et le Workspace. Nous aurons affaire à eux dans les prochains chapitres.

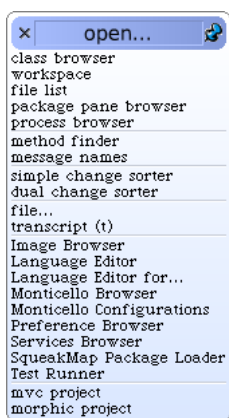


FIG. 1.5 – L'option `open ...` dans le menu World.

### 1.3 Enregistrer, quitter et redémarrer une session Squeak.


 Affichez le menu **World**, puis sélectionnez **new morph ...** et défilez jusqu'à **from alphabetical list > A-C > BlobMorph**. Vous avez maintenant un **Blob (Forme)** "en main". Positionnez-le où vous le souhaitez (bouton rouge). Votre forme s'animera.



FIG. 1.6 – Une instance d'un BlobMorph.

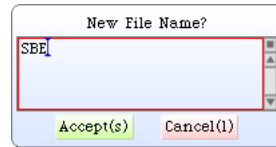




FIG. 1.7 – Le choix de menu **save as ...**.

 Sélectionnez **World > save as ...** et entrez le nom "SBE", puis cliquez sur le bouton **Accept(s)**. Pour finir, sélectionnez **World > save and quit**.

Le dossier qui contenait les fichiers `.image` et `.changes` lorsque vous avez lancé cette session de travail avec Squeak contient maintenant deux nouveaux fichiers : "SBE.image" and "SBE.changes". Ils contiennent l'image "vivante" de votre session Squeak 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 être utilisés plus tard. A vous de les invoquer depuis un lien, un glisser-déposer sur la machine virtuelle ou à partir de la ligne de commande.

 Lancez Squeak avec cette image que vous venez de créer : "SBE.image" file.

Vous retrouvez l'état de votre session exactement tel qu'il était avant que vous quittiez Squeak. Le Blob est toujours sur votre fenêtre

de travail, en train de se déplacer, au pixel près comme vous l'avez abandonné.


En lançant pour la première fois Squeak, 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 Squeak, 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.* ramasse-miettes)..

En quittant Squeak vous sauvegardez un instantané (image) de tous les objets, les vôtres et ceux de Squeak. 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 auriez 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 Squeak 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 on considère comme une pratique plutôt brusque de les employer pour diffuser et partager vos projets alors qu'il existe des outils tel que Monticello qui proposent de biens meilleurs moyens de suivre les évolutions du code et de le partager entre plusieurs développeurs.

 Cliquez avec le bouton bleu sur le Blob

Vous verrez tout autour une collection d'icônes colorées, on les appelle *Handle*(Poignées). Cliquez sur la poignée rose qui contient une

croix ; le Blob disparaît.

(Pour réussir cette manipulation, vous devrez peut-être faire plusieurs tentatives car le Blob se déplace et peut fuir votre souris et vous empêcher de cliquer à l'endroit attendu.)

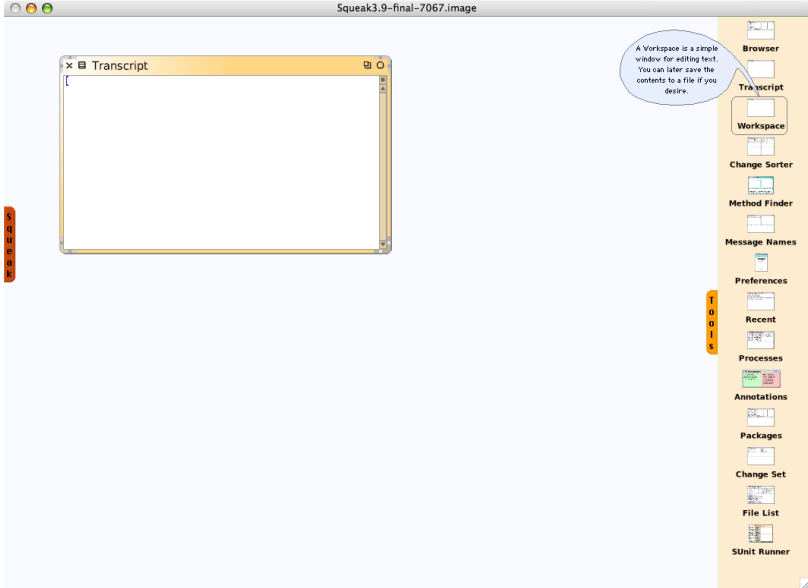




FIG. 1.8 – The Squeak l'onglet *Tools*.

## 1.4 Les Workspaces et les fenêtres Transcripts

 Fermez toutes fenêtres. Cliquez sur l'onglet **Tools** à la droite de la fenêtre principale de Squeak pour ouvrir le volet des outils (Tools Flap).


L'onglet s'élargira et présentera les icônes de certains outils importants de Squeak (la figure 1.8). Glissez alors l'icône Transcript puis l'icône Workspace.

 *Positionnez et redimensionnez le Transcript et le Workspace pour que dernier recouvre le Transcript.*

Vous pouvez redimensionner les fenêtres en glissant l'un de leurs coins, ou avec le bouton bleu qui affiche les poignées. Utilisez alors l'icône jaune située en bas à droite.

Une seule fenêtre est active à la fois, son titre est alors affiché en gras. Et surtout le pointeur de la souris doit être dans la fenêtre si vous devez entrer du texte.

Le Transcript est un objet qui est couramment utilisé pour afficher des messages du système. C'est un genre de "console". L'affichage dans la fenêtre Transcript est très lent, si vous la conservez ouverte et que vous affichez le résultat de certaines opérations celles-ci peuvent ralentir plus de 10 fois. De plus la fenêtre Transcript n'est pas conçue pour recevoir simultanément des messages à afficher provenant de plusieurs objets.

 *Type the following text into the workspace :*

---

Transcript show: 'hello world'; cr.

---

Try double-clicking in the workspace at various points in the text you have just typed. Notice how an entire word, entire string, or the whole text is selected, depending on where you click.

 *Select the text you have typed and yellow-click. Select **do it (d)**.*

Notice how the text "hello world" appears in the transcript window (la figure 1.9). Do it again. (The **(d)** in the menu item **do it (d)** tells you that the keyboard shortcut to *do it* is CMD-d. More on this in the next section !)

You have just evaluated your first Smalltalk expression ! You just sent the message show: 'hello world' to the Transcript object, followed by the message cr (carriage return). The Transcript then decided what to do with this message, that is, it looked up its *methods* for handling show: and cr messages and reacted appropriately.

If you talk to Smalltalkers for a while, you will quickly notice that they generally do not use expressions like "call an operation"

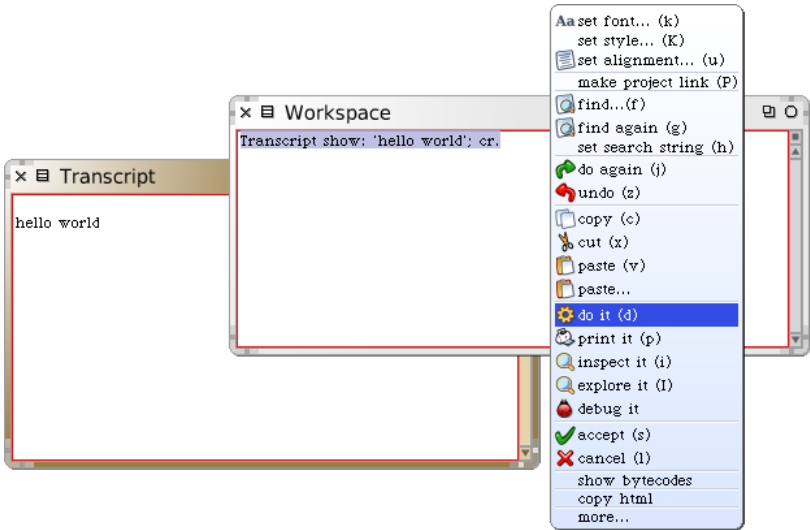


FIG. 1.9 – “Doing” an expression

or “invoke a method”, but instead they will say “send a message”. This reflects the idea that objects are responsible for their own actions. You never *tell* an object what to do—instead you politely *ask* it to do something by sending it a message. The object, not you, selects the appropriate method for responding to your message.

## 1.5 Keyboard shortcuts


If you want to evaluate an expression, you do not always have to bring up the yellow-button menu. Instead, you can use keyboard shortcuts. These are the parenthesized expressions in the menu. Depending on your platform, you may have to press one of the modifier keys (control, alt, command, or meta). (We will indicate these generically as *CMD-key*.)




*Evaluate the expression in the workspace again, but using the keyboard*

*shortcut* : CMD-d.

In addition to **do it**, you will have noticed **print it**, **inspect it** and **explore it**. Let's have a quick look at each of these.

 *Type the expression `3 + 4` into the workspace. Now **do it** with the keyboard shortcut.*

Do not be surprised if you saw nothing happen! What you just did is send the message `+` with argument 4 to the number 3. Normally the result 7 will have been computed and returned to you, but since the workspace did not know what to do with this answer, it simply threw the answer away. If you want to see the result, you should **print it** instead. **print it** actually compiles the expression, executes it, sends the message `printString` to the result, and displays the resulting string.

 *Select `3+4` and **print it** (CMD-p).*

This time we see the result we expect (la figure 1.10).

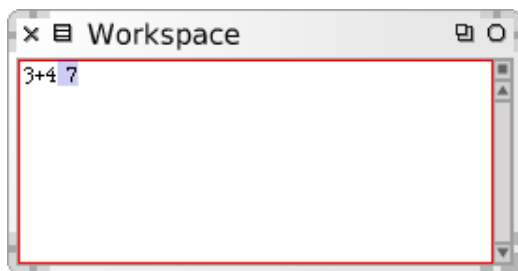



FIG. 1.10 – “Print it” rather than “do it”.

---

`3 + 4`     $\longrightarrow$     7

---

We use the notation  $\longrightarrow$  as a convention in this book to indicate that a particular Squeak expression yields a given result when you **print it**.

 *Delete the highlighted text “7” (Squeak should have selected it for you, so you can just press the delete key). Select `3+4` again and this time **inspect it** (CMD-i).*

Now you should see a new window, called an *inspector*, with the heading *SmallInteger: 7* (la figure 1.11). The inspector is an extremely useful tool that will allow you to browse and interact with any object in the system. The title tells us that 7 is an instance of the class *SmallInteger*. The left panel allows us to browse the instance variables of an object, the values of which are shown in the right panel. The bottom panel can be used to write expressions to send messages to the object.

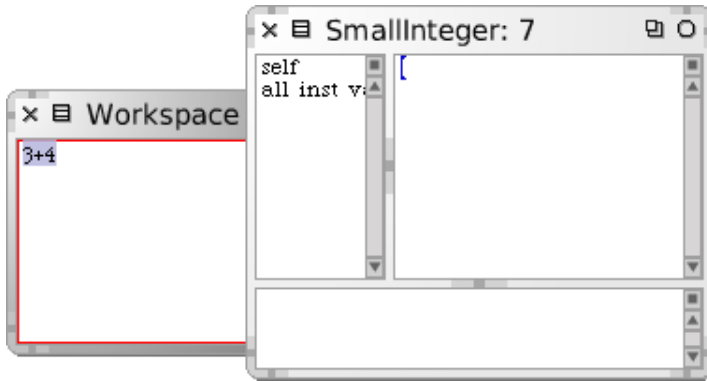




FIG. 1.11 – Inspecting an object.

 Type `self squared` in the bottom panel of the inspector on 7 and `print it`.

 Close the inspector. Type the expression `Object` in a workspace and this time `explore it` (CMD-I, uppercased i).

This time you should see a window labelled *Object* containing the text `▷ root: Object`. Click on the triangle to open it up (la figure 1.12).

The explorer is similar to the inspector, but it offers a tree view of a complex object. In this case the object we are looking at is the *Object* class. We can see directly all the information stored in this class, and we can easily navigate to all its parts.





FIG. 1.12 – Exploring an object.

## 1.6 SqueakMap

SqueakMap is a web-based catalog of “packages” — applications and libraries — that you can download to your image. The packages are hosted in many different places in the world and maintained by many different people. Some of them may only work with a specific version of Squeak.

 *Open* World ▸ open... ▸ SqueakMap package loader.

You will need an Internet connection for this to work. After some time, the SqueakMap loader window should appear (la figure 1.13). On the left side is a very long list of packages. The field in the top-left corner is a search pane that can be used to find what you want in the list. Type “Sokoban” in the search pane and hit the return key. Clicking

on the right-pointing triangle by the name of a package reveals a list of available versions. When a package or a version is selected, information about it appears in the right-hand pane. Navigate to the latest version of Sokoban. With the mouse in the list pane, use the yellow-button menu to **install** the selected package. (If Squeak complains that it is not sure this version of the game will work in your image, just say “yes” and go ahead.) Notice that once a package has been installed, it is marked with an asterisk in the list in the SqueakMap package loader.

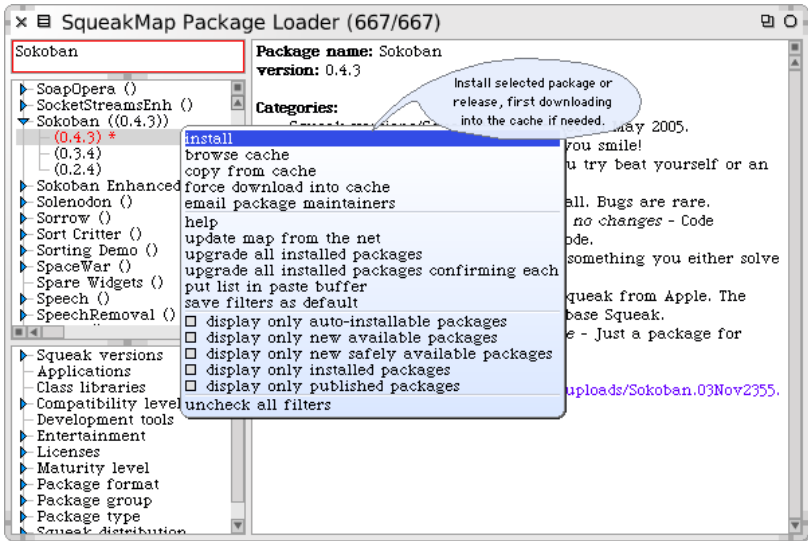



FIG. 1.13 – Using SqueakMap to install the Sokoban game.

 After installing this package, start up Sokoban by evaluating `SokobanMorph random openInWorld` in a workspace.

The bottom-left pane in the SqueakMap package loader provides various ways to filter the list of packages. You can choose to see only those packages that are compatible with a particular version of Squeak, or only games, and so on.

## 1.7 The System Browser

The system browser is one of the key tools used for programming. As we shall see, there are several interesting browsers available for Squeak, but this is the basic one you will find in any image.

 Open a browser by selecting `World > open ... > class browser`, or by dragging a Browser from the l'onglet Tools.

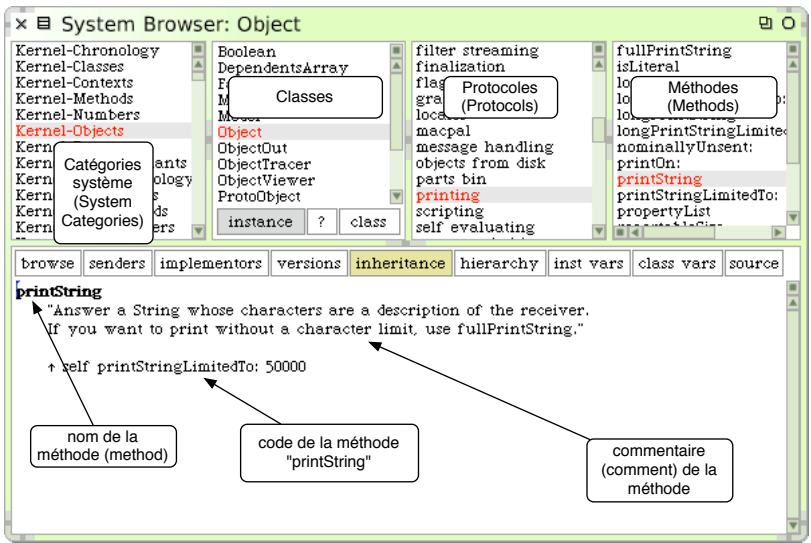


FIG. 1.14 – The system browser showing the `printString` method of class `Object`.

We can see a system browser in the figure 1.14. The title bar indicates that we are browsing the class `Object`.


When the browser first opens, all panes are empty but the leftmost one. This first pane lists all known *system categories*, which are groups of related classes.

 Click on the category `Kernel-Objects`.


This causes the second pane to show a list of all of the classes in the selected category.

 *Select the class Object.*

Now the remaining two panes will be filled with text. The third pane displays the *protocols* of the currently selected class. These are convenient groupings of related methods. If no protocol is selected you should see all methods in the fourth pane.

 *Select the printing protocol.*

You may have to scroll down to find it. Now you will see in the fourth pane only methods related to printing.


 *Select the printString method.*

Now we see in the bottom pane the source code of the printString method, shared by all objects in the system (except those that override it).


## 1.8 Finding classes

There are several ways to find a class in Squeak. The first, as we have just seen above, is to know (or guess) what category it is in, and to navigate to it using the browser.

A second way is to send the browse message to the class, asking it to open a browser on itself. Suppose we want to browse the class Boolean.

 *Type Boolean browse into a workspace and **do it**.*

A browser will open on the Boolean class (la figure 1.15). There is also a keyboard shortcut CMD-b (browse) that you can use in any tool where you find a class name ; select the name and type CMD-b.

 *Use the keyboard shortcut to browse the class Boolean.*

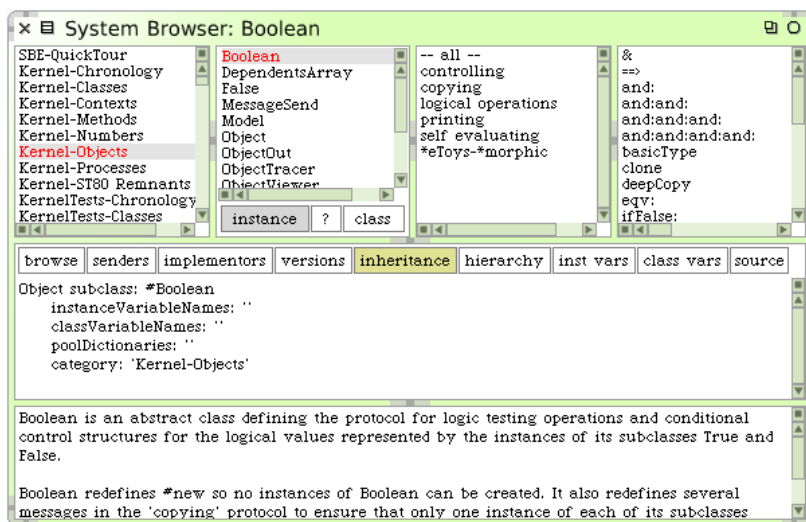



FIG. 1.15 – The system browser showing the definition of class Boolean.

Notice that when the Boolean class is selected but no protocol or method is selected, two panes rather than one appear below the four panes at the top (la figure 1.15). The upper one contains the *class definition*. This is nothing more than an ordinary Smalltalk message that is sent to the parent class, asking it to create a subclass. Here we see that the class Object is being asked to create a subclass named Boolean with no instance variables, class variables or “pool dictionaries”, and to put the class Boolean in the *Kernel-Objects* category.


The lower pane shows the *class comment* — a piece of plain text describing the class. If you click on the ? at the bottom of the class pane, you can see the class comment in a dedicated pane.

If you would like to explore Squeak’s inheritance hierarchy, the *hierarchy browser* can help you. This can be useful if you are looking for an unknown subclass or superclass of a known class. The hierarchy browser is like the system browser, except that the list of classes is arranged as an indented tree mirroring the inheritance hierarchy.

 Click on **hierarchy** in the browser while the class **Boolean** is selected.

This will open a hierarchy browser showing the superclasses and sub-classes of **Boolean**. Navigate to the immediate superclass and subclasses of **Boolean**.

Often, the fastest way to find a class is to search for it by name. For example, suppose that you are looking for some unknown class that represents dates and times.

 Put the mouse in the system category pane of the system browser and type **CMD-f**, or select **find class ... (f)** from the yellow-button menu. Type “time” in the dialog box and accept it.

You will be presented with a list of classes whose names contain “time” (see la figure 1.16). Choose one, say, **Time**, and the browser will show it, along with a class comment that suggests other classes that might be useful. If you want to browse one of the others, select its name (in any text pane), and type **CMD-b**.

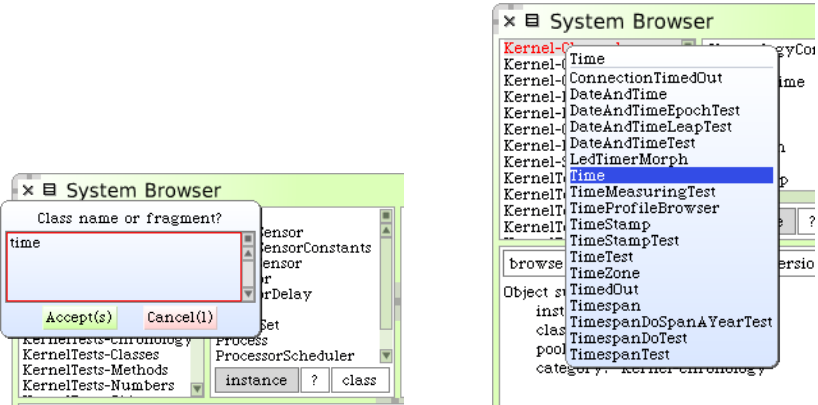



FIG. 1.16 – Searching for a class by name.

Note that if you type the complete (and correctly capitalized) name of a class in the find dialog, the browser will go directly to that class without showing you the list of options.

## 1.9 Finding methods

Sometimes you can guess the name of a method, or at least part of the name of a method, more easily than the name of a class. For example, if you are interested in the current time, you might expect that there would be a method called “now”, or containing “now” as a substring. But where might it be? The *method finder* can help you.

 Drag the **method finder** icon out of the l’onglet Tools. Type “now” in the top left pane, and **accept** it (or just press the RETURN key).

The method finder will display a list of all the method names that contain the substring “now”. To scroll to now itself, type “n”; this trick works in all scrolling windows. Select “now” and the right-hand pane shows you the three classes that define a method with this name, as shown in la figure 1.17. Selecting any one of them will open a browser on it.

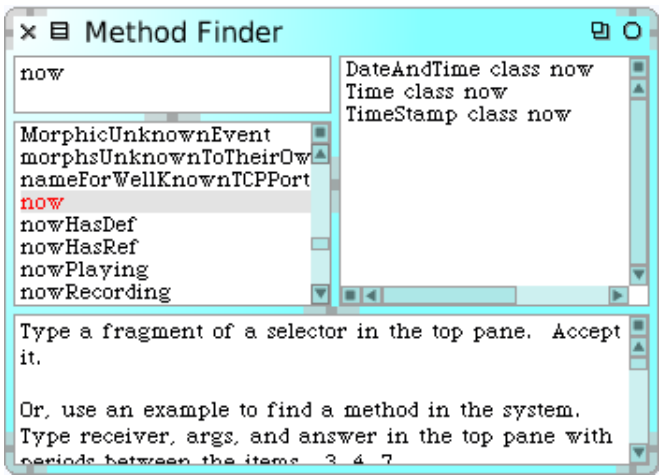



FIG. 1.17 – The method finder showing three classes that define a method named now.

At other times you may have a good idea that a method exists, but will have no idea what it might be called. The method finder can still

help ! For example, suppose that you would like to find a method that turns a string into upper case, for example, it would translate 'eureka' into 'EUREKA'.

 Type 'eureka'. 'EUREKA' into the method finder, as shown in la figure 1.18.

The method finder will suggest a method that does what you want.

A star at the beginning of a line in the right pane of the method finder indicates that this method is the one that was actually used to obtain the requested result. So, the star in front of String asUppercase lets us know that the method asUppercase defined on the class String was executed and returned the result we wanted. The methods that do not have a star are just the other methods that have the same name as the ones that returned the expected result. So Character»asUppercase was not executed on our example, because 'eureka' is not a Character object.

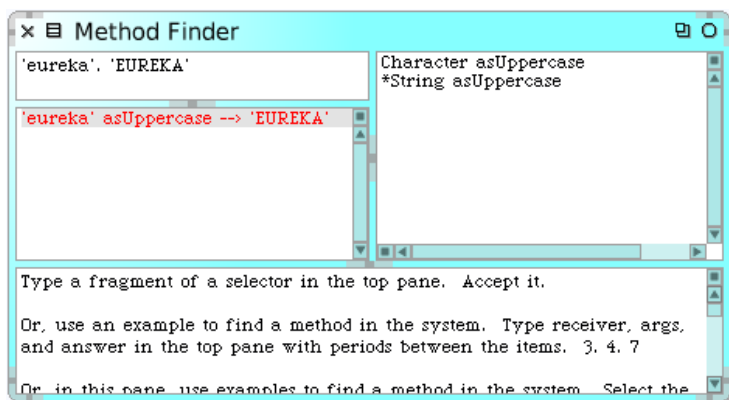


FIG. 1.18 – Finding a method by example.

You can also use the method finder for methods with arguments ; for example, if you are looking for a method that will find the greatest common factor of two integers, you might try 25. 35. 5 as an example. You can also give the method finder multiple examples to narrow the search space ; the help text in the bottom pane explains more.



## 1.10 Defining a new method

The advent of Test Driven Development<sup>1</sup> has changed the way that we write code. The idea behind Test Driven Development, also called TDD or Behavior Driven Development, is that we write a test that defines the desired behavior of our code *before* we write the code itself. Only then do we write the code that satisfies the test.

Suppose that our assignment is to write a method that “says something loudly and with emphasis”. What exactly could that mean? What would be a good name for such a method? How can we make sure that programmers who may have to maintain our method in the future have an unambiguous description of what it should do? We can answer all of these questions by giving an example :

When we send the message shout to the string “Don’t panic”  
the result should be “DON’T PANIC!”.

To make this example into something that the system can use, we turn it into a test method :

---


### Méthode 1.1 – A test for a shout method

---

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

---

How do we create a new method in Squeak? First, we have to decide which class the method should belong to. In this case, the shout method that we are testing will go in class String, so the corresponding test will, by convention, go in a class called StringTest.

 Open a browser on the class StringTest, and select an appropriate protocol for our method, in this case **tests - converting**, as shown in la figure 1.19. The highlighted text in the bottom pane is a template that reminds you what a Smalltalk method looks like. Delete this and enter the code from méthode 1.1.

Once you have typed the text into the browser, notice that the bottom pane is outlined in red. This is a reminder that the pane contains

---

<sup>1</sup>2,.

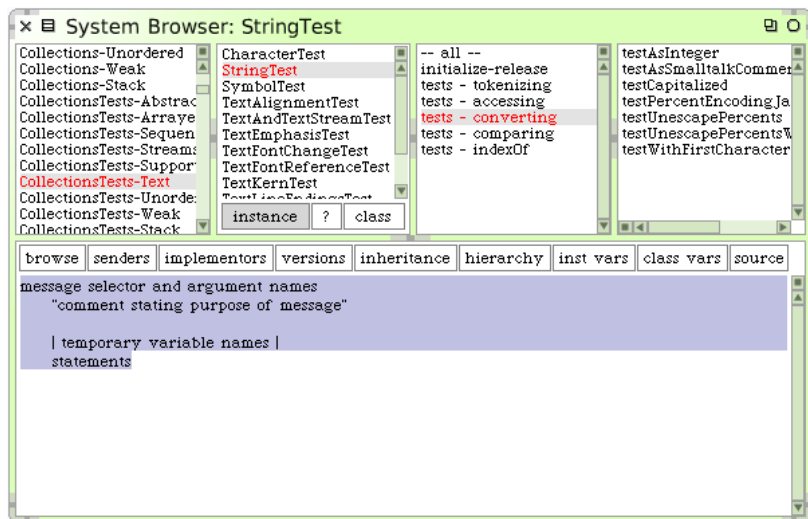



FIG. 1.19 – The new method template in class StringTest.

unsaved changes. So select **accept (s)** from the yellow-button menu in the bottom pane, or just type **CMD-s**, to compile and save your method.

Because there is as yet no method called **shout**, the browser will ask you to confirm that this is the name that you really want—and it will suggest some other names that you might have intended (la figure 1.20). This can be quite useful if you have merely made a typing mistake, but in this case, we really *do* mean **shout**, since that is the method we are about to create, so we have to confirm this by selecting the first option from the menu of choices, as shown in la figure 1.20.

 *Run your newly created test : open the SUnit TestRunner, either by dragging it from the l'onglet Tools, or by selecting **World > open... > Test Runner**.*

The leftmost two panes are a bit like the top panes in the system browser. The left pane contains a list of system categories, but it's restricted to those categories that contain test classes.

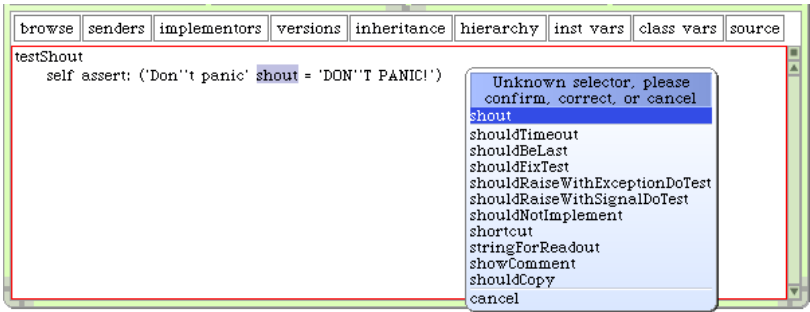



FIG. 1.20 – Accepting the testShout method class StringTest.

 Select CollectionsTests-Text and the pane to the right will show all of the test classes in that category, which includes the class StringTest. The names of the classes are already selected, so click Run Selected to run all these tests.

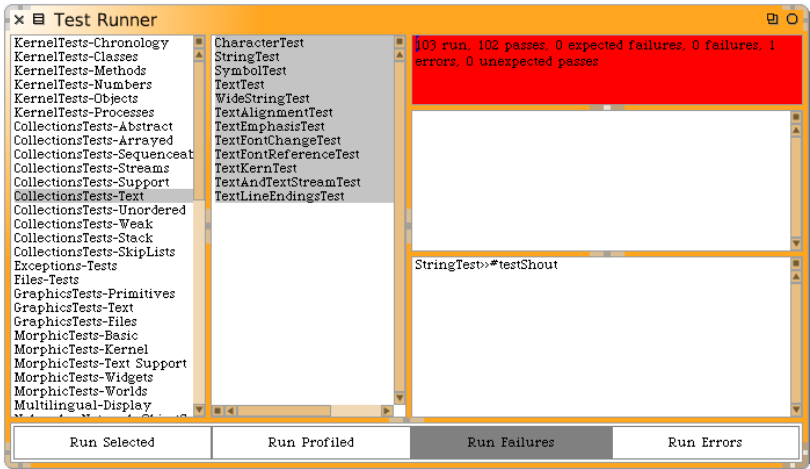


FIG. 1.21 – Running the String tests.

You should see a message like that shown in la figure 1.21, which indicates that there was an error in running the tests. The list of tests that gave rise to errors is shown in the bottom right pane ; as

you can see, `StringTest>>testShout` is the culprit. (Note that `StringTest>>#testShout` is the Smalltalk way of identifying the `testShout` method of the `StringTest` class.) If you click on that line of text, the erroneous test will run again, this time in such a way that you see the error happen : “`MessageNotUnderstood: ByteString>>shout`”.

The window that opens with the error message is the Smalltalk debugger (see la figure 1.22). We will look at the debugger and how to use it in le chapitre 6.

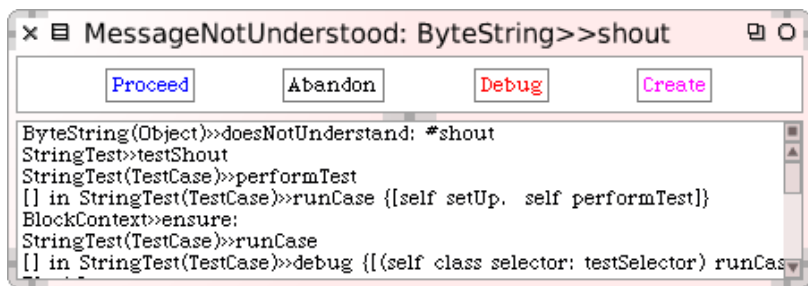



FIG. 1.22 – The (pre-)debugger.

The error is, of course, exactly what we expected : running the test generates an error because we haven’t yet written a method that tells strings how to shout. Nevertheless, it’s good practice to make sure that the test fails because this confirms that we have set up the testing machinery correctly and that the new test is actually being run. Once you have seen the error, you can **Abandon** the running test, which will close the debugger window. Note that often with Smalltalk you can define the missing method using the **Create** button, edit the newly-created method in the debugger, and then **Proceed** with the test.

Now let’s define the method that will make the test succeed !


 Select class `String` in the system browser, select the **converting** protocol, type the text in *méthode 1.2* over the method creation template, and **accept** it. (Note : to get a  $\uparrow$ , type  $\wedge$ ).

Méthode 1.2 – The shout method

```
shout
  ↑ self asUppercase, '!'
```

The comma is the string concatenation operation, so the body of this method appends an exclamation mark to an upper-case version of whatever String object the shout message was sent to. The ↑ tells Squeak that the expression that follows is the answer to be returned from the method, in this case the new concatenated string.

Does this method work ? Let’s run the tests and see.

 Click on **Run Selected** again in the test runner, and this time you should see a green bar and text indicating that all of the tests ran with no failures and no errors.

When you get to a green bar<sup>2</sup>, it’s a good idea to save your work and take a break. So do that right now !

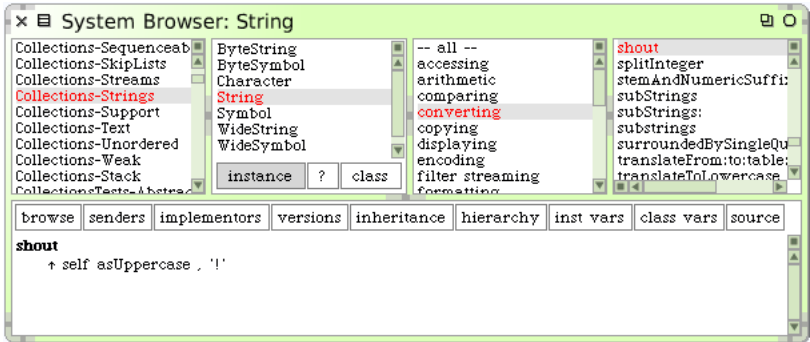


FIG. 1.23 – The shout method defined on class String.

<sup>2</sup>Actually, you might not get a green bar since some images contains tests for bugs that need to be fixed. Don’t worry about this. Squeak is constantly evolving.

## 1.11 Chapter summary

This chapter has introduced you to the Squeak environment and shown you how to use some of the major tools, such as the system browser, the method finder, and the test runner. You have also seen a little of Squeak's syntax, even though you may not understand it all yet.

- A running Squeak system consists of a *virtual machine*, a *sources* file, and *image* and *changes* files. Only these last two change, as they record a snapshot of the running system.
- When you restore a Squeak image, you will find yourself in exactly the same state — with the same running objects — that you had when you last saved that image.
- Squeak is designed to work with a three-button mouse. The buttons are known as the *red*, the *yellow* and the *blue* buttons. If you don't have a three-button mouse, you can use modifier keys to obtain the same effect.
- You use the red button on the Squeak background to bring up the *World menu* and launch various tools. You can also launch tools from the l'onglet *Tools* at the right of the Squeak screen.
- A *workspace* is a tool for writing and evaluating snippets of code. You can also use it to store arbitrary text.
- You can use keyboard shortcuts on text in the workspace, or any other tool, to evaluate code. The most important of these are `do it` (CMD-d), `print it` (CMD-p), `inspect it` (CMD-i) and `explore it` (CMD-I).
- SqueakMap is a tool for loading useful packages from the Internet.
- The *system browser* is the main tool for browsing Squeak code, and for developing new code.
- The *test runner* is a tool for running unit tests. It also supports Test Driven Development.

## Chapitre 2

# Une première application

Dans ce chapitre, nous allons développer un jeu très simple, le jeu de Quinto. En cours de route, nous allons faire la démonstration de la plupart des outils que les développeurs Squeak utilisent pour construire et déboguer leurs programmes et comment les programmes sont échangés entre les développeurs. Nous verrons notamment le navigateur de classes (system browser), 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 dû au fait que Smalltalk est 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 de Quinto

Pour vous montrer comment utiliser les outils de développement de Squeak, nous allons construire un jeu très simple nommé *Quinto*. Le tableau 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 bleue. Cliquez de nouveau et elles repassent au jaune pâle. Le but du jeu est

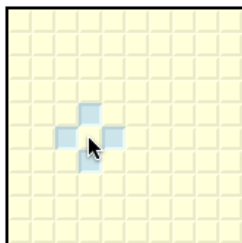



FIG. 2.1 – Le plateau de jeu de Quinto. L'utilisateur vient de cliquer sur une case avec la souris comme le montre le curseur.

de passer au bleu autant de cellules que possible.

Le jeu de Quinto 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 individuels. Le code Squeak 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 Squeak.

## 2.2 Créer une nouvelle catégorie de classe

Nous avons déjà vu le navigateur de classes (system browser) dans le chapitre 1, où nous avons appris à naviguer dans les classes et méthodes et comment définir de nouvelles méthodes. Nous allons maintenant voir comment créer des catégories systèmes et des classes.

 Ouvrir un navigateur de classes et cliquer avec le bouton jaune sur le panneau des catégories. Sélectionner `add item ...`.

Taper le nom de la nouvelle catégorie (nous allons utiliser *SBE-Quinto*) dans la boîte de dialogue et cliquer sur `accept` (ou appuyer juste la touche entrée) ; la nouvelle catégorie est créée et se positionne à la fin de la liste de catégories. Si vous sélectionné une catégorie existante, alors la nouvelle catégorie sera positionnée juste après celle sélectionnée.



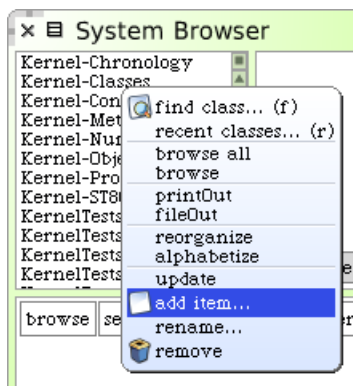


FIG. 2.2 – Ajouter une catégorie système.

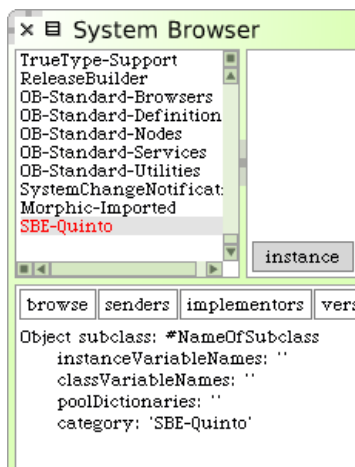



FIG. 2.3 – Le modèle de création d'une classe.

## 2.3 Définir la classe SBECcell

Pour l'instant, il n'y a aucune classe dans cette nouvelle catégorie. Néanmoins, la zone principale d'édition affiche un modèle afin de créer facilement 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 *SBE-Quinto*.

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

 *Modifier le modèle de création d'une classe comme suit :*

- Remplacer Object par SimpleSwitchMorph.
- Remplacer NameOfSubClass par SBECcell.
- Ajouter mouseAction dans la liste de variables d'instances.

Le résultat doit ressembler à la classe 2.1.

### Classe 2.1 – Définition de la classe SBECeIl

---

```
SimpleSwitchMorph subclass: #SBECeIl
  instanceVariableNames: 'mouseAction'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'SBE-Quinto'
```

---

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 SBECeIl. (En fait, comme SBECeIl n'existe pas encore, nous passons comme argument le *symbole* #SBECeIl 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 (la figure 2.4). Cela signifie qu'il y a des *modifications non sauvegardées*. Pour effectivement envoyer ce message, vous devez faire `accept`.

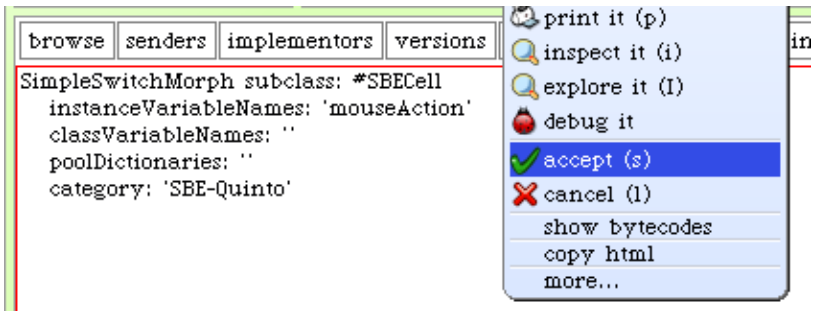



FIG. 2.4 – Le modèle de création d'une classe.

 *Accepter la nouvelle définition de classe.*

Utiliser le clic jaune ou bien sélectionner `accept` ou encore utiliser le raccourci clavier CMD-s (pour "save"). Ce message sera envoyé à


SimpleSwitchMorph, ce qui compilera la nouvelle classe.

Une fois la définition de classe acceptée, la classe va être créée et apparaître dans le panneau des classes du navigateur (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. On appelle cela un *commentaire de classe* et cela assez important d'en écrire un qui donnera aux autres développeurs une vision de haut niveau de votre classe. Les Smalltalkiens accordent une grande valeur à la lisibilité de leur code et il est inusuel de trouver des commentaires détaillés dans leurs méthodes ; la philosophie est plutôt d'avoir un code qui parle pour lui-même (si cela n'est pas le cas, vous devez le refactoriser, jusqu'à ce que le soit !). Un commentaire de classe ne nécessite pas une description détaillée de la classe, mais quelques mots la décrivant sont vital si les développeurs qui viennent après vous souhaitent passer un peu de temps sur votre classe.

 *Taper un commentaire de classe pour SBCell et accepter le ; vous aurez tout le loisir de l'améliorer par la suite.*

## 2.4 Ajouter des méthodes à la classe

Ajoutons maintenant quelques méthodes à notre classe.

 *Sélectionnez le protocole --all-- dans le panneau des contrôleurs.*

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 méthode 2.2.



FIG. 2.5 – La classe nouvellement créée SBECCell

## Méthode 2.2 – Initialisation des instances de SBECCell

---


```

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 quotes séparées avec rien entre les deux, et pas un guillemet ! " représente la chaîne de caractères vide.

 Faire un `accept` de cette définition de méthode.

Que fait le code ci-dessus ? Nous n'allons pas rentrer dans tous les détails maintenant (c'est l'objet du reste de ce livre !), mais nous allons

vous en donner un bref aperçu. Prenons le ligne par ligne.

Notons que la méthode s'appelle *initialize*. Ce nom dit bien ce qu'il veut dire ! Par convention, si une classe définit une méthode nommée *initialize*, elle sera appelée dès que l'objet aura été créé. Ainsi dès que nous évaluons `SBECCell 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 chose que cette méthode fait (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 faut mieux le faire sinon il y a le risque d'avoir un état incorrect.

Le reste de la méthode donne un état à cet objet. Envoyer `self label:` " par exemple fixe 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 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 `Squeak` est en *haut à gauche* et que les coordonnées en *y* augmente *vers le bas*.

Le reste de la méthode doit être compréhensible d'elle 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 du `python` anglais. Vous devriez être capable d'imaginer l'objet

se parlant à lui-même et dire : “Utilise des bords carrés!”, “\Eteinds les cellules!”.

## 2.5 Inspecter un objet

Vous pouvez tester l’effet du code que vous avez écrit en créant un nouvel objet SBCell et en l’inspectant.

 Ouvrir un espace de travail. Tapez l’expression `SBCell new` et choisissez `inspect it`.

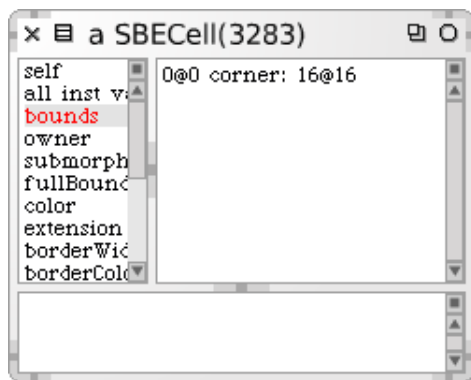




FIG. 2.6 – L’inspecteur utilisé pour examiner l’objet SBCell.

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 affiché dans le panneau droit. Vous pouvez également utiliser l’inspecteur pour changer la valeur d’une variable d’instance.

 Changer la valeur de `bounds` à `0@0 corner: 50@50` et faire `accept`.

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

 Tapez le texte `self openInWorld` dans la zone du bas et choisir **do it**.

La cellule doit apparaître à l'angle haut à gauche de l'écran, en fait à l'endroit exact où sa variable `bounds` dit qu'elle doit apparaître. Faire un clic bleu sur la cellule afin de faire apparaître son halo morphique. Déplacer la cellule avec la poignée marron (à côté de celle en haut à droite) et redimensionner la avec la poignée jaune (en bas à droite). Vérifier que les limites indiquées par l'inspecteur sont modifiées en conséquence.

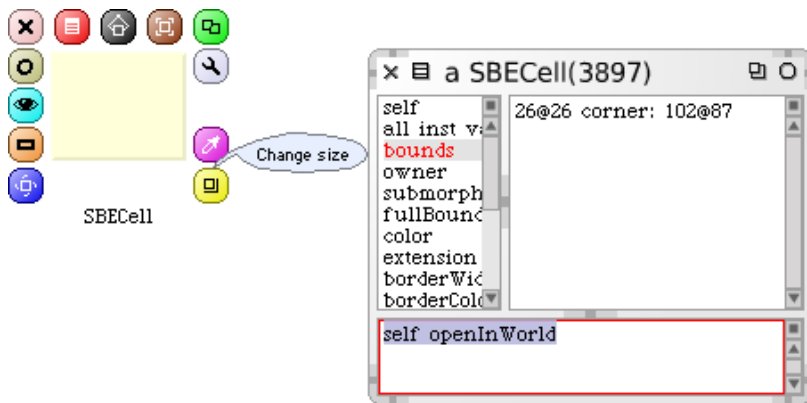




FIG. 2.7 – Redimensionner la cellule.

 Détruire la cellule en cliquant sur le x de la poignée mauve.

## 2.6 Définir la classe SBEGame

Créons maintenant l'autre classe dont nous avons besoin dans le jeu, que nous appellerons SBEGame.

 Faire apparaître le modèle de définition de classe dans la fenêtre principale du navigateur.

Pour cela re-sélectionner le nom de la catégorie de classe existante ou en affichant de nouveau la définition de SBECCell (en cliquant sur le bouton `instance`). Éditer le code telle sorte qu'il peut être lu comme suit et faire `accept`.

### Classe 2.3 – Définition de la classe SBEGame


---

```
BorderedMorph subclass: #SBEGame
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  category: 'SBE-Quinto'
```

---

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

Définissons maintenant une méthode initialize pour SBEGame.

 Tapez ce qui suit dans le navigateur comme une méthode de SBEGame et faire ensuite `accept` :

### Méthode 2.4 – Inialisation du jeu

---

```
1 initialize
2   | sampleCell width height n |
3   super initialize.
4   n := self cellsPerSide.
5   sampleCell := SBECCell 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 ].
```

---

Squeak va se plaindre qu'il ne connaît pas la signification de certains termes. Squeak vous indique qu'il ne connaît pas le message `cellsPerSide`, et suggère un certain nombre de propositions, dans le cas où il s'agirait d'une erreur de frappe.



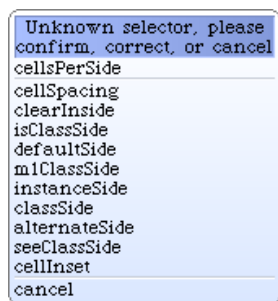


FIG. 2.8 – Squeak détecte un sélecteur inconnu.

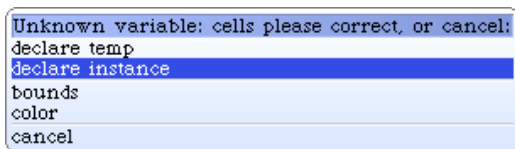




FIG. 2.9 – Déclaration d’une nouvelle variable d’instance.

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 faire dans une minute ou deux.

 Sélectionner le premier élément du menu, afin de confirmer que nous parlons bien de `cellsPerSide`.

Puis, Squeak va se plaindre de ne pas connaître la signification `cells`. Il vous offre plusieurs possibilités de le corriger.

 Choisir `declare instance` parce que nous souhaitons que `cells` soit une variable d’instance.

Enfin, Squeak va se plaindre à propos du message `newCellAt:at:` envoyé à la dernière ligne ; ce n’est toujours pas une erreur, confirmez donc ce message également.

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

Quelle taille doit faire notre plateau de jeu ? Big enough to hold some integral number of cells, and big enough to draw a border around them. Quel est le bon nombre de cellules ? 5 ? 10 ? 100 ? Nous ne le savons pas pour l’instant et si nous savions, il y aurait des chances pour que nous changions par la suite d’idée par la suite. Nous délégons donc la responsabilité de connaître ce nombre à une autre méthode, que nous appelons `cellsPerSide`, et que nous écrirons dans une minute ou deux. C’est parce que nous envoyons le message `cellsPerSide` avant de définir une méthode avec ce nom que Squeak nous demande “confirm, correct, or cancel” lorsque nous acceptons le corps de la méthode `initialize`. Ne soyez pas inquieté par cela : 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 qui fait sens et nous avons poursuivi, sans nous interrompre.

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

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

La ligne 8 fixe la valeur de `bounds` du nouvel objet. Without worrying too much about the details just yet, just believe us that the expression in parentheses creates a square with its origin (c-à-d. its top-left corner) at the point (5,5) and its bottom-right corner far enough away to allow space for the right number of cells.

The last line sets the `SBEGame` object’s instance variable `cells` to a newly created `Matrix` with the right number of rows and columns. We do this by sending the message `new:tabulate:` to the `Matrix` class (classes are objects too, so we can send them messages). We know that `new:tabulate:` takes two arguments because it has two colons (:) in its name. The arguments go right after the colons. If you are used to

languages that put all of the arguments together inside parentheses, this may seem weird at first. Don't panic, it's only syntax ! It turns out to be a very good syntax because the name of the method can be used to explain the roles of the arguments. For example, it is pretty clear that Matrix rows: 5 columns: 2 has 5 rows and 2 columns, and not 2 rows and 5 columns.

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  $(i,j)^{\text{th}}$  sera initialisé avec le résultat de l'évaluation de `self newCellAt: i at: j`.

Voilà pour initialize. When you accept this message body, you might want to take the opportunity to pretty-up the formatting. You don't have to do this by hand : from the the yellow-button menu select `more ... ▸ prettyprint`, and the browser will do it for you. You have to accept again after you have pretty-printed a method, or of course you can cancel (CMD-L — that's a lower-case letter L) if you don't like the result. Alternatively, you can set up the browser to use the pretty-printer automatically whenever it shows you code : use the the right-most button in the button bar to adjust the view.

If you find yourself using `more ...` a lot, it's useful to know that you can hold down the SHIFT key when you click to directly bring up the `more ...` menu.

## 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 de telle sorte que nous ne soyons pas submergé par une liste de nom de classes trop longue dans le second panneau, le troisième panneau nous permet de catégoriser les méthodes de telle sorte que n'ayons pas pas une liste de méthodes trop longue dans le quatrième panneau. Ces catégories de méthodes sont appelés "protocoles".

S'il y avait juste quelques méthodes par classes, ce niveau hiérarchique supplémentaire ne serait pas vraiment nécessaire. C'est pour

cela que le navigateur offre un protocole virtuel `--all--`, qui vous ne serez pas surpris d'apprendre, contient toutes les méthodes de la classe.

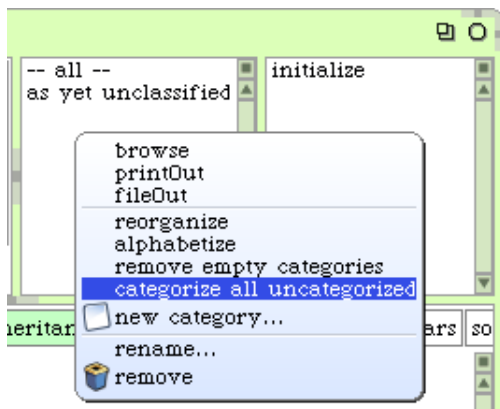



FIG. 2.10 – Catégoriser toutes les méthodes non catégorisées.

Si vous avez suivi l'exemple jusqu'à présent, le troisième panneau doit contenir le protocole *as yet unclassified*<sup>1</sup>.

 Sélectionner avec le bouton jaune l'élément du menu `categorize all uncategorized` afin de régler ce problème et déplacer les méthodes `initialize` vers un nouveau protocole appelé `initialization`.

Comment Squeak sait que c'est le bon protocole ? Well, in general Squeak can't know, but in this case there is also an `initialize` method in a superclass, and Squeak assumes that our `initialize` method should go in the same category as the one that it overrides.

You may find that Squeak has already put your `initialize` method into the `initialization` protocol. If so, it's probably because you have loaded a package called `AutomaticMethodCategorizer` into your image.

**Une convention typographique.** Les Smalltalkiens utilisent fréquemment la notation `">>"` afin d'identifier la classe à laquelle la méthode

<sup>1</sup>NdT : non encore classifié

appartient, ainsi par exemple la méthode `cellsPerSide` de la classe `SBEGame` sera référencé comme `SBEGame>>cellsPerSide`. Afin d'indiquer que cela ne fait pas parti de la syntaxe de Smalltalk, nous utiliserons plutôt le symbole spécial » de telle sorte que cette méthode apparaîtra dans le texte comme `SBEGame»cellsPerSide`

From now on, when we show a method in this book, we will write the name of the method in this form. Of course, when you actually type the code into the browser, you don't have to type the class name or the » ; instead, you just make sure that the appropriate class is selected in the class pane.

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

---

#### Méthode 2.5 – Une méthode constante.

---

`SBEGame»cellsPerSide`

*"The number of cells along each side of the game"*

↑ 10

---

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

---

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

---

`SBEGame»newCellAt: i at: j`

*"Create a cell for position (i,j) and add it to my on-screen representation at the appropriate screen position. Answer the new cell"*

| c origin |

c := SBECell 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].

---



Ajouter les méthodes `SBEGame»cellsPerSide` et `SBEGame»newCellAt:at:`

.

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

Méthode 2.6 answers a new `SBECCell`, specialized to position  $(i, j)$  in the Matrix of cells. The last line defines the new cell's `mouseAction` to be the *block* `[self toggleNeighboursOfCellAt: i at: j]`. In effect, this defines the callback behaviour to perform when the mouse is clicked. The corresponding method also needs to be defined.


#### Méthode 2.7 – The callback method

---

```
SBEGame>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].
```

---

Méthode 2.7 toggles the state of the four cells to the north, south, west and east of cell  $(i, j)$ . The only complication is that the board is finite, so we have to make sure that a neighboring cell exists before we toggle its state.

 Place this method in a new protocol called game logic.

To move the method, you can simply click on its name and drag it to the newly-created protocol (la figure 2.11).

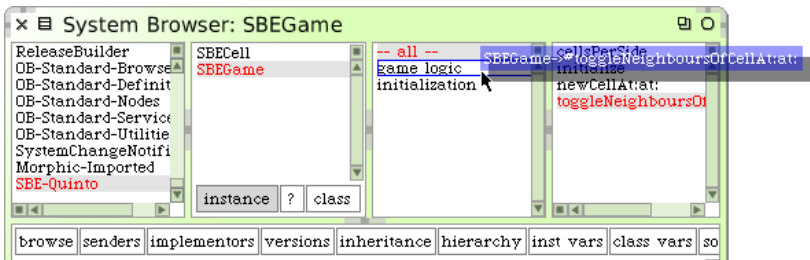


FIG. 2.11 – Faire un glisser-déposer de la méthode dans un protocole.

To complete the Quinto game, we need to define two more methods in class `SBECCell` to handle mouse events.

Méthode 2.8 – *A typical setter method*


---

SBECCell»mouseAction: aBlock


↑ mouseAction := aBlock

---

Méthode 2.8 does nothing more than set the cell's mouseAction variable to the argument, and then answers the new value. Any method that *changes* the value of an instance variable in this way is called a *setter method*; a method that *answers* the current value of an instance variable is called a *getter method*.

If you are used to getters and setters in other programming languages, you might expect these methods to be called setmouseAction and getmouseAction. The Smalltalk convention is different. A getter always has the same name as the variable it gets, and a setter is named similarly, but with a trailing “:”, hence mouseAction and mouseAction:.

Collectively, setters and getters are called *accessor methods*, and by convention they should be placed in the *accessing* protocol. In Smalltalk, *all* instance variables are private to the object that owns them, so the only way for another object to read or write those variables in the Smalltalk language is through accessor methods like this one<sup>2</sup>.

 Go to the class SBECCell, define SBECCell»mouseAction: and put it in the accessing protocol.

Finally, we need to define a method mouseUp:; this will be called automatically by the GUI framework if the mouse button is released while the mouse is over this cell on the screen.


Méthode 2.9 – *Un gestionnaire d'évènement.*


---

SBECCell»mouseUp: anEvent

mouseAction value

---

 Ajouter la méthode SBECCell»mouseUp: ensuite faire categorize all uncategorized.

What this method does is to send the message value to the object stored in the instance variable mouseAction. Recall that in SBEGame»

---

<sup>2</sup>In fact, the instance variables can be accessed in subclasses too.

newCellAt: i at: j we assigned the following code fragment to `mouseAction` :


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

Sending the value message causes this code fragment to be evaluated, and consequently the state of the cells will toggle.

## 2.8 Essayons notre code

Voilà, le jeu de Quinto est complet !

Si vous avez suivi toutes les étapes, vous devez pouvoir jouer au jeu qui consistent en 2 classes et 7 méthodes.

 Dans un espace de travail, tapez `SBEGame new openInWorld` et faire `do it`.

The game will open, and you should be able to click on the cells and see how it works.

Well, so much for theory... When you click on a cell, a *notifier* window called the `PreDebugWindow` window appears with an error message ! As depicted in la figure 2.12, it says `MessageNotUnderstood: SBEGame>>toggleState`.

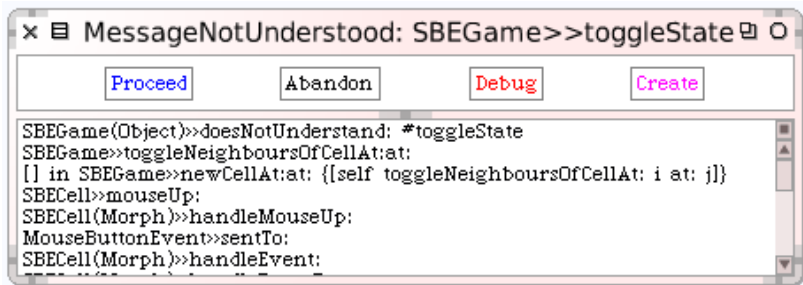



FIG. 2.12 – Il y a une erreur dans notre jeu lorsqu’une cellule est sélectionnée !


What happened ? To find out, let’s use one of Smalltalk’s more power-



ful tools : the debugger.

 Click on the `debug` button in the notifier window.

The debugger will appear. In the upper part of the debugger window you can see the execution stack, showing all the active methods ; selecting any one of them will show, in the middle pane, the Smalltalk code being executed in that method, with the part that triggered the error highlighted.

 Click on the line labelled `SBEGame>>toggleNeighboursOfCellAt:at:` (near the top).

The debugger will show you the execution context within this method where the error occurred (la figure 2.13).

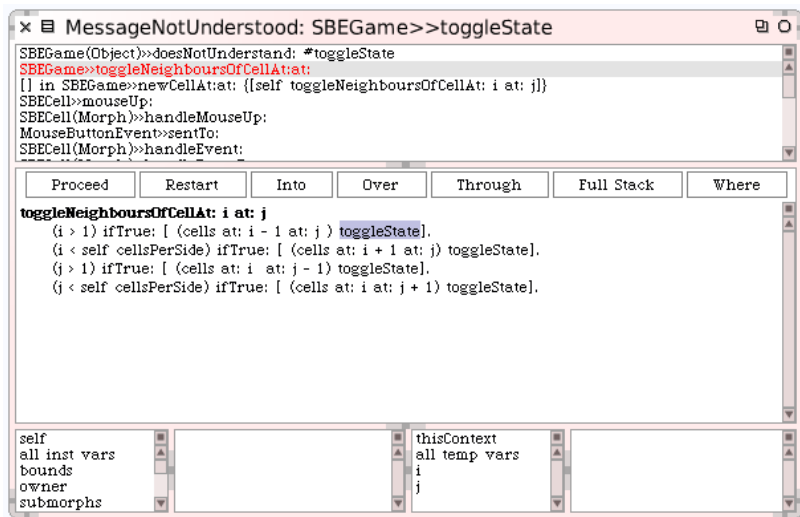



FIG. 2.13 – Le débogueur avec la méthode `toggleNeighboursOfCell:at:` sélectionnée.

At the bottom of the debugger are two small inspector windows. On the left, you can inspect the object that is the receiver of the message that caused the selected method to execute, so you can look here to see

the values of the instance variables. On the right you can inspect an object that represents the currently executing method itself, so you can look here to see the values of the method's parameters and temporary variables.

Using the debugger, you can execute code step by step, inspect objects in parameters and local variables, evaluate code just as you can in a workspace, and, most surprisingly to those used to other debuggers, change the code while it is being debugged ! Some Small-talkers program in the debugger almost all the time, rather than in the browser. The advantage of this is that you see the method that you are writing as it will be executed, with real parameters in the actual execution context.

In this case we can see in the first line of the top panel that the `toggleState` message has been sent to an instance of `SBEGame`, while it should clearly have been an instance of `SBECCell`. The problem is most likely with the initialization of the `cells` matrix. Browsing the code of `SBEGame»initialize` shows that `cells` is filled with the return values of `newCellAt:at:`, but when we look at that method, we see that there is no return statement there ! By default, a method returns `self`, which in the case of `newCellAt:at:` is indeed an instance of `SBEGame`.

 *Fermer la fenêtre du débogueur. Ajouter l'expression "`↑ c`" à la fin de la méthode `SBEGame»newCellAt:at:` de telle sorte quelle retourne `c`. (Voir méthode 2.10.)*

#### Méthode 2.10 – Corriger l'erreur.


---

```
SBEGame»newCellAt: i at: j
  "Create a cell for position (i,j) and add it to my on-screen
  representation at the appropriate screen position. Answer the new cell"
  | c origin |
  c := SBECCell 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
```

---

Recall from le chapitre 1 that the construct to return a value from a method in Smalltalk is `↑`, which you obtain by typing `^`.

Often, you can fix the code directly in the debugger window and click **Proceed** to continue running the application. In our case, because the bug was in the initialization of an object, rather than in the method that failed, the easiest thing to do is to close the debugger window, destroy the running instance of the game (with the halo), and create a new one.

 *Exécuter : SBEGame new openInWorld de nouveau.*

Le jeu doit maintenant se dérouler sans problèmes.


## 2.9 Sauvegarder et partager le code Smalltalk

Maintenant que nous avons un jeu de Quinto qui fonctionne, vous avez probablement envie de le sauvegarder quelque part de telle sorte à pouvoir le partager avec des amis. Bien sur, vous pouvez sauvegarder l'ensemble de votre image Squeak et montrer votre premier programme en l'exécutant, mais vos amis ont probablement leur propre code dans leurs images et ne veulent pas sans passer pour utiliser votre image. Ce dont nous avons besoin est de pouvoir extraire le code source d'une image Squeak afin que d'autres développeurs puissent le charger dans leurs images.

The simplest way of doing this is by *filing out* the code. The yellow-button menu in the System Categories pane will give you the option to file out the whole of category *SBE-Quinto*. The resulting file is more or less human readable, but is really intended for computers, not humans. You can email this file to your friends, and they can file it into their own Squeak images using the file list browser.

 *Yellow-click on the SBE-Quinto category and **fileOut** the contents.*

You should now find a file called "SBE-Quinto.st" in the same folder on disk where your image is saved. Have a look at this file with a text editor.

 Open a fresh Squeak image and use the File List tool to file in the SBE-Quinto.st fileout. Verify that the game now works in the new image.

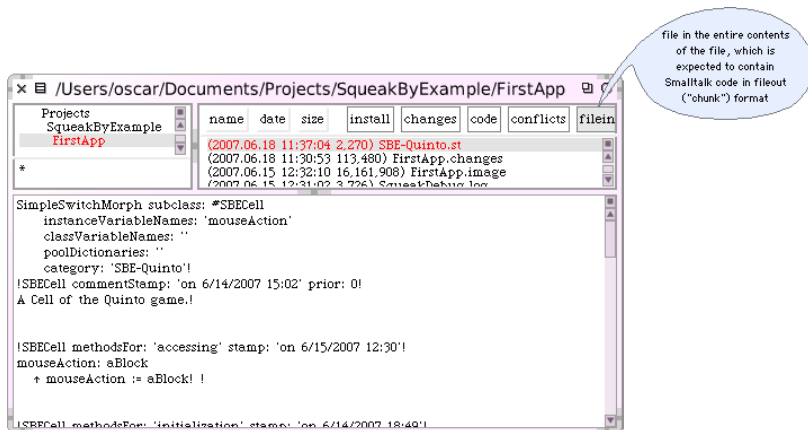


FIG. 2.14 – Filing in Squeak source code.

## Les paquetages Monticello

Although fileouts are a convenient way of making a snapshot of the code you have written, they are decidedly “old school”. Just as most open-source projects find it much more convenient to maintain their code in a repository using CVS<sup>3</sup> or Subversion<sup>4</sup>, so Squeak programmers find it more convenient to manage their code using Monticello packages. These packages are represented as files with names ending in .mcz; they are actually zip-compressed bundles that contain the complete code of your package.

Using the Monticello package browser, you can save packages to repositories on various types of server, including FTP and HTTP servers; you can also just write the packages to a repository in a local file system directory. A copy of your package is also always cached


<sup>3</sup> [www.nongnu.org/cvs](http://www.nongnu.org/cvs)

<sup>4</sup> [subversion.tigris.org](http://subversion.tigris.org)

on your local hard-disk in the *package-cache* folder. Monticello lets you save multiple versions of your program, merge versions, go back to an old version, and browse the differences between versions. In fact, it supports the same sort of operations that you are used to if you share your work using CVS or Subversion.

A good trick is to always develop in the same folder. This way you get a copy of all the code that you publish on squeaksource on your local machine. You can then backup and browse at will.

You can also send a .mcz file by email. The recipient will have to place it in her *package-cache* folder ; she will then be able to use Monticello to browse and load it.

 *Open the Monticello browser by selecting*  
World ▷ open ... ▷ Monticello browser .

In the right-hand pane of the browser (see la figure 2.15) is a list of Monticello repositories, which will include all of the repositories from which code has been loaded into the image that you are using.

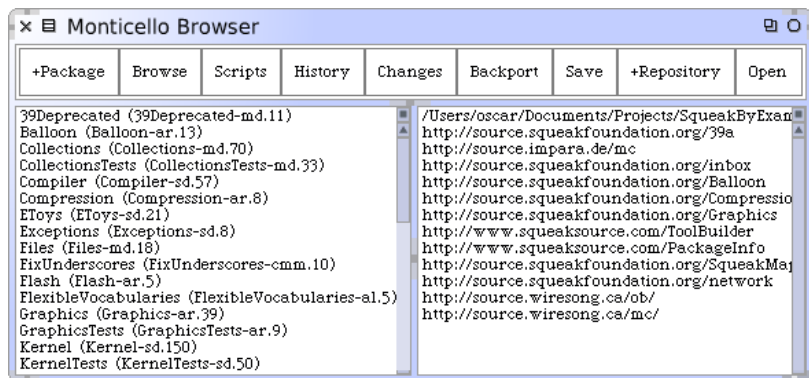


FIG. 2.15 – The Monticello browser.


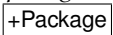
At the top of the list in the Monticello browser is a repository in a local directory called the *package cache*, which caches copies of the packages that you have loaded or published over the network. This local cache is really handy because it lets you keep your own local

history ; it also allows you to work in places where you do not have internet access, or where access is slow enough that you do not want to save to a remote repository very frequently.

## Sauvegarder et charger du code avec Monticello.

On the left-hand side of the Monticello browser is a list of packages that have a version loaded into the image ; packages that have been modified since they were loaded are marked with an asterisk. (These are sometimes referred to as dirty packages.) If you select a package, the list of repositories is restricted to just those repositories that contain a copy of the selected package.

What is a package ? For now, you can think of a package as a group of class and method categories that share the same prefix. Since we put all of the code for the Quinto game into the class category called *SBE-Quinto*, we can refer to it as the SBE-Quinto package.

 *Ajouter le paquetage SBE-Quinto à votre navigateur Monticello en utilisant le bouton* .


## SqueakSource : un SourceForge pour Squeak.

Nous pensons que la meilleure façon de sauvegarder votre code et de le partager est de créer un compte sur un serveur SqueakSource<sup>5</sup>. SqueakSource est similaire à SourceForge<sup>6</sup> : il s'agit d'un frontal 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 enregistré sur <http://www.squeaksource.com/SqueakByExample.html>. Vous pouvez consulter ce projet à l'aide d'un navigateur internet, mais il est beaucoup plus productif de le faire depuis Squeak, un outil ad-hoc appelé navigateur Monticello, qui vous permet de gérer vos paquets.

---

<sup>5</sup><http://www.squeaksource.com/>

<sup>6</sup><http://sourceforge.net/>

 Ouvrir un navigateur web à l'adresse `http://www.squeaksource.com/`. Ouvrir un compte et ensuite créer un projet (c-à-d. "register").

SqueakSource va vous montrer l'information que vous devez utiliser lorsque on ajoute un dépôt au moyen d'un navigateur Monticello.

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

 Avec le paquetage SBE-Quinto sélectionné, cliquer le bouton `+Repository` dans le navigateur Monticello.

Vous verrez une liste des différents type de dépôts disponible ; pour ajouter un dépôt SqueakSource, sélectionner le menu `HTTP`. You will be presented with a dialog in which you can provide the necessary information about the server. You should copy the presented template to identify your SqueakSource project, paste it into Monticello and supply your initials and password :

---

```
MCHttpRepository
location: 'http://www.squeaksource.com/YourProject'
user: 'yourInitials'
password: 'yourPassword'
```

---


If you provide empty initials and password strings, you can still load the project, but you will not be able to update it :

---

```
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.

 Cliquer sur le bouton `Save` pour faire une première sauvegarde de jeu Quinto 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 dépôt, que vous pouvez ouvrir avec le bouton `Open`

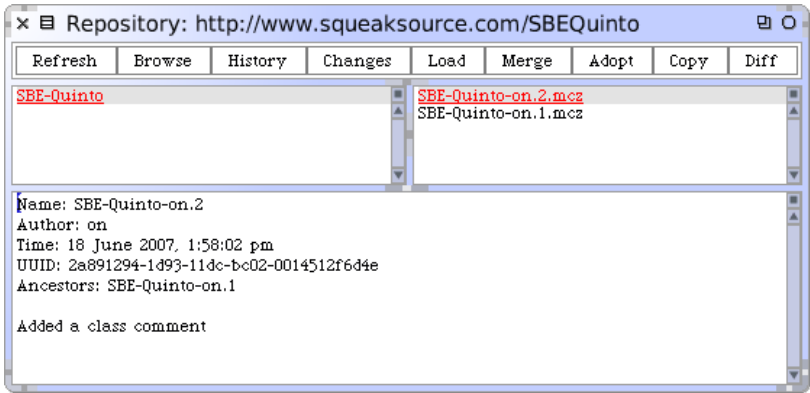



FIG. 2.16 – Parcourir un dépôt Monticello

ou le bouton jaune de la souris. Une fois que vous avez sélectionné une version, vous pouvez la charger dans votre image.

 *Ouvrir le dépôt SBE-Quinto que vous venez de sauvegarder.*

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

## 2.10 Résumé du chapitre

Dans ce chapitre, nous avons vu comment créer des catégories, classes et méthodes. Nous avons vu comment utiliser le navigateur de classes (system browser), l'inspecteur, le débogueur et le gestionnaire Monticello.

- Les catégories sont des groupes de classes qui ont à voir entre elles.
- Une nouvelle classe est créée en envoyant un message à sa super-classe.



- Les protocoles sont de groupes de méthodes qui ont à voir entre elles.
- 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 en Squeak. Vous pouvez y mettre tout code d’initialisation spécifique.
- Le débogueur procure un outil de haut-niveau pour inspecter et modifier l’état d’un programme en cours d’exécution.
- Vous pouvez partager le code source en le sauvegardant une catégorie sous forme d’un fichier.
- Une meilleure façon de partager le code est d’utiliser Monticello afin de gérer un dépôt externe, défini par exemple comme un projet SqueakSource.



## Chapitre 3

# Un résumé de la syntaxe

Squeak, comme la plupart des dialectes modernes, 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 voix haute ressemble à du pidgin :

---

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

---

La syntaxe de Squeak est minimaliste.

Pour l'essentiel conçue uniquement pour *envoyer des messages* (c-à-d. des expressions) et *des déclarations de méthodes*. Les expressions sont construites à partir d'un nombre très réduit de primitives. Seulement 6 mots clés, et pas de 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 if-then-else, Smalltalk envoie des messages comme ifTrue: à des objets Booléens . 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` et `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 closures ou fermetures lexicales - closures en anglais -, et (vi) des messages.

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

**Variables locales.** `startPoint` est un nom de variable, ou identifiant. Par convention, les identifiants sont composés de mots au format "camelCase" (c-à-d. 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.

**Variables partagées** Les identifiants qui débutent par une lettre majuscule sont des variables globales, des variables de classes, des dictionnaires 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 appelée une "pseudo-variable" puisque nous ne pouvons rien lui affecter.

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

**Les nombres flottants** peuvent être spécifiés avec leur exposant en base dix : `2.4e7` est  $2.4 \times 10^7$ .

**Les caractères.** Un signe dollar définit un littéral : `$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`, comme `Character space` et `Character tab`.

Syntaxe	ce qu'elle représente
startPoint	un nom de variable
Transcript	un nom de variable globale
self	une pseudo-variable
1	un entier decimal
2r101	un entier binaire
1.5	un nombre flottant
2.4e7	une notation exponentielle
\$a	le caractère 'a'
'Hello'	la chaine "Hello"
#Hello	le symbole #Hello
#{1 2 3}	un tableau de littéraux
{1. 2. 1+2}	un tableau dynamique
"a comment"	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 MV (Machine Virtuelle ou ann
3 factorial	un message unaire
3+4	un message binaire
2 raisedTo: 6 modulo: 10	un message à mot-clé
↑ true	retourne la valeur true
Transcript show: 'hello'. Transcript cr	un séparateur d'expression (.)
Transcript show: 'hello'; cr	un message en cascade (;)

TAB. 3.1 – Résumé de la syntaxe de Squeak

**Les chaines de caractères.** les apostrophes sont utilisées pour définir un littéral chaine. Si vous désirez une chaine comportant une apostrophe , il suffira de doubler l’apostrophe , comme dans 'G"day'.

**Les symboles** sont comme les chaines de caractères , en ce sens qu’ils comportent une suite de caractères. Mais, contrairement à une chaine, un littéral doit être globalement unique. Il y a seulement

un objet symbole #Hello mais il peut y avoir plusieurs objets chaînes de caractères ayant la valeur 'Hello'.

**Des tableaux à la compilation** 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) est un littéral de trois éléments : l'entier 27, le tableau à la compilation contenant deux booléens, et le symbole #abc.

**Des tableaux à l'exécution.** Les accolades {} définissent un (dynamique à l'exécution). Les 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. (La notation entre accolades est particulière à Squeak un dialecte de Smalltalk ! Dans d'autres Smalltalks vous devez explicitement construire des tableaux dynamiques.)

**Les commentaires** sont encadrés par des guillemets. *"hello"* est un commentaire, et non une chaîne, et donc est ignoré par le compilateur de Squeak. Les commentaires peuvent se répartir sur plusieurs lignes.

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

**Affectation.** := affecte un objet à une variable. Quelquefois vous verrez à la place une ← . Malheureusement, tant qu'elle ne sera pas un caractère ASCII, il apparaîtra sous la forme d'underscore à moins que vous utilisiez une fonte spéciale. Ainsi, x := 1 est identique à x ← 1 ou x \_ 1. Vous devrez utiliser := puisque les autres représentations ont été supprimées depuis la version 3.9 de Squeak.

**Blocs.** Des crochets [ ] définissent un bloc, aussi connu comme une closure ou une 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.

**Primitives.** <primitive: ...> code l'invocation d'une primitive de la VM (machine virtuelle (<primitive: 1> est la VM primitive de SmallInteger»+.). Tout code suivant la primitive est exécuté seulement si la primitive échoue. Depuis la version 3.9 de Squeak,

la même syntaxe est aussi employé pour des annotations de méthode.

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

**Les messages binaires** 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** 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 *message selector* `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* d'une méthode. (Il faut taper `^` pour obtenir le caractère `↑`.)

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

**Cascades.** un point virgule peut être utilisé pour envoyer une *cascade* de messages à un receveur unique. Dans `Transcript show: 'hello'` ; cr nous envoyons d'abord le message mot-clé `show: 'hello'` au receveur `Transcript`, et 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 de sorte qu'il démarre de la `super`-classe relative à la classe contenant la méthode qui utilise `super`. Pour plus de détails voir le chapitre 5.

`nil` est l'objet non défini. C'est l'unique instance de la classe `UndefinedObject`. Les variables d'instances (et les variables de classe) 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 `BlockContext` en cours d'exécution. `thisContext` est normalement pas utile à la plupart des programmeurs, mais il est essentiel pour implémenter des outils de développement comme le débogueur et il est aussi utilisé pour gérer les exceptions et les ZZZ reprises, poursuites, prolongements ZZZ.

### 3.3 Envois de messages

Il y a trois types de messages dans Squeak.

1. Les messages *unaires*, messages sans argument. `1 factorial` envoie le message `factorial` à l'objet 1.
2. Les messages *binaires* avec un seul argument. `1 + 2` envoie le message `+` avec l'argument 2 à l'objet 1.
3. Les messages à *mots-clés* qui comportent un nombre arbitraire d'arguments. `2 raisedTo: 6 modulo: 10` envoie le message comprenant les sélecteurs `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 débutent 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  $\longrightarrow$  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*  $\longrightarrow$  *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, ainsi

---

1 + 2 \* 3  $\longrightarrow$  9

---

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

---

1 + (2 \* 3)  $\longrightarrow$  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: 'hello world'.  
Transcript cr

---

Ce code enverra cr à l'objet Transcript , puis enverra show: 'hello world', et pour finir enverra un nouveau cr.

Quand une suite 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: 'hello world';
cr

```

---

Ce qui a précisément le même effet que l'exemple précédent.

### 3.4 Syntaxe relative aux méthodes

Bien que les expressions peuvent être évaluées n'importe où dans Squeak (par exemple, dans un espace de travail (workspace), dans un débogueur (debugger), ou dans un browser), les méthodes sont en principe définies dans une fenêtre du browser, ou dans débogueur. (Les méthodes peuvent aussi ZZZZZZ être rangées sur un périphérique externe ZZZZZZZZ, mais ce n'est pas la façon habituelle de programmer en Squeak.)

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 a pas de syntaxe spécifique pour créer une classe.)

Voilà la méthode `lineCount` dans la classe `String`. (La convention habituelle est de se référer aux méthode comme suit `ClassName»methodName`, ainsi nous nommerons cette méthode `String»lineCount`.)

---

#### Méthode 3.1 – *Line count*

---

```

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 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`) ; et
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 `↑` (`ZZZ` taper comme `ZZZ ^`) 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 débiter 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 `ZZZ` pointent `ZZZ` vers l'objet représentant cette classe.

### 3.5 La syntaxe des blocs

Les blocs apportent `ZZZ` un moyen de différer `ZZZ` 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 `↑`), et dans ce cas il ne retourne aucune valeur.

---

```
[ 1 + 2 ] value  →  3
```

---

Les blocs peuvent prendre des paramètres, chacun doit être déclaré avec un deux points. Une barre verticale sépare les déclarations de paramètres du corps du bloc. Pour évaluer un bloc avec un paramètre, vous devez lui envoyer le message `value:` avec un argument. Pour un bloc à deux paramètres `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 est 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. *ZZZ* Les variables locales *ZZZ* sont déclarées après chaque argument :

---

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

---

Les blocs sont en fait des *fermetures* lexicales, dès lors qu'ils peuvent se référer à des variables du contexte *ZZZ* qu'ils engendrent *ZZZ*. Le bloc suivant concerne la variable *x* *ZZZ* de son environnement *ZZZ* :

---

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

---

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

*Avertissement :*

Dans l'actuelle version (la 3.9), Squeak ne supporte pas en réalité *ZZZ* les vraies fermetures lexicales *ZZZ*, puisque les arguments des blocs sont en fait simulés comme étant des variables temporaires de la méthode qu'ils *ZZZ* contiennent *ZZZ* . Il existe un nouveau compilateur qui supporte complètement les fermetures lexicales (block closures en anglais), mais il est encore en développement et non utilisé par défaut.

Dans quelques situations ce problème peut entraîner des conflits de nommage.

Cette situation se produit car Squeak est construit sur une des premières implémentations de Smalltalk. Si vous rencontrez ce problème,

examinez `ZZZ` l'expediteur `ZZZ` de la méthode `fixTemps`, ou chargez le *Closure Compiler*.

## 3.6 Un résumé des Tests et des Itérations

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

Les tests sont obtenus par l'envoi de ces messages `ifTrue:`, `ifFalse:` ou `ifTrue:ifFalse:` au résultat d'une expression booléenne. Voir le chapitre 8 pour plus de détails sur les booléens.

---

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

---

Les boucles (ou itérations) sont obtenues typiquement par l'envoi de message à des blocs, ou 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 procédure itérative :

---

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

---

`whileFalse:` reverses the exit condition.

---

```
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 lequel alors `ZZZZ` which then acts as `ZZZZ` la valeur initiale d'un compteur de boucle. Les deux arguments sont la borne supérieure, et 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
```

---

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

Un Intervalle est une collection qui définit un itérateur sur une suite de nombre 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 envoyer lui 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 conditions, ainsi pouvez aussi itérer sur tous les caractères.

---

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

---

Finalement, vous devez garder à l'esprit que les collections acceptent aussi un style fonctionnel avec l'opérateur *fold* au travers de la méthode `inject:into:`. Cela vous amène à générer un résultat cumulatif utilisant une expression qui débute avec *ZZZ* une valeur initiale *ZZZ* et injecte chaque élément de la collection. Sommes et produits sont des exemples typiques.

---

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

---

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

Plus de détails sur les collections et les flux dans le chapitre 9 et dans le chapitre 10.

## 3.7 Primitives et Pragmas

Dans Smalltalk tout est objet, et tout est produit par l'envoi de messages. Néanmoins, à certains points nous "touchons le fond". Certains objets peuvent seulement être productif en invoquant la machine virtuelle et les primitives.

Par exemple, les objets suivants sont tous implémentés au titre de primitives : allocation de la mémoire (`new`, `new:`), manipulation des bits (`bitAnd:`, `bitOr:`, `bitShift:`), pointeur et arithmétique des entiers (`+`, `-`, `<`, `>`, `*`, `/`, `=`, `==...`), et accès aux tableaux (`at:`, `at:put:`).

Les primitives sont invoquées avec la syntaxe `<primitive: aNumber>`. Une méthode qui invoque une telle primitive peut aussi embarquer du code Smalltalk, qui sera évalué *seulement* si la primitive est en échec.

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

Méthode 3.2 – *A primitive method*


---

+ 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."*

&lt;primitive: 1&gt;

↑ super + aNumber

---

Depuis la version 3.9 de Squeak, la syntaxe avec <....> est aussi utilisée pour les annotations de méthode que l'on appelle des pragmas.

## 3.8 Résumé du chapitre

- Squeak a (seulement) six mots réservés aussi appelés *pseudo-variables* : true, false, nil, self, super, and thisContext.
- Il y a cinq types d'objets littéraux : les nombres (5, 2.5, 1.9e15, 2r111), les caractères (\$a), les chaînes ('hello'), les symboles (#hello), et les tableaux (#('hello' #hello))
- 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 obligés d'être globalement unique.
- 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, 'hi' , ' there'), and à mots-clés (par ex., 'hi' at: 2 put: \$o)
- Un envoi de message *en cascade* est une suite de messages envoyés à la même cible, tous séparés par des ; : OrderedCollection new add: #calvin; add: #hobbes; size → 2
- Les variables locales sont déclarées à l'aide de barres verticales. Employez := pour les affectations ; ← ou \_ marche aussi mais est abandonnée depuis la version 3.9 de Squeak. |x| x:=1



- Les expressions sont les messages envoyés, les cascades et les affectations, et possiblement regroupées avec des parenthèses. *Les instructions* sont des expressions séparées par des points.
- Les blocs ou clotures lexicales sont des expressions limitées par des crochets. Les blocs peuvent prendre des arguments et peuvent contenir des variables locales (temporaires). Les expressions du bloc ne sont évaluées que lorsque vous envoyez un message `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, mais seulement des messages qui sous condition 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, excepté pour les éléments syntaxiques vus au le chapitre 3 (`:=` `↑` `.` `;` `#` `()` `{}` `[:|]`), tout est un envoi de message. Comme en C++, vous pouvez définir vos opérateurs comme `+` pour vos propres classes, mais tous les opérateurs ont la même précedence. De plus, il n'est pas possible de changer l'arité d'une méthode : `-` est toujours un message binaire, et plus il n'y a pas de possibilité 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 y a juste trois formes de

messages : 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. Et 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 terminologie 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.

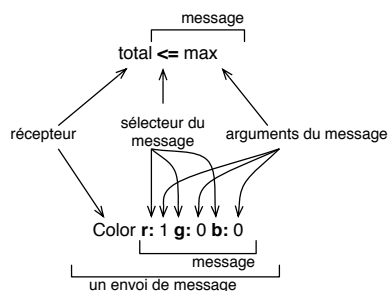


FIG. 4.1 – Deux messages composés d'un receveur, d'un sélecteur de méthode et d'un ensemble d'arguments.

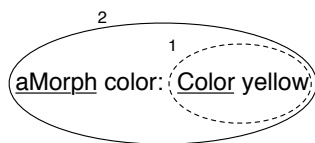


FIG. 4.2 – aMorph color: Color yellow est composé de deux expressions : Color yellow et aMorph color: Color yellow.

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

On se propose de faciliter la lecture au moyen d'une notation graphique : nous soulignons le receveur afin de vous aider à l'identifier.

Expression	Type de messages	Résultat
Color yellow aPen go: 100	unaire à mots-clés	Crée une couleur. Le stylo 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-clé et binaire	Le style receveur se déplace vers l'avant de 120 pixels.

TAB. 4.1 – Exemples de messages

Nous entourons également chaque expression dans une ellipse et numérotons les expressions à partir de la première qui va être évalué afin de voir l’ordre d’envoi des messages envoyés.

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, d’où son ellipse qui est numéroté 1. Il y a deux receveurs : aMorph qui reçoit le message color: ... et Color qui reçoit le message yellow. Chacun des receveurs est souligné.

Un receveur peut être le premier élément d’un message, comme 100 dans l’expression 100 + 200 ou Color 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 est l’objet retourné par cette expression Pen new. Dans tous les 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 forcément d’arguments.

Un message unaire comme `open` ne nécessite pas d'arguments. Les messages à mots clés simple 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 l'argument étant 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 basés sur la distinction établis 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` est un message unaire.
- *Les messages binaires* sont des messages formés avec des opérateurs (souvent arithmétiques). Ils sont binaires car ne concernant que deux objets : le receveur et l'objet argument objet. Par exemple dans `10 + 20`, `+` est un message binaire qui est envoyé au receveur `10` avec l'argument `20`.
- *Les messages à mot-clé* sont des messages formés avec plusieurs mot-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:` 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 caractère ne contenant pas 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 <i>"La classe de Object est Object class (!)"</i>

---

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

---

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 modèle syntaxique : receveur **sélecteur** argument

## Messages à mots clés

Les messages à mot-clés sont des messages qui nécessitent un ou plusieurs arguments et dont le sélecteur consiste en un ou plusieurs mot-clés se finissant par deux points `:`. Les messages à mot-clés suivent le modèle suivant : receveur **selecteur****MotUn:** 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`, qui crée 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) <i>"création d'un intervalle"</i>
Color r: 1 g: 0 b: 0	→	Color red <i>"création d'une nouvelle couleur"</i>
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 mot-clés sont des messages qui nécessitent un ou plusieurs arguments. Leurs sélecteurs consistent en un ou plusieurs mot-clés chacun se terminant par deux points (`:`). Ils suivent le schéma suivant :

receveur **selecteur****MotUn:** argumentUn **motDeux:** argumentDeux



## 4.3 Composition de messages

Les trois formes d'envoi de message 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 mot-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: un message à mot-clé, ainsi l'expression Color yellow est envoyé en premier. Néanmoins comme les expressions entre parenthèses sont envoyées en premier, mettre des parenthèses (non nécessaires) autour de Color yellow permet d'accentuer le fait qu'il doit être envoyé en premier. Le reste de cette section illustre chacun de ses différents points.

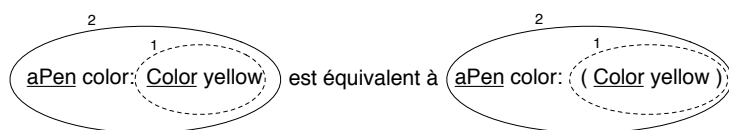


FIG. 4.3 – Les messages unaires sont envoyés en premier, on commence donc par Color yellow. Ceci retourne un objet de couleur qui est passé comme argument du message aPen color:.

### Unaire > Binaire > Mots clés

Les messages unaires sont d'abord envoyés, puis les messages binaires et enfin les messages à mots-clé. On peut 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 mot-clés.

Unaire > Binaire > Mot-clé

Comme ces exemples 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, de telle sorte que des parenthèses doivent être introduites chaque fois que l'on veut imposer un ordre de priorité entre deux opérateurs binaires :

---

1 + 2 * 3	→	9
1 + (2 * 3)	→	7

---

L'exemple suivant qui est un peu plus complexe (!) est une illustration que même des expressions Smalltalk compliqués 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 sont abstraites. Nous interrogeons la classe argument aClass, pour récupérer les clés de son dictionnaire de méthodes, et sélectionner les méthodes de la classe qui sont abstraites. Ensuite nous lions 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 nous montre quelles méthodes doivent être implémenter 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 *mot-clé* color: envoyé à aPen. Les messages unaires sont d’abord envoyés, de telle sorte que l’expression Color yellow est d’abord exécutée (1). Celle-ci retourne un objet couleur qui est passé en argument du message aPen color: aColor (2) comme indiqué dans 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

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

**Exemple.** Dans le message aPen go: 100 + 20, il y a le message *binaire* + 20 et un message à *mot-clé* go:. Les messages binaires sont d’abord envoyés avant les messages à mot-clés, ainsi 100 + 20 est envoyé en premier (1) : le message + 20 est envoyé à l’objet 100 et retourne le nombre 120. Puis le message aPen go: 120 est envoyé avec comme argument 120 (2). Exemple 4.2 montre comme l’expression est évalué.

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 mot-clé"

**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 à mot-clé et un message binaire (voir la figure 4.5).

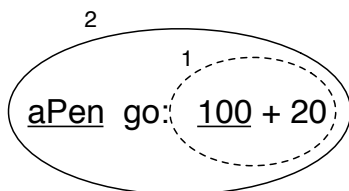


FIG. 4.4 – Les messages unaires sont envoyés en premier, ainsi Color yellow est d’abord envoyé. Ceci retourne un objet de couleur qui est passé en argument du message aPen color:.

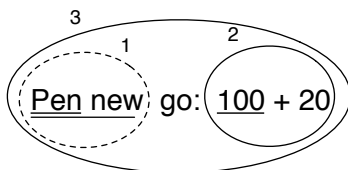


FIG. 4.5 – Décomposition de Pen new go: 100 + 20

## Les parenthèses en premier

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

(Msg) > Unaire > Binaire > Mot-clé

---

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 gauche haut est (65, 325) et dont

la taille est 134×100. 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 d’un message à mot-clé extent: qui est ensuite envoyé et qui retourne un rectangle. Finalement le message unaire center est envoyé au rectangle et le point est retourné.

Évaluer ce message sans parenthèses déclencherait une erreur car l’objet 100 ne comprends 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	"mot-clé"
	→ aRectangle	
(4)	aRectangle center	"unaire"
	→ 132@375	

## De gauche à droite

Maintenant nous savons comment les messages de différentes natures ou priorités sont traités. La question qui reste 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 exemple 4.3, où 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.

**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. Ceci retourne un nouveau stylo auquel le deuxième message down est envoyé comme il est montré dans la figure 4.6.

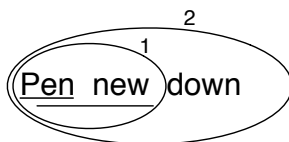


FIG. 4.6 – Décomposition de Pen new down

## Inconsistances 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. Voici des situations habituelles où des parenthèses supplémentaires sont nécessaires.

---

$3 + 4 * 5$	$\longrightarrow$	35	<i>"(pas 23) les messages binaires sont envoyés de gauche à droite"</i>
$3 + (4 * 5)$	$\longrightarrow$	23	
$1 + 1/3$	$\longrightarrow$	(2/3)	<i>"et pas 4/3"</i>
$1 + (1/3)$	$\longrightarrow$	(4/3)	
$1/3 + 2/3$	$\longrightarrow$	(7/9)	<i>"et pas 1"</i>
$(1/3) + (2/3)$	$\longrightarrow$	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 sont juste 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 on le voit dans exemple 4.4.

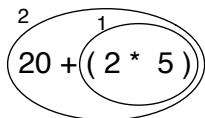
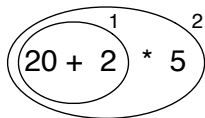
### Exemple 4.4 – Décomposer $20 + 2 * 5$

*"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 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 on le voit dans 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 \* sont juste des messages binaires, de telle sorte que \* n'a pas priorité sur +. Utiliser des parenthèses pour obtenir le résultat désiré.

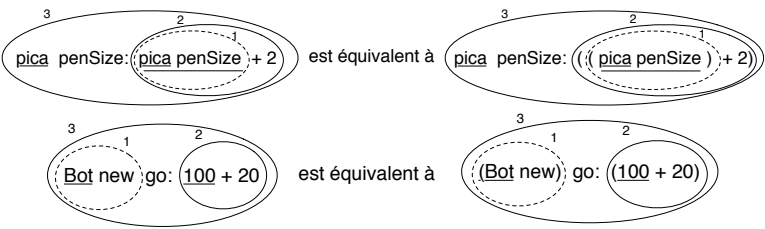


FIG. 4.7 – Messages équivalents en utilisant des parenthèses.

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

FIG. 4.8 – Des expressions et les versions équivalentes complètement parenthésées.

Notez que la première règle qui dit que les messages unaires sont envoyés avant les messages binaires ou à mot-clé 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.

### 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 mot-clé sont reconnus par le compilateur.



## Des parenthèses ou pas ?

Les caractères [, ], and (, ) délimite des zones distinctes. Dans ces zones, un message à mot-clé 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 mot-clé *local* à la zone définie.

Dans cet exemple, il y a deux mot-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 mot-clé 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 mot-clé 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 mot-clé `ifTrue: .`

---

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

---

L'expression qui suit nécessite des parenthèses car les messages `includes: et ifTrue:` sont chacun des messages à mot-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 cloture lexicale ou bloc (*c-à-d.* un objet) à partir de *expression*, qui peut être évaluée autant de fois qu'il faut (possiblement zéro), suivant le contexte.

Ainsi les branches 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* évaluée exactement une fois (en supposant que le code du son est évalué une fois).

---

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

---

## 4.5 Séquences d'expression

Les expressions (*c-à-d.* envois de message, affectation, ...) séparés par des points sont évalués 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 exceptés 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 aux même receveur en utilisant le point-virgule (;). Ceci s'appelle la *cascade* dans le jargon Smalltalk.

Expression Msg1 ; Msg2

Transcript show: ' Squeak is '.	est équivalent à :	Transcript show: 'Squeak is'; show: 'fun ';
Transcript show: 'fun '.		cr
Transcript 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 cascadié `setX:setY` est le nouveau point résultant de l'évaluation de `Point new`, et *non pas* `Point`. Le message qui suit `isZero` est envoyé au même receveur.

---

Point new setX: 25 setY: 35; isZero     $\longrightarrow$     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 **sélecteur** argument
- Les messages à mots-clés sont des messages qui concernent plus d'un objet et qui contiennent au moins un caractère deux points (:).  
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 du même type, l'ordre d'évaluation est de gauche à droite.
- En Smalltalk, les opérateurs arithmétiques traditionnels comme + ou \* ont la même priorité. + et \* sont juste des messages binaires, de telle sorte que \* n'a pas priorité sur +. Vous devez utiliser les parenthèses pour obtenir un résultat différent.

Deuxième partie

# Développer avec Squeak



## 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 envoi de messages (nous disons aussi transmission de messages). Cependant, ces caractéristiques de simplicité et d'homogénéité peuvent être source de quelque difficulté 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 appliqués de manière *uniforme*. Les règles s'énoncent comme suit :

**Règle 1.** Tout est un 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 un objet

Le mantra "tout est un objet" est hautement contagieux. Après seulement peu de temps passé avec Smalltalk, vous serez progressivement surpris par comment 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 feriez avec n'importe quel autre objet.

---

$3 + 4$	→	7	<i>"envoie '+' à 4 à 3, donnant 7"</i>
20 factorial	→	2432902008176640000	<i>"envoie factorial, donnant un grand nombre"</i>

---

La représentation de 20 factorial est certainement différente de la représentation de 7, mais aucune partie du code — pas même l'implémentation de factorial<sup>1</sup> — 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 Squeak est vraiment un système réflexif offrant une grande expressivité aux développeurs.

---

<sup>1</sup> En anglais, factoriel.



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 références ; il y a (2) *les petits entiers* <sup>2</sup> qui sont passés par valeur, et enfin, il y a (3) les objets indexables 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 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. Une classe 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 méta-classe automatiquement. La méta-classe définit la structure et le comportement de la classe qui est son instance. 99% du temps vous n'aurez pas à penser aux méta-classes et vous pourrez joyeusement les ignorer. (Nous porterons notre attention aux métaclasses dans le chapitre 11.)

---

<sup>2</sup>En anglais, small integers.

## 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 qu'elle autre instance de la même classe. Nous disons que l'*espace d'encapsulation*<sup>3</sup> 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 méta-programmation 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 préférons 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.

### 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`.

---

<sup>3</sup>En anglais, *encapsulation boundary*.

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 implémentations 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 méthode 5.1 fonctionnera toujours.

## Les méthodes

Toutes les méthodes sont publiques <sup>4</sup>. Les méthodes sont regroupées en protocoles qui indique 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.

---

<sup>4</sup>En fait, presque toute. En Squeak, 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.

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.

## Le côté instance et le côté classe

Puisque les classes sont des objets, elles peuvent avoir leur propre variables d'instance ainsi que leur propre 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 méta-classe. Quant aux méthodes de classe, elles correspondent juste aux méthodes définies par une méta-classe.

Une classe et sa méta-classe 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.

De ce fait, le navigateur de classes nommé System Browser vous aide à parcourir à la fois classes et méta-classes comme si elles n'étaient qu'une seule entité avec deux "côtés" : le "côté instance" et le "côté classe", comme le montre 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 défini côté classe de la classe Color et non du côté instance.

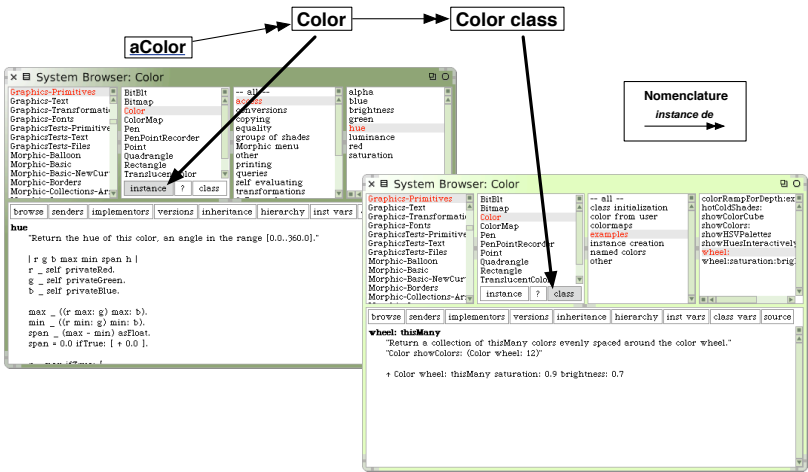


FIG. 5.1 – Naviguer dans une classe et sa méta-classe.

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

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 méta-classe correspondante. Vous pouvez naviguer dans la méta-classe en cliquant sur le bouton `class`. Du patron employé pour la création de la méta-classe, seule la liste de 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 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 méta-classe (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 trouverez occasionnellement des méthodes de classe utilisées d'une autre manière.

Il est coutumier 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 guillemets englobant le commentaire "Color showColorCube" et tapez 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 dynamique <sup>5</sup> 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.

---

<sup>5</sup>En anglais, dynamically-dispatched methods.

## 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`<sup>6</sup> afin de suivre le nombre d'instances 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 `Hyena`<sup>7</sup> 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: 'SBE-CIV'
```

```
Dog class
  instanceVariableNames: 'count'
```

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

---

<sup>6</sup>En français, compteur.

<sup>7</sup>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 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 conception <sup>8</sup> nommé Singleton<sup>9</sup> offre un exemple-type de l'usage de variables d'instance de classe et de méthodes de classe. Imaginez que nous souhaitons 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'
```

---

---

<sup>8</sup>En anglais, nous parlons de *Design Patterns*.

<sup>9</sup>?,.

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 conséquent, 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 `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 `WebServer uniqueInstance` est exécuté, 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.

---

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

---

Traditionnellement, la racine de la hiérarchie d'héritage en Smalltalk est la classe `Object` ("Objet" en anglais ; puisque tout est un objet). En Squeak, 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 hérite 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 œil au protocole `Kernel-Classes` ▸ `Class` ▸ `subclass creation` pour voir desquelles il s'agit.

Bien que Squeak ne dispose pas d'héritage multiple, il incorpore depuis la version 3.9 un mécanisme appelé *traits*<sup>10</sup> 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 envoient, mais qui ne sont pas définies elles-même — 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`<sup>11</sup>. Ceci est connue 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

---

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

<sup>11</sup>Dans le sens, laissée à la responsabilité de la sous-classe.

Magnitude devraient implémenter les méthodes <, = et hash <sup>12</sup>. 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 de l'argument".*

---

Character»< aCharacter

*"Answer true if the receiver's value < aCharacter's value."*

↑self asciiValue < aCharacter asciiValue

---



---

<sup>12</sup>Relatif au code de hachage.

## 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.

---

### Classe 5.10 – Définir un nouveau trait

---

```
Trait named: #TAuthor
  uses: { }
  category: 'SBE-Quinto'
```

---

Nous définissons ici le *trait* TAuthor dans la catégorie *SBE-Quinto*. Ce *trait* n'utilise<sup>13</sup> 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 SBEGame que nous avons définie dans le chapitre 2. Nous n'avons qu'à modifier le patron de création de la classe SBEGame pour y inclure cette fois l'argument-clé *uses*: suivi du *trait* à utiliser : TAuthor.

---

<sup>13</sup> Terme anglais : *uses* : il signifie "utilise".

---

 Classe 5.12 – Utiliser un *trait*


---

```
BorderedMorph subclass: #SBEGame
  uses: TAuthor
  instanceVariableNames: 'cells'
  classVariableNames: "
  poolDictionaries: "
  category: 'SBE-Quinto'
```

---

Si nous instantions maintenant SBEGame, l'instance répondra comme prévu au message author.

---

```
SBEGame 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.

Les *traits* sont employés dans le noyau du système <sup>14</sup>. 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*).

---

<sup>14</sup>System kernel.

## 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 terminologique 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 *demandeur* 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.

L'une des conséquences du modèle de transmission de message 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 <sup>†</sup>.

<sup>†</sup>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 message 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 message. 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 message. 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 <sup>†</sup>.

<sup>†</sup>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 ?*
- *Qu'elle 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 rusent 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`<sup>15</sup> :

---

```
anEllipse defaultColor  →  Color yellow
```

---

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

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

---

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

---

A contrario, si nous transmettons 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). Si le mode MVC est actif alors lui fournir une fenêtre Morphic.”.*

---

```
Morph»openInWorld
  "Add this morph to the world. If in MVC, then provide a Morphic window for it.
  "
  self couldOpenInMorphic
    ifTrue: [self openInWorld: self currentWorld]
    ifFalse: [self openInMVC]
```

---



---

<sup>15</sup>Yellow est la couleur jaune.

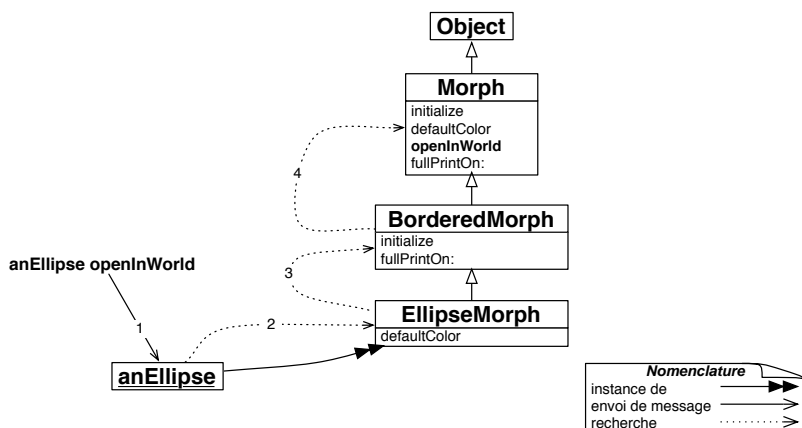


FIG. 5.2 – Recherche par référencement d’une méthode suivant la hiérarchie d’héritage.

## 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* un message avec une valeur — qui est, bien entendu, un objet. La réponse peut être définie par la construction en `↑` dans la méthode, mais si l’exécution atteint la fin de la méthode sans exécuter un `↑`, la méthode retournera toujours une valeur : l’objet receveur lui-même. Nous disons habituellement que la méthode “renvoie `self`”, parce qu’en Smalltalk la pseudo-variable `self` représente le receveur du message. En Java, nous utilisons `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. If in MVC,
  then provide a Morphic window for it."
  self couldOpenInMorphic
    ifTrue: [self openInWorld: self currentWorld]
    ifFalse: [self openInMVC].
  ↑ self    "Don't do this unless you mean it"

```

---

Pourquoi écrire `↑ self` explicitement n’est pas une bonne chose à faire ? Parce que, quand vous renvoie quelque chose explicitement, 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”<sup>16</sup> :

Renvoyez une valeur seulement quand votre objet expéditeur en a l’usage <sup>†</sup>.

<sup>†</sup>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.

---

<sup>16</sup>?, .

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

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 transmissions 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 la transmission sur `self` .

Comment la transmission sur `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 la transmission sur `super` se produit.

Remarquez que `super` n'est *pas* la supre-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 `initString`, que nous pouvons envoyer à n'importe quel morph :

---

```
anEllipse initString  → '(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 initString` trouvera la méthode `initString` dans la classe `Morph`, comme vu dans la figure 5.3.

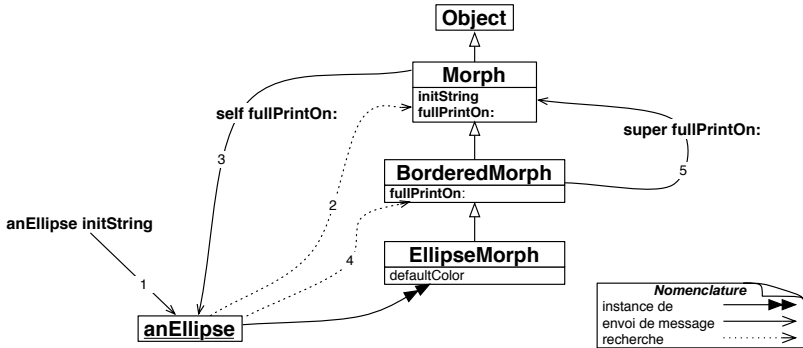
### Méthode 5.18 – Une transmission sur `self`

---

```
Morph»initString
  ↑ String streamContents: [:s | self fullPrintOn: s]
```

---

La méthode `Morph»initString` envoie `fullPrintOn:` à `self` , ce qui entraîne une seconde recherche, démarrant dans la classe `EllipseMorph`, trouvant `fullPrintOn:` dans `BorderedMorph` (voir la figure 5.3 encore une fois). Ce qu'il faut noter, c'est que la transmission sur `self` fait démarrer encore la recherche de méthode dans la classe du receveur, *c-à-d.* la classe de `anEllipse`.

FIG. 5.3 – Les transmissions sur `self` et `super`.

Une transmission sur `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 la transmission sur `super` pour étendre le comportement `fullPrintOn:` hérité de sa super-classe. Parce qu'il s'agit d'un envoi sur `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 !



Une transmission sur `super` déclenche un départ de recherche *statique* 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 la transmission sur `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 transmettions 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'objet 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:`<sup>17</sup>. 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.

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

---

<sup>17</sup>Le nom du message peut se traduire par : *ne comprend pas*.

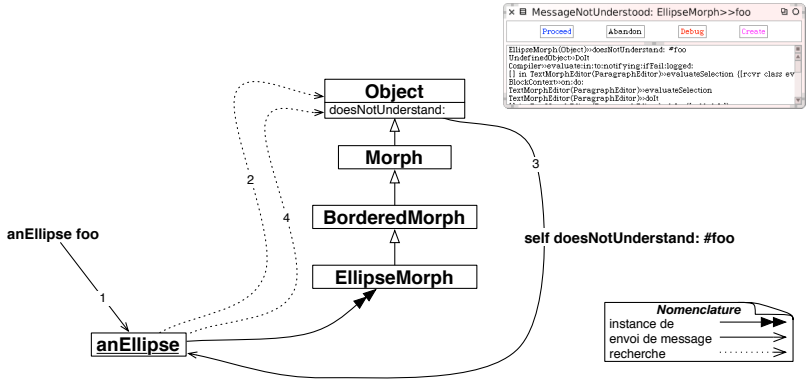


FIG. 5.4 – Le message foo n'est pas compris (not understood).

sous-classe de Object en étendant ses possibilités en offrant une façon différente de capturer l'erreur.

En fait, nous nous simplifions la vie en implémentant une délégation automatique de messages d'un objet à un autre. Un objet Delegator peut simplement déléguer tous messages qu'il ne comprend pas à un autre objet responsable 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 couverts 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 capitale (majuscule), pour nous informer qu'elles sont partagées entre plusieurs objets.

## Les variables globales

En Squeak, 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: 'Squeak est extra' ; cr
```

---

Avant vous lanciez la commande `do it`, ouvrez un transcript en faisant un glisser-déposer (drag) d'une fenêtre *Transcript* depuis l'onglet (flap) *Tools* <sup>18</sup>.

**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)
```

---



---

<sup>18</sup>Outils en français.

et

---

(Smalltalk at: #Smalltalk) == Smalltalk     $\longrightarrow$     true

---

- Sensor est une instance of EventSensor. Il représente les entrées interactives ou interfaces de saisie (en anglais, *input*) dans Squeak. 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.
- Undeclared<sup>19</sup> 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     $\longrightarrow$     #'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 Squeak devait être

---

<sup>19</sup>Non déclaré, en français.

reprogrammé à zéro <sup>20</sup>, 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 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.

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 tous les deux à ces variables partagées.

---

<sup>20</sup>Le terme anglais est : *from scratch*, signifiant depuis le début.

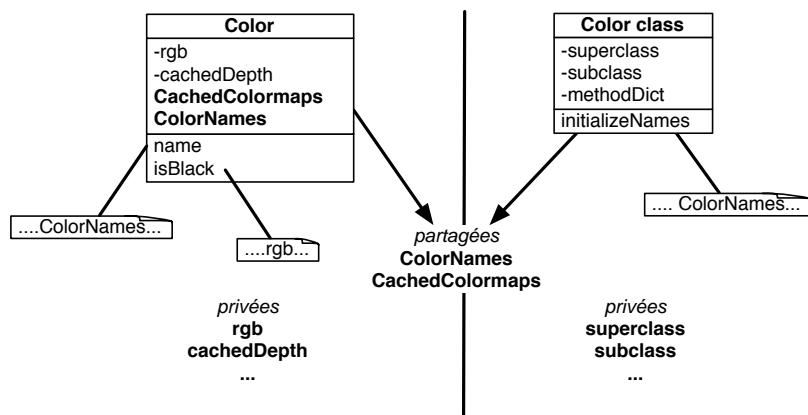


FIG. 5.5 – Des méthodes d'instance et de classe accédant à différentes variables.

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 lève 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 de 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 `initialize` 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* <sup>21</sup> 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 les variables de pool suffisamment pour les comprendre quand vous les rencontrer 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.

---

<sup>21</sup> Pool signifie piscine en anglais, ces variables sont dans un même bain !



Méthode 5.24 – *Text»testCR*


---

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 Squeak est à la fois simple et uniforme. Tout est un objet, et quasiment tout se passe via l'envoi de messages.

- Tout est un 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 méta-classe. 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. Les classes que vous définissez, cependant, 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 Squeak 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 méthode suit la chaîne d'héritage ; Les transmissions sur `self` sont dynamiques et la recherche de méthode démarre dans le receveur de la classe, alors que celles sur `super` sont statiques et la recherche commence dans la super-classe de la classe dans laquelle l'envoi sur `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 Squeak

L'objectif de ce chapitre est de vous montrer comment développer des programmes dans l'environnement de programmation de Squeak. Vous avez déjà vu comment définir des méthodes et des classes en utilisant le navigateur de classe, System Browser. Ce chapitre vous présentera plus de caractéristiques du System Browser ainsi que de nouveaux navigateurs.

Bien entendu, vous pouvez occasionnellement rencontrer des situations dans lesquelles votre programme ne marche pas comme voulu. Squeak 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 session de débogage 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. La technique plus récente en Squeak 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 Squeak ; 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 auto-hébergé et tout le développement en Smalltalk se déroulait dessus. Commençons par jeter un œil sur les principaux outils de Squeak, tous pouvant être glissé-déposé (drag) depuis l'onglet *Tools* dans l'image *Squeak-dev* (voir la section 1.1). Selon vos réglages personnels, l'onglet *Tools* pourra être ouvert par déplacement de la souris avec ou sans clic sur l'onglet orange dans le côté droit de la fenêtre principale de Squeak.

- 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-chaîne de caractères.

- Le **Method Finder** vous permet aussi de trouver des méthodes mais, soit selon leur *action*, 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 1.4.
- 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**<sup>1</sup>. L'utilisation de Workspace a aussi fait l'objet d'une étude dans la section 1.4.

L'outil **Debugger** a un rôle évident, mais vous découvrirez qu'il a une place plus centrale en comparaison des débogueurs dans d'autres langages de programmation car en Smalltalk vous pouvez *programmer* dans Debugger. Il n'est pas lancé depuis un menu ou via l'onglet *Tools* ; il apparaît normalement en situation d'erreur, en tapant **CMD-.** pour interrompre un processus lancé ou encore en insérant une expression **self halt** dans le code.

## 6.2 Le System Browser

En fait, Squeak dispose de nombreux navigateurs : le standard System Browser, le Package Browser, Omnibrowser et le Refactoring Browser. Nous explorerons tout d'abord le System Browser classique. Les autres sont des variations de celui-ci. Nous voyons dans la figure 6.1 le navigateur tel qu'il apparaît lorsque vous le glissez depuis l'onglet *Tools*.

---

<sup>1</sup>En anglais, *do it* correspond à l'exclamation "fais-le !".

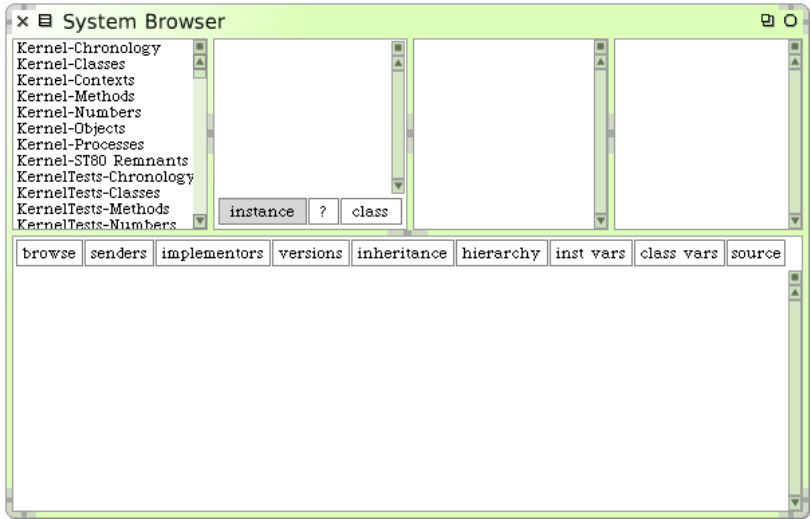


FIG. 6.1 – Le navigateur de classes : System Browser.

Les quatre petits panneaux en haut du Browser représente 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 *catégories* des classes ; sélectionnez-en une (disons *Kernel-Objects*) et alors le panneau immédiatement à droite affichera toutes les classes incluses dans cette catégorie.

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 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 le découpant en petites divisions cohérentes. Le quatrième panneau montre les noms de

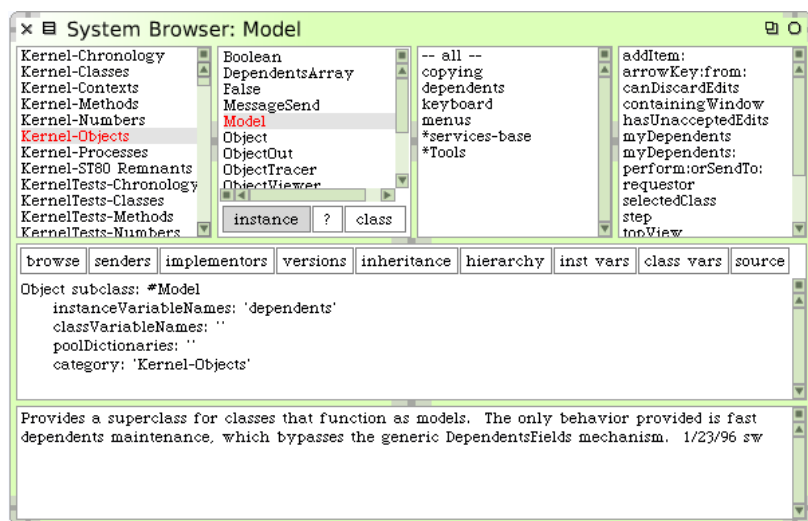


FIG. 6.2 – Le System Browser avec la classe Model sélectionnée.

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 catégories système et les protocoles de message ne sont que des convenances 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 une nouvelle catégorie ou un

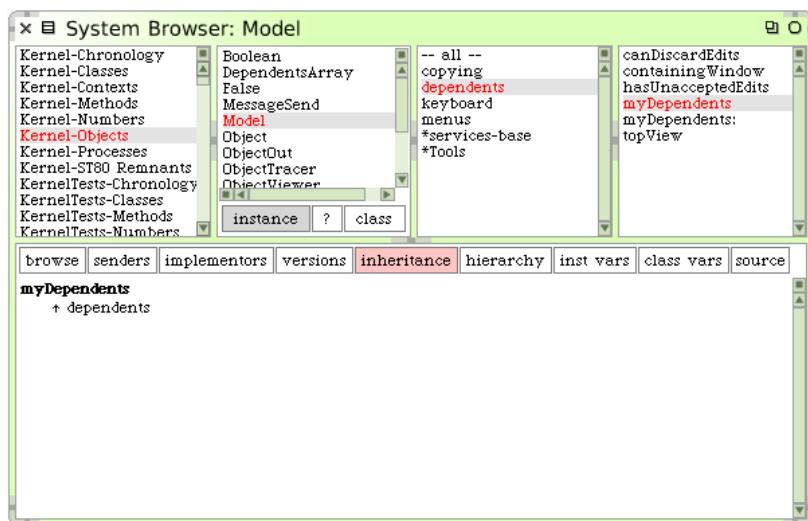


FIG. 6.3 – Le System Browser affichant la méthode `myDependents` de la classe `Model`.

nouveau protocole est différent de la manière avec laquelle vous créez une nouvelle classe ou une nouvelle méthode. Pour créer une nouvelle catégorie, sélectionnez `new category` via le menu contextuel accédé avec le bouton jaune dans le panneau des catégories ; pour créer un nouveau protocole, sélectionnez `new protocol` via le menu accédé avec le même bouton dans le panneau des protocoles. Entrez le nom de la nouvelle entité (catégorie ou protocole) dans la zone de saisie, et voilà ! Une catégorie ou un protocole, ça n'est qu'un nom et son contenu.

À l'opposé, créer une classe ou une méthode nouvelle nécessite l'écriture de code Smalltalk. Si vous désélectionnez la classe actuellement sélectionnée de manière à ce qu'aucune classe ne soit sélectionnée, le panneau principal affichera un patron de création de classe (voir la figure 6.4). Vous créez une nouvelle classe en éditant ce patron ou *template*. Pour se 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



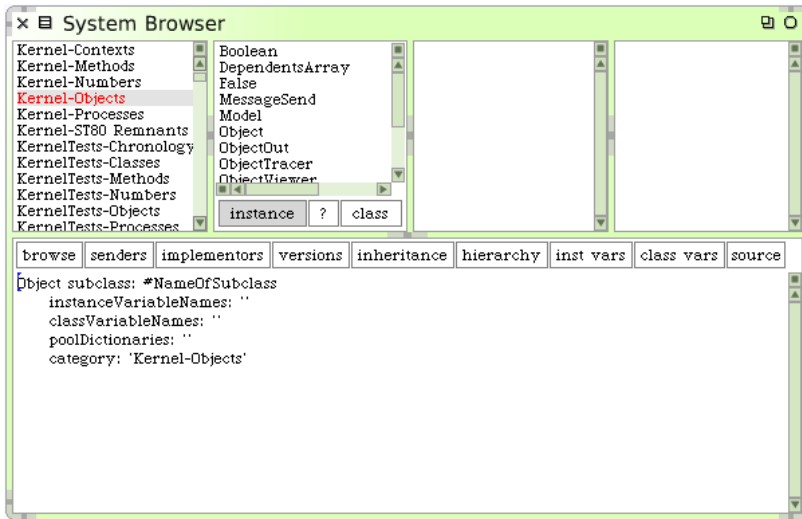


FIG. 6.4 – Le System Browser montrant le patron de création de classe.

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 actuellement sélectionnée mais vous pouvez changer celle-ci aussi si vous le désirez. 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 utilisant le menu du bouton jaune dans le panneau des classes et en sélectionnant `more ... ▸ 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 méta-classe 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.

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 montré par la figure 6.5.

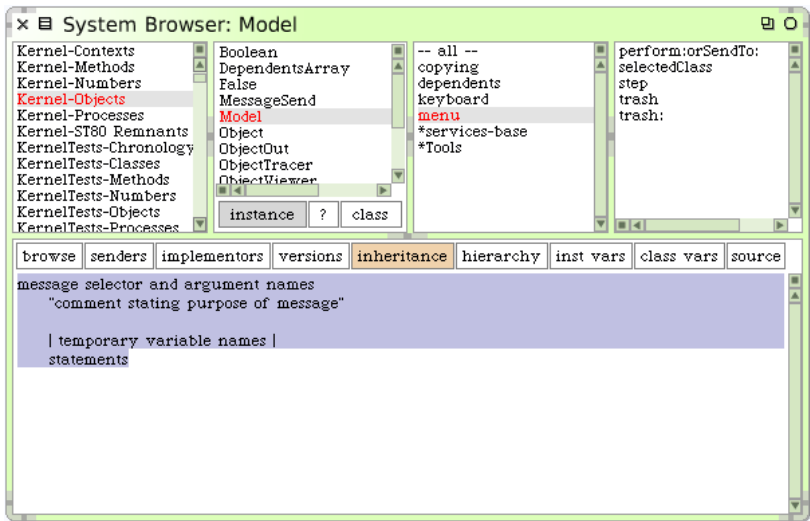



FIG. 6.5 – Le System Browser montrant le patron de création de méthode.

## La barre de boutons

Le System Browser fournit plusieurs outils pour l'exploration et l'analyse de code. Ces outils sont accédés le plus communément via la barre de boutons horizontale au milieu de la fenêtre du navigateur. Les boutons sont labélisés `browse`, `senders`, `implementors`... la figure 6.5 montre la liste complète.

## Naviguer dans le code

Le bouton `browse` ouvre une nouvelle fenêtre de navigateur sur la classe ou la méthode sélectionnée. Il est souvent utile d'avoir plusieurs navigateurs ouverts au même moment. Quand vous écrivez du code, vous aurez presque toujours besoin d'au 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 tout aussi bien ouvrir un navigateur 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, `ScaleMorph`), 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

Le bouton `senders` vous renvoie à une liste de toutes les méthodes pouvant utiliser la méthode sélectionnée. En prenant le navigateur ouvert sur `ScaleMorph`, cliquez sur la méthode `checkExtent` : dans le panneau des méthodes dans le coin supérieur droit du navigateur ; le corps de `checkExtent` : affiché dans sa partie inférieure. Si vous appuyez maintenant sur le bouton `senders`, un menu apparaîtra avec `checkExtent` : comme premier élément de la pile, suivi de tous les messages que `checkExtent` : envoie (voir la figure 6.6). Sélectionner un message dans ce menu ouvrira un navigateur avec la liste de toutes les méthodes dans l'image qui envoie le message choisi.

Le bouton `implementors` fonctionne de la même manière mais, au lieu de renvoyer une liste de `senders` d'un message (ou méthodes-envoyeuses), il sort toutes les classes qui implémentent une méthode avec le même sélecteur. Pour le voir, sélectionnez `drawOn` : dans le panneau des méthodes puis affichez le navigateur "`implementors of drawOn :`", soit en utilisant le bouton `implementors`, soit via le bouton jaune, soit encore en tapant simplement `CMD-m` (pour `implementors`) avec la méthode `drawOn` : sélectionnée dans le panneau des méthodes.

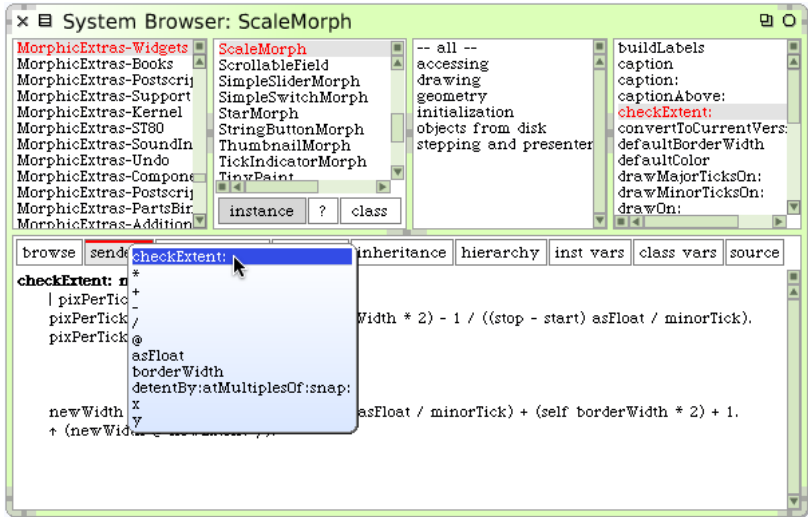


FIG. 6.6 – Un navigateur de classes ouvert sur la classe `ScaleMorph`. Notez la barre horizontale de boutons en son centre ; nous appuyons ici sur le bouton `senders`.

Vous devriez avoir une fenêtre à ascenseur montrant une liste des 96 classes implémentant une méthode `drawOn:`. Il n'y a rien de surprenant à ce qu'autant de classes implémentent cette méthode : `drawOn:` est le message compris par tout objet apte à se dessiner lui-même sur l'écran. Essayez de naviguer dans la liste des senders du message `drawOn:` ; nous nous trouvons face à 63 méthodes émettrices. Vous pouvez aussi ouvrir un navigateur d'implementors chaque fois que vous sélectionnez un message (en incluant les arguments s'il s'agit d'un message à mots-clés) puis que vous appuyez sur `CMD-m`.

Si vous regardez l'envoi de `drawOn:` dans `AtomMorph>drawOn:`, vous verrez que c'est un envoi de `super`. Ainsi nous savons que la méthode exécutée sera dans la super-classe de `AtomMorph`. Quelle est cette classe ? Cliquez sur le bouton `hierarchy` (pour hiérarchie) et vous saurez qu'il s'agit de `EllipseMorph`.

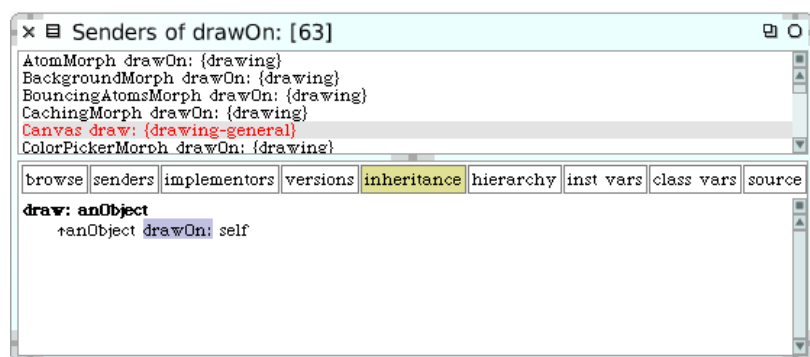


FIG. 6.7 – Le navigateur Senders Browser montrant que la méthode Canvas»>draw envoie le message drawOn: à son argument.

Maintenant observons le cinquième *sender* de la liste, Canvas»>draw, comme le montre la figure 6.7. 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 méthodes émettrices) nous montre exactement ce que son nom suggère : tous les envois d’un message ayant le sélecteur choisi. N’importe comment, le bouton `senders` 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 devrait être semblable.

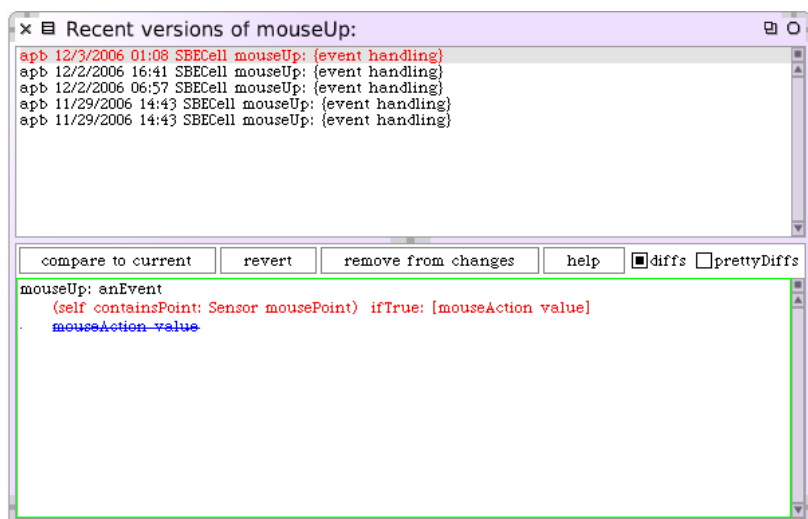


FIG. 6.8 – Le Versions Browser montre plusieurs versions de la méthode SBCell»mouseUp:.

### Les versions d'une méthode

Quand vous sauvegardez une nouvelle version d'une méthode, l'ancienne version n'est pas perdue. Squeak garde toutes les versions passées et vous permet de comparer les différentes versions entre elles et de revenir (en anglais, "revert") à une ancienne versions. Le bouton `versions` donne accès aux modifications successives effectuées sur la méthode sélectionnée. Dans la figure 6.8 nous pouvons voir les versions de la méthode `mouseUp:` qu'un des auteurs a créée lors de l'écriture du jeu de *Quinto* décrit dans le chapitre 2.

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 ; quelque soit la version sélectionnée affichée dans le panneau inférieur. Si le bouton (checkbox) `diffs` est sélectionné,

comme c'est le cas dans la figure 6.8, les différences entre la version sélectionnée et celle qui la précède immédiatement sont affichées. 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 bouton `prettyDiffs` est utile s'il y a eu changement dans la mise en pages : il affiche en mode *pretty-print* (affichage élégant) à la fois les versions antérieures et choisies de façon à ce que les différences liées au formatage ne soient pas prises en compte.

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 Smalltalkiens<sup>2</sup> accorde une extrême importance à la lisibilité du code.

**ASTUCE** *Qu'en est-il du cas où vous décidez de revenir à une méthode que vous avez entièrement effacée ? Vous pouvez trouver l'effacement dans un change set dans lequel vous pouvez demander à visiter les versions via le menu du bouton jaune. Le navigateur de change set est décrit dans la section 6.8*

## Les surcharges de méthodes

Le bouton `inheritance` ouvre un navigateur spécialisé affichant toutes les méthodes surchargées par la méthode affichée. Pour voir son fonctionnement, affichez la méthode `ScaleMorph»defaultColor` et cliquez sur `inheritance`. La définition de cette méthode surcharge `RectangleMorph»defaultColor`, elle-même surchargeant `Morph»defaultColor`, comme montré dans la figure 6.9. La couleur du bouton `inheritance` dépend de comment s'opère la surcharge. Les couleurs sont expliquées dans le ballon d'aide ou *help balloon* :

---

<sup>2</sup>En anglais, nous les appelons *Smalltalkers*.

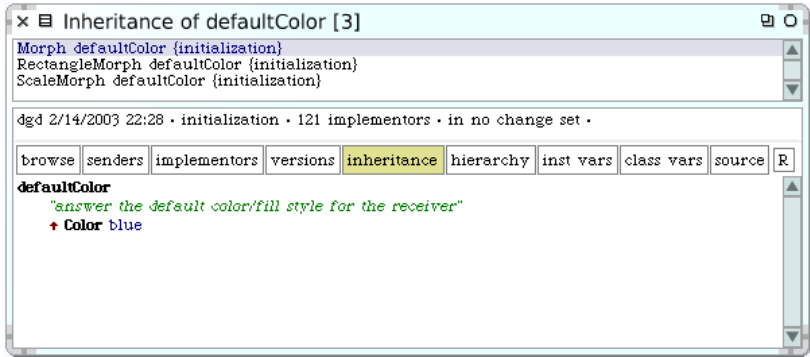


FIG. 6.9 – ScaleMorph»defaultColor et les méthodes qu’il surcharge dans l’ordre hiérarchique d’héritage. Le bouton `inheritance` est de couleur orange parce que la méthode affichée est surchargée dans une sous-classe.

- rose* : la méthode affichée surcharge une autre méthode mais ne l’utilise pas ;
- vert* : la méthode affichée surcharge une autre méthode et l’utilise via `super` ;
- or* : la méthode affichée est elle-même surchargée dans une sous-classe ;
- saumon* : la méthode affichée surcharge une autre méthode et est elle-même surchargée ;
- mauve* : la méthode affichée surcharge, est surchargée et émet un envoi sur `super`.

Notez qu’il y a deux versions de ce navigateur. Si vous utilisez la version du System Browser basée sur la librairie applicative (ou framework) OmniBrowser, le bouton `inheritance` ne change pas de couleur et Inheritance Browser a une apparence différente. Il propose aussi plus d’informations en n’affichant pas seulement les méthodes de la chaîne d’héritage mais aussi les méthodes apparentées (ou frères <sup>3</sup>) comme le montre la figure 6.10.

<sup>3</sup>En anglais, siblings.



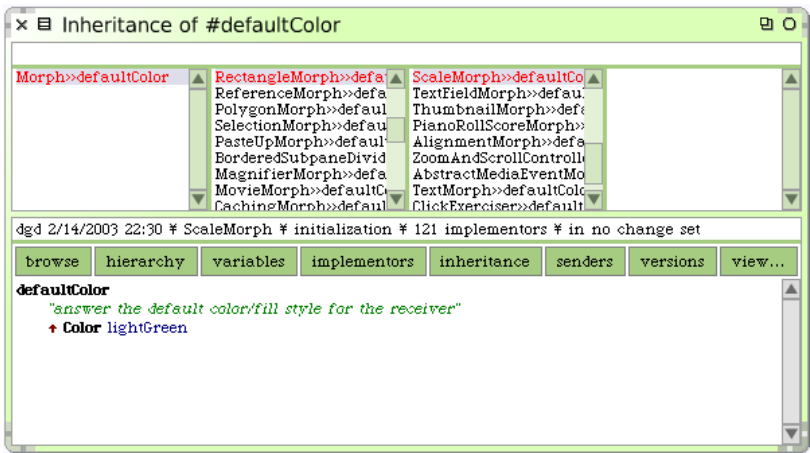


FIG. 6.10 – ScaleMorph»defaultColor et les méthodes qu’il surcharge, comme nous pouvons le voir avec le nouveau Inheritance Browser basé sur OmniBrowser. Les méthodes apparentées des méthodes sélectionnées sont montrées dans des listes.

Le navigateur hiérarchique

Le bouton `hierarchy` ouvre un navigateur hiérarchique ou Hierarchy Browser sur la classe actuelle ; ce navigateur peut aussi être ouvert en utilisant `browse hierarchy` dans le menu du panneau de classe. Le Hierarchy Browser se présente comme le System Browser à cette exception près qu’il affiche une simple liste de classes indentées pour représenter l’héritage là où le navigateur de classes classique affiche les catégories et les classes dans chaque catégorie. La catégorie de la classe sélectionnée apparaît en annotation dans un panneau supérieur horizontal. Le navigateur hiérarchique est destiné à faciliter la navigation au travers de la hiérarchie d’héritage mais, au lieu de montrer toutes les classes du système, il affiche seulement les super-classes ou sous-classes de la classe initiale. Dans la figure 6.11, le navigateur hiérarchique nous informe que RectangleMorph est la super-classe directe de ScaleMorph.

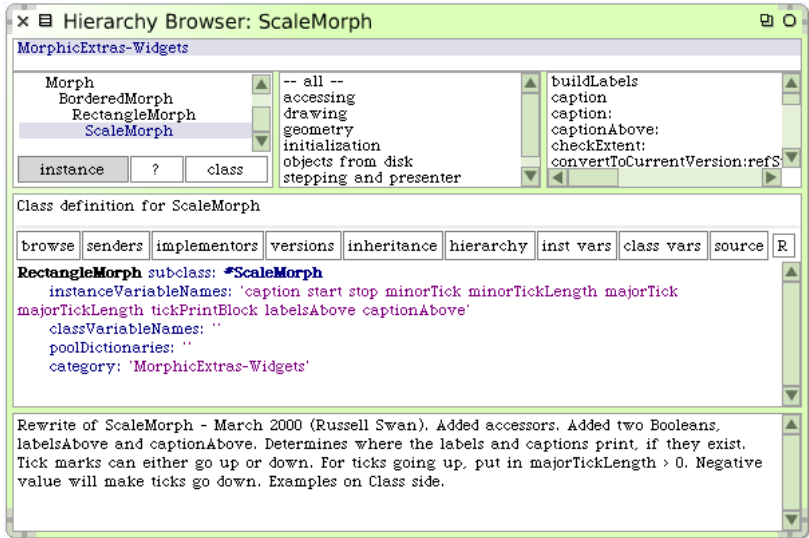


FIG. 6.11 – Un Hierarchy Browser ouvert sur ScaleMorph.

## Trouver les références aux variables

Les boutons `inst vars` et `class vars` vous aide à savoir dans quelles méthodes sont utilisées respectivement les variables d'instance et les variables de classe ; cette information est aussi disponible depuis `inst var refs` et `class var refs` du menu contextuel accessible via le bouton jaune dans le panneau de la classe. Le menu permet aussi d'afficher le jeu des références aux variables d'instance qui affecte la variable choisie par `inst var defs`. Une fois que vous avez cliqué sur le bouton ou que vous avez choisi une des propositions de menu, un menu flottant s'affichera, vous invitant ainsi à sélectionner une variable parmi toutes les variables définies et héritées dans la classe courante. La liste suit l'ordre d'héritage ; il peut d'ailleurs être utile d'afficher cette liste à chaque fois vous avez besoin de vous remémorer le nom d'une variable d'instance. Si vous cliquer en dehors de la liste, cette dernière disparaîtra sans avoir affiché le navigateur de variable.

Signalons que nous pouvons accéder à `class vars`, grâce au menu de panneau de classes accessible par le bouton jaune de la souris, ouvrant donc un inspecteur affichant les variables de classe de la classe actuelle ainsi que *leurs valeurs* ; ou encore à `class refs (N)` affichant une liste de toutes les méthodes référant directement cette même classe.

## Le panneau source

Le bouton `source` affiche un menu que nous pourrions appeler “ce qui est à voir” ; il nous permet de choisir ce que le navigateur affiche dans le panneau inférieur ou panneau de source. Parmi les propositions, nous avons l’affichage du code `source`, du code source en mode `prettyPrint` (affichage élégant), du code compilé ou `byteCodes` ou encore du code source décompilé depuis les *bytes codes* via `decompile`. Le label du bouton change pour afficher le mode choisi. Il y a d’autres options ; si vous promenez la souris sur ces options, vous verrez apparaître un ballon d’aide (ou *help balloon*). Essayez-en quelques-uns.

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 sauvegarde. 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 `byteCodes` dans l’un et `decompile` dans l’autre est vraiment une bonne manière d’en apprendre plus sur le jeu d’instruction codé (dit aussi *byte-codé*) de la machine virtuelle Squeak.

## La refactorisation

Avez-vous noté la petit bouton `R` au bout de la barre de boutons ? <sup>4</sup> Ce bouton très discret donne accès à une des techniques les plus importantes et les plus puissantes de l’environnement de Smalltalk. En cliquant sur `R`, vous obtenez une hiérarchie de menus pour refacto-

---

<sup>4</sup>Ceci dit, vous n’observerez pas ce bouton sans que les paquetages AST et RefactoringBrowser ne soit installé. Ces paquetages sont accessibles sur SqueakSource ou dans une image *Squeak-dev*.

riser (*c-à-d.* réusiner <sup>5</sup>) votre code. Il existe d'autres façons d'obtenir l'outil de refactorisation. Par exemple, via le menu accessible par le bouton jaune dans les panneaux de classes, de méthodes et de code. 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 dans les menus du System Browser accessible par le bouton jaune. Ces menus sont contextuels, autrement dit, chaque panneau a son propre menu. Même si les éléments du menu ont le même nom, leur *signification* dépend du contexte. Par exemple, le panneau de catégories, le panneau de classes, le panneau de protocoles et enfin, celui des messages ont tous `file out` dans leur menu respectif. Cependant, chaque `file out` fait une chose différente : dans le panneau des catégories, il enregistre entièrement dans un fichier la catégorie sélectionnée ; dans le celui des classes, des protocoles ou des messages, 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 probablement la plus utile du menu est `find class... (f)` dans le panneau de catégories. 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 connaît 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 quelle catégorie 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.

Dans le panneau de classes, le menu propose `find method` (pour "trouver une méthode") et `find method wildcard...` qui s'avèrent utiles si souhaitez naviguer dans une méthode particulière. Cependant, à moins que la liste des méthodes soient très longues, il est souvent plus


---

<sup>5</sup>Refactoring en anglais.

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 cherchez. Ceci va faire glisser l'ascenseur du panneau jusqu'à ce que la méthode souhaitée soit visible.

 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.

 Comparez le résultat de `Browse Protocol`, `Browse Hierarchy`, et `Show Hierarchy` dans le menu contextuel du panneau de classes.

## Les autres navigateurs de classes

Au début de cette section nous avons mentionné un autre navigateur : le *Package Pane Browser*. Il peut être ouvert via le menu `World : World ▷ open... ▷ package pane browser`. Il se présente comme le navigateur de classes sauf qu'il connaît les conventions de nommage des catégories-système. Vous aurez noté que les noms des catégories sont décomposés en deux parties. Par exemple, la classe `ScaleMorph` appartient à la catégorie *Morphic-Widgets*. Le *Package Browser* suppose que la partie précédant le trait d'union (soit *Morphic*) est le nom du "paquetage" ou "package". Il ajoute une cinquième colonne permettant de naviguer dans les catégories de chaque paquetage particulier. Cependant, si vous ne choisissez aucun paquetage, toutes les catégories seront disponibles (avec leur nom entier) comme avec un navigateur ordinaire à quatre panneaux.

Malheureusement, le sens donné au terme paquetage a changé depuis le développement du *Package Pane Browser*. "Package" a aujourd'hui une définition plus précise en accord avec l'outil de gestion de paquetages *Monticello*. Nous aborderons cet outil dans la prochaine section. Il n'y a pour l'instant aucun outil pour naviguer dans les paquetages tels qu'arrangés dans *Monticello*, cependant une telle application est en cours de développement.

La communauté Squeak est en train de développer toute une famille de nouveaux navigateurs reposant sur une nouvelle librairie très adaptable nommée *OmniBrowser*. L'implémentation de *OmniBrowser* vaut d'être étudiée au moins comme un bon exemple de *conception* orientée objet. Du reste, la plupart des outils basés sur *OmniBrowser* ressemble beaucoup à ceux que nous avons décrits. L'amélioration principale que vous remarquerez dans l'*Omni System Browser* est l'ajout de *protocoles virtuels* ou *virtual protocols*. Aux protocoles traditionnellement définis par le programmeur s'ajoutent un certain nombre de protocoles virtuels définis pour chaque classe par des systèmes de règles. Par exemple, le protocole *--supersend--* inclut toutes les méthodes qui envoient *super*, alors que le protocole *--required--* liste tous les messages qui sont émis par les méthodes dans la classe courante ou ses super-classes mais qui n'y sont pas définies.

## 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*.



Ouvrez un espace de travail *Workspace* et exécutez le code suivant pour naviguer dans la liste des senders du message *checkExtent*: en utilisant **do it** :

---

```
SystemNavigation default browseAllCallsOn: #checkExtent: .
```

---

Pour restreindre le champ de la recherche à une classe spécifique :

---

```
SystemNavigation default browseAllCallsOn: #drawOn: from: ScaleMorph .
```

---

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 du bouton **implementors** est :

---

```
SystemNavigation default browseAllImplementorsOf: #checkExtent: .
```

---

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.

## Résumé

Comme vous avez pu le voir, il y a plusieurs façon de naviguer dans le code Smalltalk. Vous pouvez être quelque peu désorienté de prime abord, mais vous retomberez toujours sur vos pieds en revenant sur le traditionnel System Browser. N'importe comment, nous constatons habituellement qu'une fois que les débutants ont acquis plus d'expérience avec Squeak, la disponibilité de plusieurs navigateurs différents devient une des plus importantes fonctionnalités proposées, car elle offre beaucoup de manières d'appréhender la compréhension et l'organisation de votre code. Comprendre le code est l'une des plus grandes difficultés du développement logiciel à grande échelle.

## 6.3 Monticello

Nous vous avons donné un aperçu de Monticello, l'outil de gestion de paquetages de Squeak 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 Squeak

Le système du paquetage (ou packaging) est façon simple et légère d'organiser le code source de Smalltalk. Il est influencé par la convention de nommage employée depuis longtemps et dont nous avons parlé plus haut (dans la section 6.2) ; ce système enrichit cette convention de manière conséquente.

Prenons l'exemple suivant en guise d'explication. Supposons que nous sommes en train de développer une librairie pour nous faciliter l'utilisation d'une base de données relationnelles depuis Squeak. Vous avez décidé d'appeler votre librairie ou *framework* SqueakLink et avez créé une série de catégories-système contenant toutes les classes que vous avez écrites, *par ex.*,

La catégorie 'SqueakLink-Connections' contient OracleConnection  
MySQLConnection PostgresConnection

La catégorie 'SqueakLink-Model' contient 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 SQL <sup>6</sup> :

---

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

---

Ces méthodes appartiennent au même paquetage que les classes dans les catégories SqueakLink-Connections et SqueakLink-Model. Mais la classe Object n'appartient clairement pas à notre paquetage ! Ainsi vous avez 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 *\*squeaklink* (remarquez l'astérisque en début de nom et l'utilisation des minuscules). L'association des catégories en *SqueakLink-...* et des protocoles *\*squeaklink* forme un paquetage nommé SqueakLink. Précisément, 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-* ;

---


<sup>6</sup>Nous dirions que ce format est SQL-friendly.



2. toutes les définitions de méthodes dans n'importe quelle classe dont le protocole se nomme *\*foo* ou n'importe quel nom commençant par *\*foo-* (durant la comparaison de ces noms, la casse des lettres est parfaitement ignorée) 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<sup>7</sup> est ignorée dans la règle 2 est que, par convention, les noms de protocole sont tous en minuscule (et peuvent inclure des espaces), alors que les noms de catégorie utilise une écriture en chameau c'est-à-dire les mots composants ces noms sont en capitale et forment les noms sans espaces comme dans CamelCase (nom anglais de cette technique de formatage de nom).

La classe `PackageInfo` implémente ces règles et vous pouvez mieux les appréhender en expérimentant cette classe.

 Essayez ceci dans votre image qui devrait contenir `PackageInfo` et `RefactoringBrowser`.

Le code du `Refactoring Browser` code utilise ces conventions de nommage de paquetages avec `RefactoringEngine` comme nom de paquetage. Dans le `Workspace`, créez un modèle de paquetage avec :

---

```
refactory := PackageInfo named: 'RefactoringEngine'.
```

---


Il est possible maintenant de faire une introspection de ce paquetage. Par exemple, `refactory classes` nous retourne la longue liste des classes qui font le `Refactoring Engine` et le `Refactoring Browser`. L'expression `refactory coreMethods` nous renvoie une liste de `MethodReferences` ou références de méthodes pour toutes les méthodes de ces classes. La requête `refactory extensionMethods` est peut-être une

---

<sup>7</sup>La hauteur minuscule ou majuscule d'une lettre.

des plus intéressantes : elle retourne la liste de toutes les méthodes contenues dans le paquetage `RefactoringEngine` qui ne sont pas dans une classe de `RefactoringEngine`. Cette expression inclut, par exemple, `ClassDescription>chooseThisClassInstVarThenDo:` et `SharedPool class>>keys`.

Les paquetages sont des ajouts à Squeak 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.

 Évaluez (`PackageInfo` named: 'Collections') `externalSubclasses` ; cette expression répond une liste de toutes les sous-classes de `Collection` qui ne sont pas dans le paquetage `Collections`.

Vous pouvez envoyer `fileOut` à une instance de `PackageInfo` pour obtenir un *change set* du paquetage entier. Pour un versionage plus sophistiqué des paquetages, nous utilisons Monticello.

## Monticello basique

Monticello est nommé ainsi d'après la villégiature de Thomas Jefferson, troisième président des États-Unis d'Amérique et auteur de la statue pour le libértés 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 "quet-sche" : Monn-ti-tchel-lo. <sup>8</sup>

Quand vous ouvrez le navigateur Monticello, vous voyez deux panneaux de listes et une ligne de boutons, comme sur la figure 6.12.

La colonne 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.

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

---

<sup>8</sup>Note du traducteur : c'est aussi une commune de Haute-Corse.

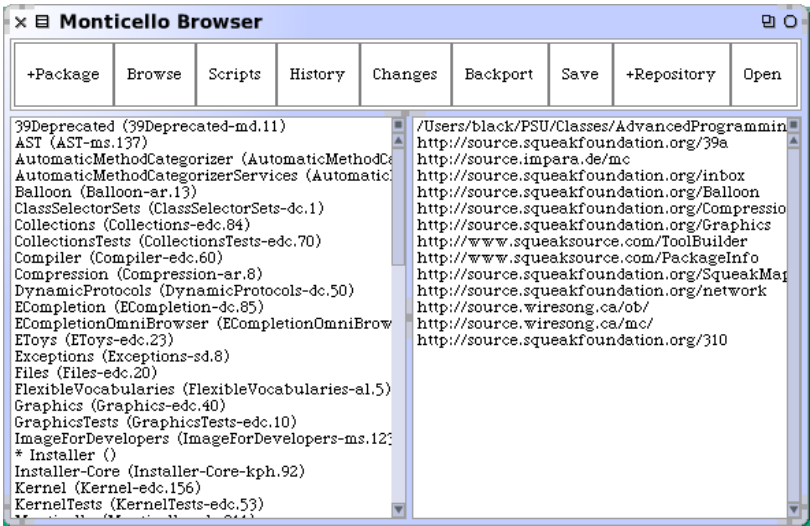


FIG. 6.12 – Le navigateur Monticello.

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 d’où vous avez votre image. Quand vous chargez du code depuis un dépôt distant (ou remote repository) ou quand vous y 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 avez besoin d’accéder à un paquetage. De plus, si vous avez directement reçu un fichier Monticello (.mcz), par exemple, en attache dans un courriel, la façon la plus convenable d’y accéder depuis Squeak 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.

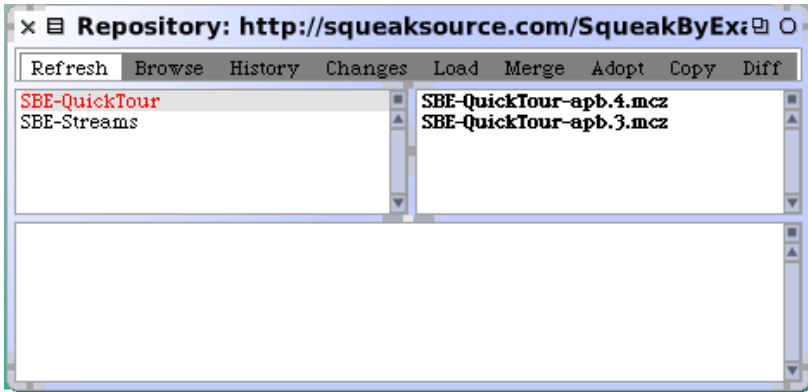



FIG. 6.13 – Un navigateur de dépôt ou Repository Browser.

 Ouvrez Monticello, cliquez sur `+Repository` et choisissez `HTTP`. Éditez la zone de texte à lire :

---

MCHttpRepository

location: 'http://squeaksource.com/SqueakByExample'

user: "

password: "

---

Ensuite cliquez sur `Open` pour ouvrir un navigateur de dépôt ou Repository Browser. Vous devriez voir quelque chose comme la figure 6.13. Sur la gauche, nous voyons une liste de tous les paquetages présents dans le dépôt ; si vous en sélectionnez un, la colonne 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 une copie grâce au bouton `Copy` d'une version d'un paquetage que vous pourriez ensuite écrire dans un autre dépôt.

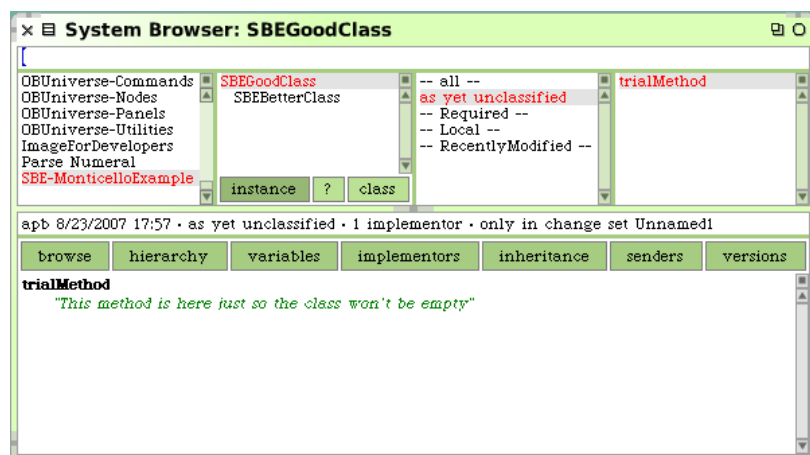



FIG. 6.14 – Deux classes dans le paquetage (ou package) “SBE”.

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 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 catégorie appelée SBE-Monticello, et mettez-y une paire de classes, comme vu sur la figure 6.14. Créez une méthode dans une classe existante, et mettez-la dans le même paquetage que vos classes en utilisant les règles de la page 152 — voir la figure 6.15.

Pour mentionner à Monticello l’existence de votre paquetage, cliquez sur le bouton +Package et tapez le nom du paquetage, dans notre

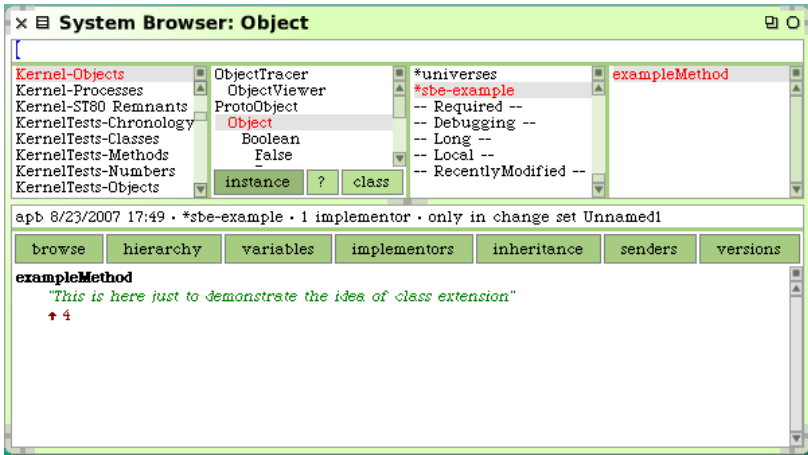


FIG. 6.15 – Une extension de méthode qui sera aussi incluse dans le paquetage (ou package) “SBE”.

cas “SBE”. Monticello ajoutera SBE 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.

Initialement, le seul dépôt associé à ce paquetage sera votre *package cache* comme nous pouvons le voir sur la figure 6.16. 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 6.17 ; quand vous acceptez le message entré, Monticello sauvegardera votre paquetage et l’astérisque décorant le nom du paquetage de la colonne de gauche de Monticello disparaîtra avec le changement le numéro de 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 Browser sur le package

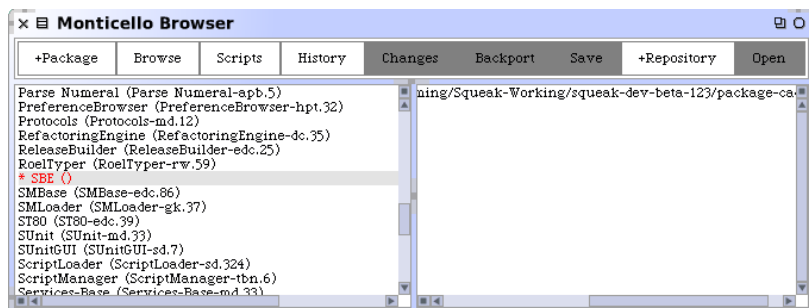


FIG. 6.16 – Le paquetage SBE pas encore sauvegardé dans Monticello.

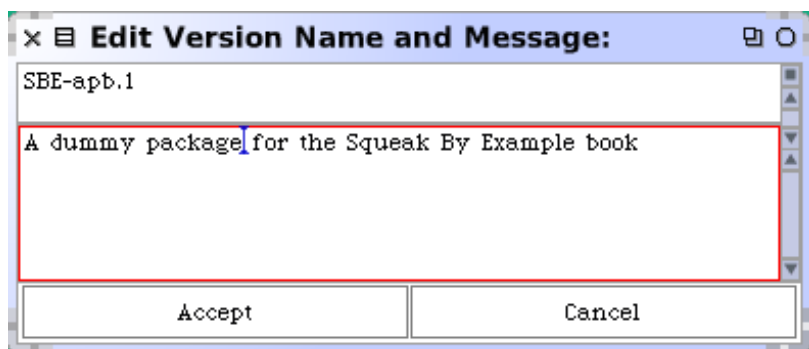


FIG. 6.17 – Fournir un *log message* pour une version d'un paquetage.

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 6.18.

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.

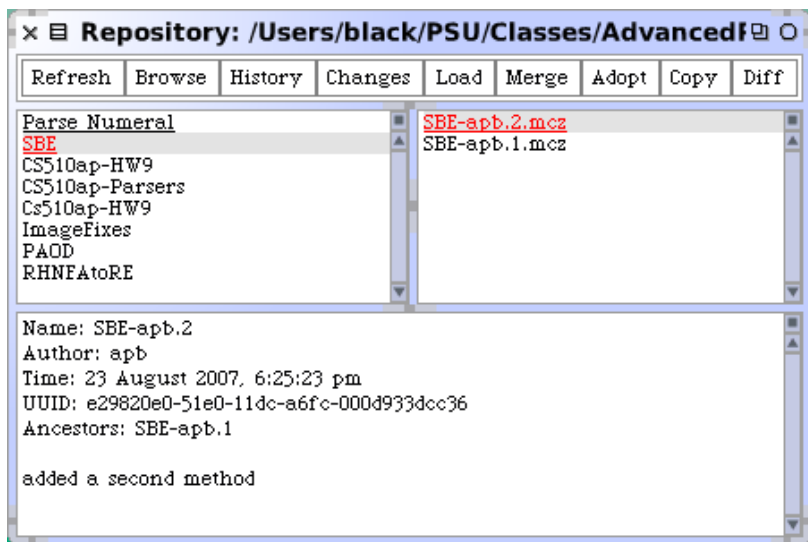


FIG. 6.18 – Deux versions de notre paquetage sont maintenant le dépôt *package cache*.

Alors vous pouvez utiliser le bouton **Copy** dans le Repository Browser de package-cache et choisir le dépôt vers lequel le paquetage devrait être copié. Vous pouvez aussi associer le dépôt désiré avec le paquetage en utilisant **add to package...** dans le menu contextuel du répertoire accédé par le bouton jaune, comme nous pouvons le voir dans la figure 6.19. Une fois que le paquetage est lié à un dépôt, vous pouvez sauvegarder toute nouvelle version en sélectionnant le dépôt et le paquetage dans le Monticello Browser puis en cliquant sur la bouton **Save**. Bien entendu, vous devez avoir une permission d'écrire dans un dépôt. Le dépôt SqueakByExample sur *SqueakSource* est lisible pour tout le monde mais n'est pas inscriptible ; 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



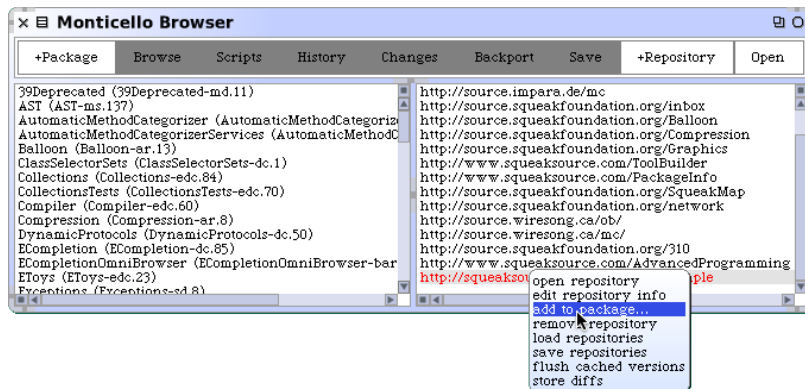


FIG. 6.19 – Ajouter un dépôt à l'ensemble des dépôts liés au packaging.

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 n'importe comment écrite dans le package-cache. Donc vous pourrez corriger en éditant les informations du dépôt (par le menu accessible par le bouton jaune de Monticello Browser) 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 Inspector et l'explorateur Explorer

Une des caractéristiques de Smalltalk qui le rend différent de nombreux environnements de programmation est qu'il vous offre une fenêtre sur un monde d'objets vivants et non pas sur un monde de code statique. Chacun de ces objets peut être examiné par le programmeur et même changé — bien qu'un certain soin doit être apporté

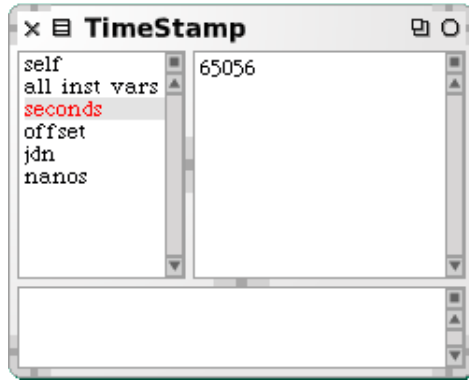



FIG. 6.20 – Inspector TimeStamp now.

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 choisissez `inspect it` via le menu contextuel accédé par le bouton jaune.

(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 6.20 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 *classe* de l'objet en cours d'inspection. Si vous sélectionnez `self` dans la colonne supérieure de gauche, le panneau de droite affi-

chera la description de l'objet en chaîne de caractères ou *printstring* de l'objet. Si vous sélectionnez `all inst vars` dans le panneau de gauche, celui de droite vous présentera une liste de toutes les variables d'instance de l'objet accompagné de leur description *printstring*. Les éléments à suivre dans la liste de cette colonne de gauche représente les variables d'instance ; une par une, elles peuvent ainsi être facilement examinées et même modifiées dans le panneau de droite.

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` est liée à tout objet en cours d'inspection. 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 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.

Vous pouvez changer les variables d'instance directement en les sélectionnant, puis en remplaçant l'ancienne valeur dans la colonne de droite par une expression *Squeak* et en acceptant cette dernière. *Squeak* évaluera l'expression et assignera le résultat à la variable d'instance.

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.

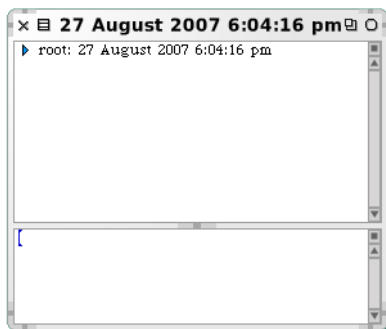


FIG. 6.21 – Explorer TimeStamp now.

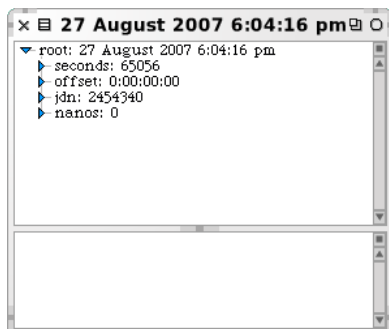



FIG. 6.22 – Explorer les variables d'instance.

## Object Explorer

L'*Object Explorer* ou explorateur d'objet est conceptuellement parlant semblable à l'inspecteur mais présente ses informations de manière différente. Pour voir la différence, nous allons *explorer* le même objet que nous venons juste d'inspecter.



 Sélectionnez **self** dans le panneau gauche de notre inspecteur et choisissez **explore (l)** dans le menu contextuel obtenu via le bouton jaune.

La fenêtre Explorer apparaît alors comme sur la figure 6.21. Si vous cliquez sur le petit triangle à gauche de root (racine, en anglais), la vue changera comme dans la figure 6.22 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 portée <sup>9</sup>.

<sup>9</sup>En anglais, vous entendrez souvent le terme "scope" pour désigner la portabilité

Pour comprendre l'importance de l'explorateur, employons-le pour explorer une structure profonde imbriquant beaucoup d'objets.

 Ouvrez un navigateur et cliquez deux fois avec le bouton droit de la souris ou bouton bleu sur la colonne des méthodes de manière à afficher le halo morphique sur le morph `PluggableListMorph` qui est utilisé pour représenter la liste des messages. Cliquez avec le bouton gauche ou bouton rouge sur l'icône debug  et sélectionnez dans le menu flottant `explore morph`. Ceci ouvrira un Explorateur sur l'objet `PluggableListMorph` qui représente la liste de méthodes du navigateur à l'écran. Ouvrez l'objet root (en cliquant sur son triangle), ouvrez ses sous-morphs submorphs et continuez d'explorer la structure des objets sur lesquels reposent ce morph comme nous pouvons le voir sur la figure 6.23.

## 6.5 Debugger, le débogueur

Le débogueur Debugger est sous conteste l'outil le plus puissant dans la suite d'outils de Squeak. 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 écrire 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

---

Bien sûr, nous sommes certain qu'une méthode si triviale fonctionnera, ainsi plutôt que d'écrire un test *SUnit* (que nous verrons

---

des variables d'instance.

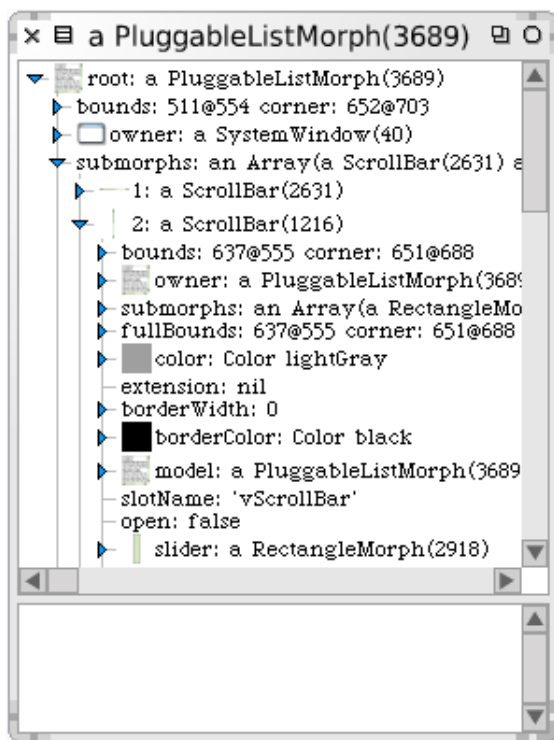


FIG. 6.23 – Explorer une PluggableListMorph.

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 6.24.

Le PreDebugWindow nous indique dans sa barre de titre qu'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»Dolt 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 de-

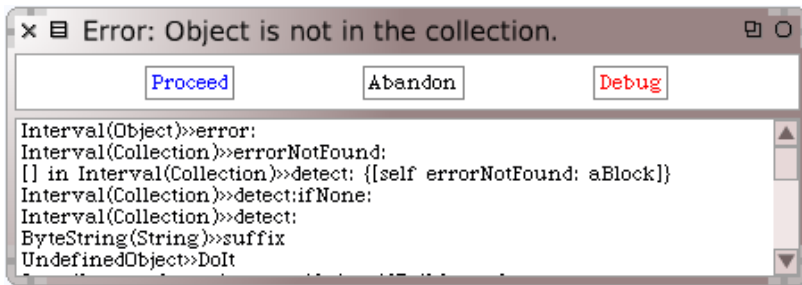



FIG. 6.24 – Un PreDebugWindow nous alarme de la présence d’un bug.

mandé à Squeak de l’imprimer sur cet espace par `print it`. Ce code envoya, bien sûr, le message `suffix` à l’objet `ByteString` (`'readme.txt'`). S’en suit l’exécution de la méthode `suffix` héritée dans 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, cliquez sur le bouton `Debug`.

 Vous pouvez aussi ouvrir *Debugger* en cliquant à l’aide du bouton rouge sur n’importe quelle ligne du *stack trace*. Si vous faites ainsi, le débogueur s’ouvrira sur la méthode correspondante.

Le débogueur est visible sur la figure 6.25; il semble intimidant au début, mais il est assez facile à utiliser. La barre de titre et le panneau supérieur sont très similaire 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 pour poursuivre l’exécution. Ceci comprend tous les objets impliqués dans le calcul, avec leurs variables d’instance et toutes les variables



FIG. 6.25 – Le débogueur.

temporaires des méthodes exécutées.


Dans la figure 6.25 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 et attend une réponse.

Les quatre panneaux inférieurs du débogueur sont véritablement deux mini-inspecteurs (sans panneaux-espace de travail). L'inspecteur sur le pan 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 devons




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-I`.

L'inspecteur de droite affiche les variables temporaires du contexte courant. Dans la figure 6.25, `value` a été envoyé au paramètre `exceptionBlock`.

 Pour voir la valeur actuelle de ce paramètre, cliquez sur `exceptionBlock` dans cet inspecteur de contexte. Cela vous informera que `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 demander `inspect (i)` ou `explore (l)` depuis le menu contextuel au bouton jaune : 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 la pénultième (soit l'avant-avant-dernière) ligne de la méthode et que l'exécution n'aurait jamais du atteindre la dernière ligne. Squeak ne nous permet pas de lancer une exécution en arrière mais il ne permet de relancer une méthode, ce qui marche parfaitement dans notre code qui ne mute pas les objets mais qui en crée de nouveaux.

 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 6.26).

Les boutons `Into` et `Over` offrent deux façons différentes de parcourir l'exécution pas-à-pas. Si vous cliquez sur le bouton `Over`, Squeak exécutera sauf erreur l'envoi de 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 où nous avons démarré et ça ne nous aide

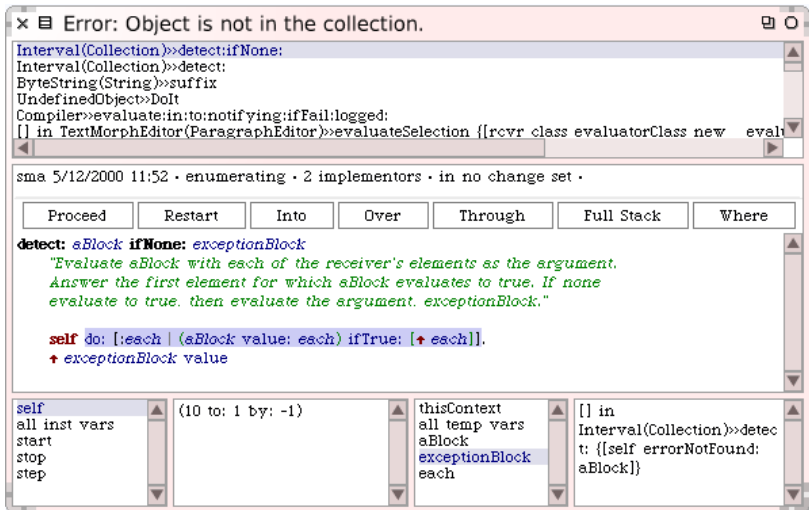





FIG. 6.26 – Debugger après avoir relancé la méthode `detect: ifNone:`.

pas beaucoup. En fait, nous avons besoin de trouver pourquoi `do:` ne trouve pas le caractère que nous cherchons.

 Cliquez sur le bouton `Over` puis cliquez sur le bouton `Restart` pour obtenir la situation vue dans la figure 6.26.

 Cliquez sur le bouton `Into`; Squeak 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 Squeak de faire. `Through` est le bouton approprié à ce cas : nous voulons ignorer les détails de `do:` lui-même et se focaliser sur l'exécution du bloc argument.

 Cliquez sur le bouton `Through` plusieurs fois. Sélectionnez `each` dans le mini-inspecteur de contexte (en bas à droite). Vous remarquez que `each` décompose depuis 10 au fur et à mesure de l'exécution de la méthode `do:`.

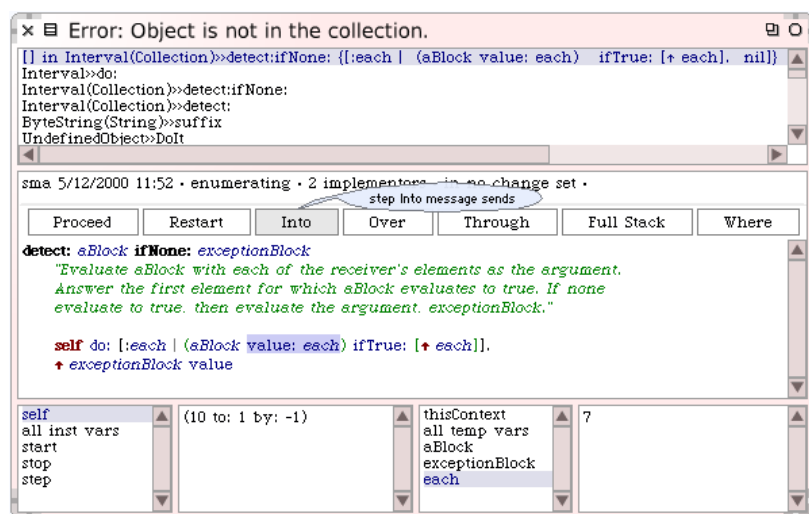


FIG. 6.27 – Debugger après un *pas* dans la méthode `do`: plusieurs fois grâce au bouton `Through`.

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 6.27.

Après avoir cliqué sur le bouton `Into`, nous nous trouvons dans la position illustrée par la figure 6.28. 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 `i` dans le mini-inspecteur contextuel, vous pouvez voir sa valeur actuelle, qui devrait être 7 si vous avez suivi jusqu'ici la procédure. Vous pouvez alors sélectionner l'élément correspondant de `self` dans l'inspecteur de `self`. Dans la figure 6.28, vous pouvez voir que l'élément 7 de la chaîne de caractères est le caractère 46 : ce n'est pas un caractère-point. Si vous sélectionnez `dot` dans l'inspecteur 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



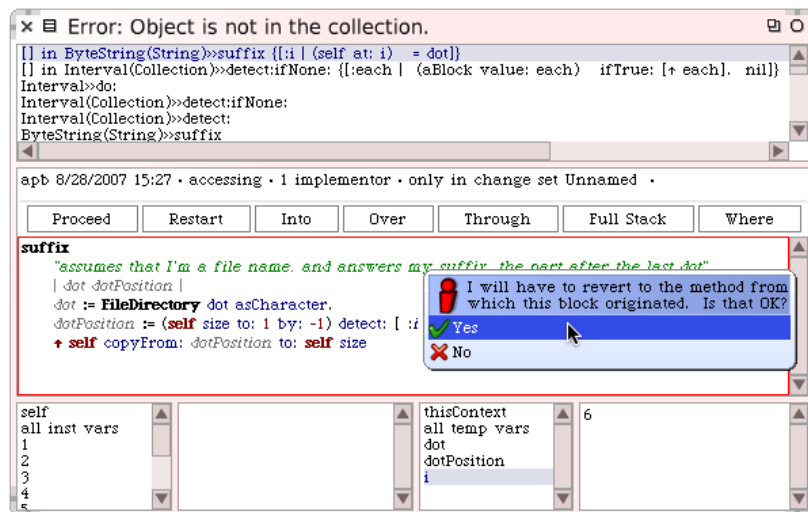



FIG. 6.29 – Changer la méthode suffix dans Debugger : demander la confirmation de la sortie du bloc interne. La boîte d’alerte nous dit : “Je devrai revenir à la méthode d’où ce bloc est originaire. Est-ce bon?”. Elle propose les réponses : *yes* pour oui, *no* pour non.

la nouvelle méthode.

 Cliquez sur le bouton **Restart** et ensuite **Proceed** ; Debugger disparaîtra et 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 précise. La façon d’éviter cette sorte de problème est d’écrire un test SUnit pour définir en fois pour toute 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 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 6.30. 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

---

```
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 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-`.<sup>11</sup> 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 :

---

<sup>11</sup>Sachez que vous pouvez ouvrir un débogueur d'urgence n'importe quand en tapant `CMD-SHIFT`.

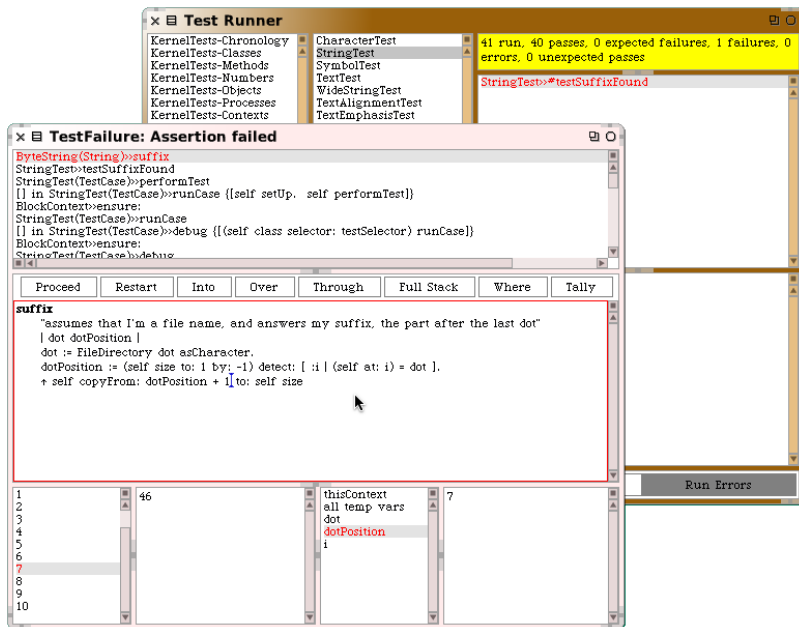


FIG. 6.30 – Changer la méthode suffix dans Debugger : corriger l’erreur du plus-d’un-point après l’assertion fautive SUnit.

#### 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 asCharacter.
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 notificateur 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 de 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:`<sup>12</sup> où le second argument un bloc qui retourne tout simplement une chaîne.

Nous 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*) fonctionnent de façon concurrente dans votre image. Dans l'avenir la machine virtuelle de `Squeak` bénéficiera d'avantage des multi-processeurs lorsqu'ils seront disponibles, mais le partage d'accès est actuellement programmé sur le principe de tranches temporelles (ou *time-slice*).

Le `Process Browser` ou navigateur de processus est un cousin de `Debugger` qui vous permet d'observer les divers processus tournant dans le système `Squeak`. La figure 6.31 nous en présente une capture d'écran. Le panneau supérieur gauche liste tous les processus présents dans `Squeak`, dans l'ordre de leur priorité depuis le *timer interrupt watcher* (système de surveillance d'interruption d'horloge) de priorité 80

---

<sup>12</sup>En anglais, *if none* signifie "s'il n'y a rien".



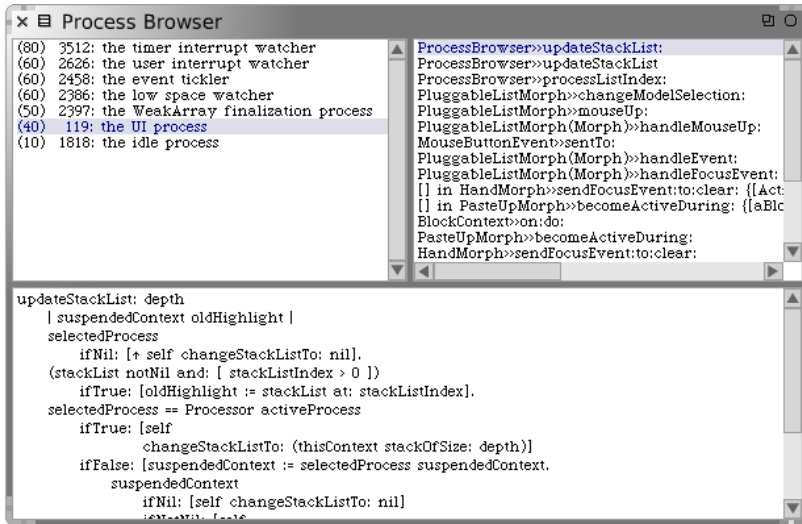


FIG. 6.31 – Le Process Browser.

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*<sup>13</sup> *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 utilisant le menu contextuel par le bouton jaune de la souris voire, être basculé dans un mode de rafraîchissement automatique via l'option **turn on auto-update (a)** du même menu.

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 des options du menu contextuel sur des tranches de la pile permettent des fonctions équivalentes.

<sup>13</sup>UI désigne *User Interface* ; en français, interface utilisateur.



FIG. 6.32 – Le Method Names Browser montrant toutes les méthodes contenant le sous-élément de chaîne random dans leur sélecteur.

## 6.7 Trouver les méthodes

Il y a deux outils dans Squeak pour vous aider à trouver des messages ; chacun pouvant être trouvé dans l'onglet *Tools*. Ils diffèrent en terme d'interface et de fonctionnalité.

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. Ceci peut vite devenir accablant.

La fonctionnalité de recherche du Message Names Browser ou navigateur de *noms de message* 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 6.32. Cependant, c'est un navigateur complet : si vous sélectionnez un des noms dans la colonne de gauche, toutes les méthodes ayant ce nom seront listées dans celle de droite et vous pourrez alors naviguer dans le panneau inférieur. Le Message Names Browser a une barre de bouton, comme le navigateur classique, pouvant être utilisée pour ouvrir d'autres navigateurs sur la méthode choisie ou sur sa classe.

## 6.8 Change set et son gestionnaire Change Sorter

À chaque fois que vous travaillez dans Squeak, 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 inclus ; ainsi si, par exemple, vous créez une nouvelle variable globale par affectation dans un espace de travail, la création de variable ne sera pas dans un *change set*.

À n'importe quel 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 Set Browser disponible dans le menu principal dans **World > open... > simple change sorter** ou en glissant l'icône **Change Set** de l'onglet *Tools*.

la figure 6.33 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 la colonne supérieure 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*. La colonne supérieure de droite

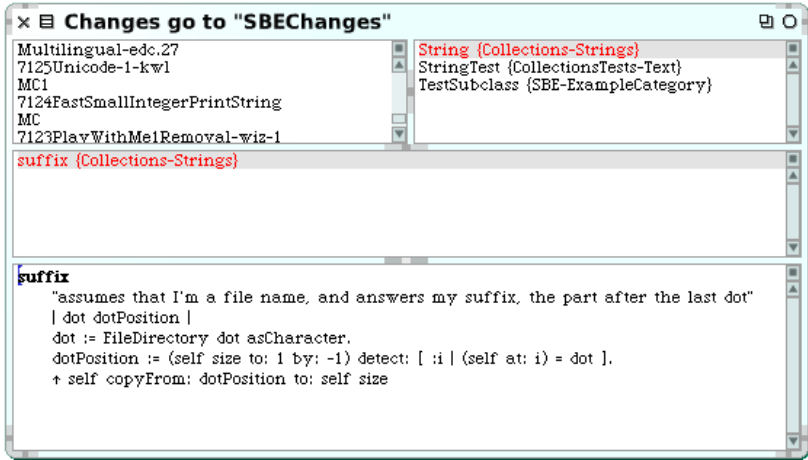


FIG. 6.33 – Le Change Set Browser.

liste toutes les classes (accompagnées 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 Set Browser vous permet d'effacer des classes et des méthodes du *change set* en utilisant le menu du bouton jaune sur les éléments correspondants. Cependant, pour une édition plus élaborée, vous devez utiliser un deuxième programme, le *Change Sorter* (ou trieuse de *change set*), disponible sous ce nom dans l'onglet *Tools* ou en passant par *World > open... > dual change sorter*. Nous pouvons le voir dans la figure 6.34.

Le Change Sorter est essentiellement un double navigateur de *change set* côte-à-côte ; chaque côté affiche un *change set*, une classe ou une méthode différente. Cette implémentation supporte les principales

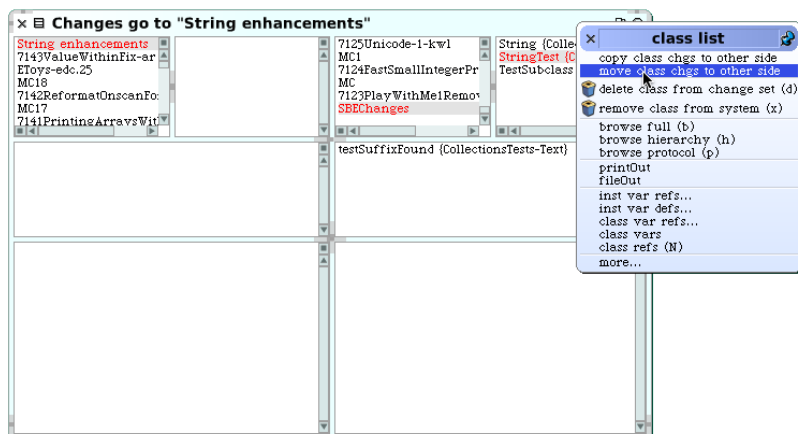


FIG. 6.34 – Le Change Sorter.

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 dans le menu contextuel accessible via le bouton jaune de la souris dans la figure 6.34. 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 Squeak vers le système de fichiers d'où il peut être importé dans une autre image Squeak ou vers un autre Smalltalk que Squeak. L'exportation de *change set* est connu sous le nom "filing-out" et peut être réalisé en utilisant le menu contextuel (bouton jaune) 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 versionage 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 Squeakeurs. Ils ont l'avan-

tage d'être simple et relativement portable (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 paquets 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 inexistante 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 silenceusement un changement fait par un autre.

À l'inverse, les paquetages (dits aussi packages) de Monticello représente 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* reste 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. Cet étape requiert l'usage d'un autre outil, le File List Browser.

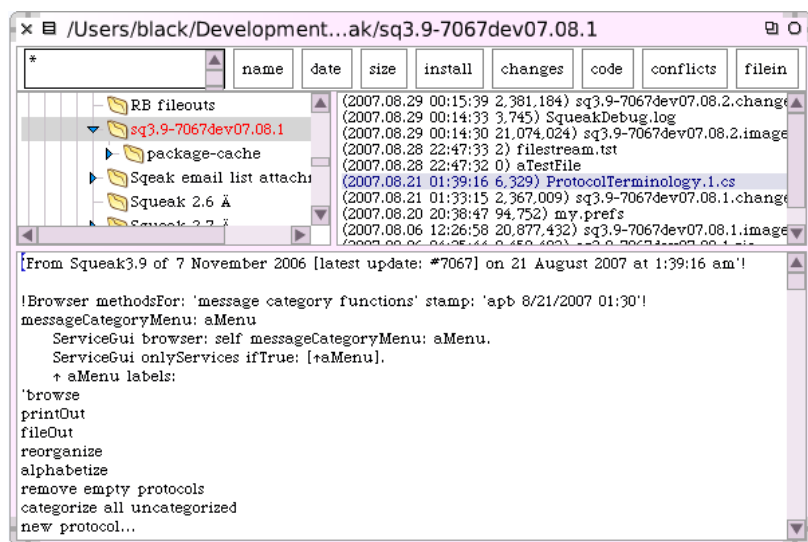


FIG. 6.35 – Le File List Browser.

## 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 Squeak. Vous pouvez l'ouvrir depuis le menu **World**▷**open...**▷**file list** ou en le glissant depuis l'onglet **Tools**. 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 6.35.

Quand vous ouvrez un navigateur de fichiers, il pointera tout d'abord le répertoire actuel, *c-à-d.* celui depuis lequel vous avez démarré Squeak. 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. 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 6.35, le nom des fichiers ont 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 saut de ligne 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. N'importe comment, le jeu complet des options est toujours disponible grâce à l'option `more...` du menu contextuel accessible via le bouton jaune, 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 6.36. Le File Contents Browser est proche d'un navigateur classique 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 *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 6.36. Les options du menu contextuel de chacun des quatre panneaux supérieurs vous permettra de charger (en anglais,



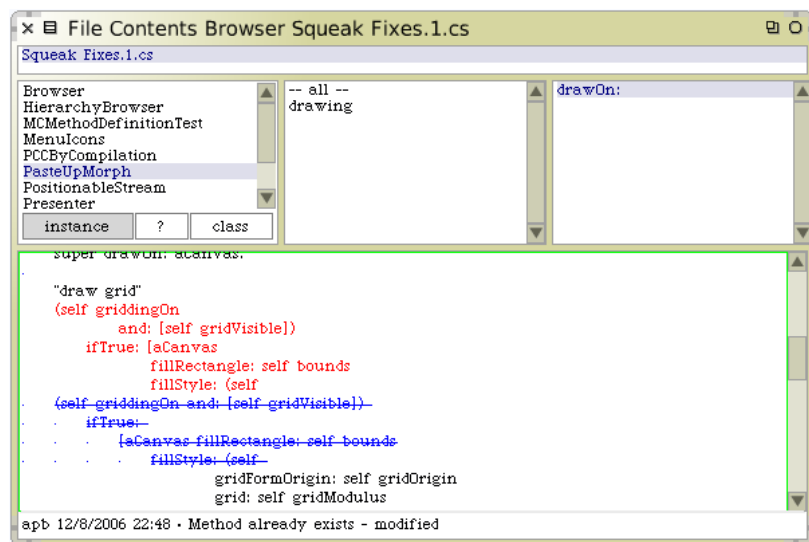



FIG. 6.36 – Le File Contents Browser.

*file in*) le *change set* complet, la classe, le protocole ou la méthode correspondante.

## 6.10 En Smalltalk, pas de perte de codes

Squeak peut parfois planter : en tant que système expérimental, Squeak vous permet de changer n'importe quoi dont les éléments vitaux qui font que Squeak fonctionne !

 Pour crasher malicieusement Squeak, 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 sauvegardé 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 sur comment rappeler 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. Squeak vous offre de meilleurs façons de vous en sortir.

## La pêche au code

Redémarrez Squeak depuis la sauvegarde (ou *snapshot*) la plus récente et sélectionnez `World>help>useful expressions`. Ceci vous ouvrira un Workspace plein d'expressions utiles. Les trois premières,

---

```
Smalltalk recover: 10000.  
ChangeList browseRecentLog.  
ChangeList browseRecent: 2000.
```

---

sont les plus utiles pour le recouvrement de données.

Si vous exécutez `ChangeList browseRecentLog`, vous aurez 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 tantonnant 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 Squeak durant tout ce temps. Vous pouvez effacer des articles de cette liste en utilisant le menu accessible par le bouton jaune de la souris. 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 dolts`. 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 *dolts* créateur de classes, ensuite utilisez `remove dolts` (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 Squeak sans sauvegarder l'image, redémarrer et m'assurer que mon nouveau fichier se charge parfaitement.

## 6.11 Résumé du chapitre

Pour développer efficacement avec Squeak, il est important d'investir quelques efforts dans l'apprentissage des outils disponibles dans l'environnement.

- Le *System Browser* standard est votre principale interface pour naviguer dans les catégories de classes, les classes, les protocoles et les méthodes existants et pour en définir de nouveaux. Ce navigateur de classe offre plusieurs boutons pour accéder directement aux *senders* ou aux *implementors* de message, aux versions d'une méthode, etc.
- Plusieurs navigateurs de classes différents existent (tel que *Omni-Browser* 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 surliquant 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 versionage (organi-

sation et maintien de versions) et de partage de *paquetages* de classes et de méthodes nommés aussi *packages*. Un paquetage Monticello comprend une catégorie-système, 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 via le bouton bleu (bouton de droite) de la souris 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 lorsque qu'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 savez 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 fichiers. Il vous permet aussi d'insérer du code source depuis le système de fichiers via `fileIn`.
- Dans le cas où votre image plante <sup>14</sup> avant que vous l'ayez sau-

<sup>14</sup>Nous parlons de *crash*, en anglais.

vegardée 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 rejouer et les charger dans la copie la plus récente de votre image.



## Chapitre 7

# SUnit

### 7.1 Introduction

SUnit est un environnement simple mais pourtant puissant pour la création et le 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 Squeak 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](http://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 débusser 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 achevée. Quelques-fois, 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 pourquoi nous testons et de quoi est fait 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 le test 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 a dit, à un moment ou à un autre : “j'écrirais des tests si j'avais plus de temps”. Si vous n'écrivez jamais de bug, 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 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 évolue.

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 documentation est à jour. Deuxièmement, les tests aident les développeurs à garantir que certaines modifications qu'ils viennent juste d'apporter à un package n'a 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 oeuvre. 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-le.

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, assoupis, 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 test 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-le. Nous avons trouvé que d'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 «*test 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

 Premièrement vous devriez créer une nouvelle sous-classe de `TestCase` nommée `ExampleSetTest`. Ajoutez 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'
  classInstanceVariableNames: ''
  category: 'MyTest'
```

---


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 nommez la classe de test correspondante `PatternTest`, les deux classes seront présentées ensembles, 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.

## 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: 3)
```

---

Le second test nommé `testOccurrences` vérifie que le nombre d'occurrences de 5 dans le Set `full` est égal à un, même si on ajoute 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

Le plus facile pour exécuter les tests est d'utiliser *l'évaluateur de tests* SUnit (*TestRunner*), que vous pouvez ouvrir depuis le menu `World > open...` ou en glissant le `TestRunner` depuis la barre d'outils (l'onglet *Tools*). L'évaluateur de tests, montré dans la figure 7.1, est conçu pour faciliter l'exécution de groupes de tests. Le panneau le plus à gauche présente toutes les catégories système 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`.

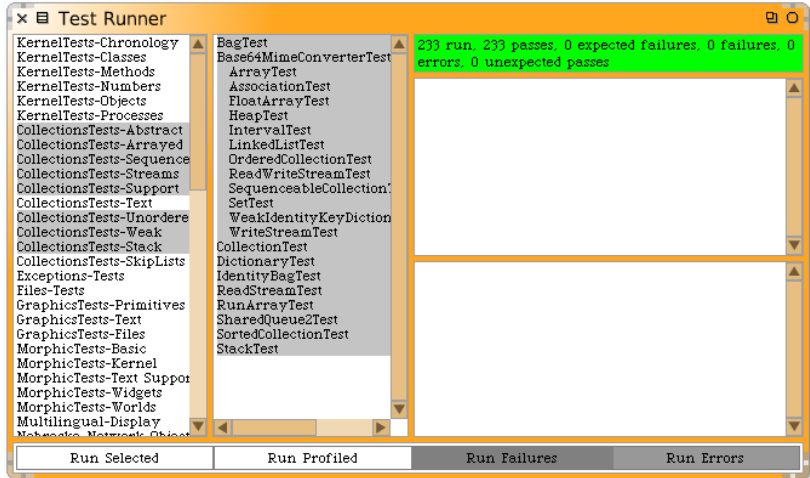



FIG. 7.1 – SUnit, l'évaluateur de test de Squeak.

 Ouvrez un évaluateur de test, sélectionnez la catégorie `MyTest` et cliquez le bouton `Run Selected`.

Vous pouvez aussi exécuter votre test en évaluant un `print it` sur le code suivant : (ExampleSetTest selector: `#testRemove`) run. L'expression suivante est équivalente mais plus concise : ExampleSetTest run: `#testRemove`. Il nous est habituel d'inclure un commentaire exécutable à notre méthode de test ce qui nous permet de les exécuter avec un `do it` depuis le System Browser, comme il est montré dans méthode 7.6.


---

### Méthode 7.6 – Les commentaires exécutables dans les méthodes de test

---

```
ExampleSetTest>>testRemove
"self run: #testRemove"
full remove: 5.
self assert: (full includes: 6).
self deny: (full includes: 5)
```

---

 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. Une alternative est d'évaluer l'expression suivante :

---

```
(ExampleSetTest selector: #testRemove) debug
```

---

ou bien

---

```
ExampleSetTest debug: #testRemove
```


---

## 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 argument est à vrai (`true`), le test est réussi ; quand cet argument 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'évaluateur de tests (*TestRunner*), quand tous les tests réussissent, la barre du haut devient verte. Pourtant, 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. Pourtant, il est aussi possible qu'une erreur intervienne pendant l'exécution du test, telle qu'une erreur *message non compris* ou une erreur *d'index hors limites*. Si une erreur survient, les assertions de la méthode de test peuvent ne pas avoir été exécutées du

tout, ainsi on ne peut pas dire que le test a échoué. Toutefois, quelque chose est clairement faux ! Dans l'évaluateur 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 environnement de test comme JUnit<sup>1</sup>, ceci vous sera familier puisque tous ces environnements sont issus de SUnit. Normalement, vous utiliserez l'IHM<sup>2</sup> 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 7.7 illustre l'utilisation de `should:raise:`.

 *Essayez d'évaluer ce test.*

---

<sup>1</sup> [junit.org](http://junit.org)

<sup>2</sup> Interface Homme Machine.



Notez que le premier argument des méthodes `should:` et `shouldnt:` est un *bloc* qui contient l'expression à évaluer.

---

#### Méthode 7.7 – 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 retirés 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 quelque soit le dialecte de Smalltalk, vous pouvez écrire la méthode 7.7 ainsi :

---

#### Méthode 7.8 – 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'évaluateur de tests (`TestRunner`). Si vous ne voulez pas lancer l'évaluateur de tests depuis le menu `open...` ou depuis l'onglet *Tools*, vous pouvez évaluer `TestRunner open` à l'aide d'un `print it`.

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. L’environnement est déjà là : utilisez-le.

## 7.6 L’environnement SUnit

Comme montré dans la figure 7.2, 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.

### 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<sup>3</sup>.

---

<sup>3</sup>En français, démolir.

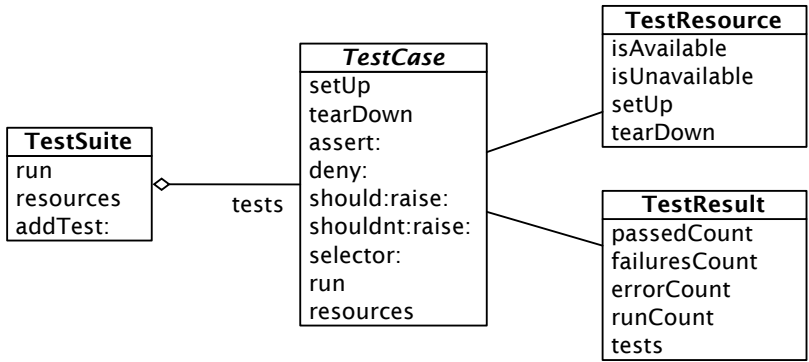


FIG. 7.2 – Les quatres classes constituant le coeur de SUnit.

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 test. 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 patron<sup>4</sup>.

---

<sup>4</sup>?, .

## 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 levées.

## 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 on sait 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, il est sensé d'initialiser et d'ouvrir une connexion vers la base de données ou de compiler du code source avant l'évaluation 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 ces 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 7.9 – 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

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'évaluateur 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 flot de données Stream tel que le Transcript ou un flot 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 flot 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 de 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 7.3).

#### Méthode 7.10 – Exécuter un cas de test

---

```
TestCase»run
  | result |
  result := TestResult new.
  self run: result.
  ↑result
```

---

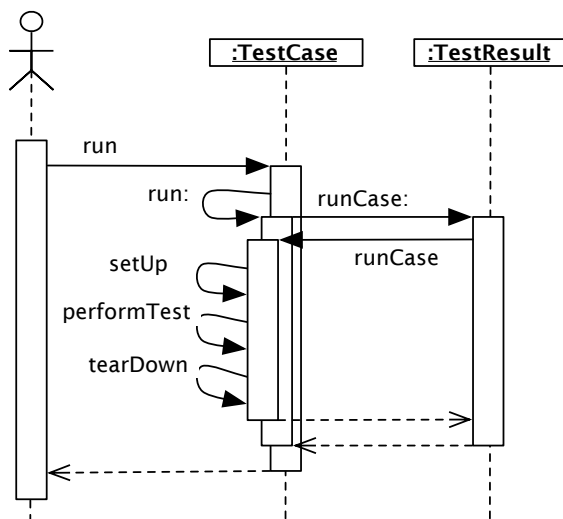


FIG. 7.3 – Exécuter un test.

La méthode `TestCase»run:` envoie le message `runCase:` au résultat de test de classe `TestResult` :

#### Méthode 7.11 – Passage du case 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 7.12 – 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 montré ci-dessous.

---

#### Méthode 7.13 – *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 adéquates. `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 7.14 – *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 7.15 – Exécuter une suite de tests

---

```
TestSuite»run
| result |
result := TestResult new.
self areAllResourcesAvailable
  ifFalse: [↑TestResult signalErrorWith:
    'Resource could not be initialized'].
[self run: result] ensure: [self resources do:
  [:each | each reset]].
↑result
```

---



---

#### Méthode 7.16 – Passage de la suite de tests au TestResult

---

```
TestSuite»run: aResult
self tests do:
  [:each |
    self sunitChanged: 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 7.17 – Disponibilité de la ressource de test

---

```
TestResource class»isAvailable
↑self current notNil
```

---



---

#### Méthode 7.18 – Création de la ressource de test

---

```
TestResource class»current
current isNil ifTrue: [current := self new].
↑current
```

---

## Méthode 7.19 – Initialisation de la ressource de test

---

```
TestResource»initialize  
    self setUp
```

---

## 7.9 Quelques conseils sur le test

Bien que les mécanismes du test soient simples, il n'est pas aussi facile d'écrire de bons tests. Voici quelques conseils pour leur conception.

**Les règles de Feathers.** Michael Feathers, un auteur et consultant en processus agile écrit <sup>5</sup> :

*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 fonctionnalité et, comme tels, permettent de

---

<sup>5</sup>Voir <http://www.artima.com/weblogs/viewpost.jsp?thread=126923>. 9 Septembre 2005

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 récurrentes 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 tout 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 test 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 nuisible : ils rendent la compréhension de la classe plus difficile à déduire à la lecture des tests et un bug dans le code est susceptible de casser beaucoup de tests en même temps. 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 de l'environnement 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'évaluateur de tests, SUnit (dans la barre d'outils).



## Chapitre 8

# Les classes de base

Une grande partie de l'intérêt de Smalltalk ne réside pas dans son langage seul 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 paquet 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 Squeak, 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 Squeak, 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.

Object se trouve dans la catégorie *Kernel-Objects*. Étonnamment, on y trouve 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 ce qu'elles font ou non. Notez que certaines de ces méthodes pourraient être supprimées et que dans les nouvelles versions de Squeak certaines méthodes superflues devraient l'être.

Le commentaire de la classe Object dit :

*Object est la classe racine de la plupart des autres classes dans la hiérarchie de 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, qui ont des implémentations particulières (SmallInteger et UndefinedObject par exemple) ou à certaines classes standard que la VM connaît et dont elle dépend de la structure et du fonctionnement.*

Si nous naviguons dans les catégories de méthodes d'instance de Object, nous commençons à voir quelques uns des comportements importants qu'elle apporte.

## 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:` is a hook qui peut être spécialisé.

Object»`printOn:` est une des méthodes que vous surchargerez le plus



souvent. Cette méthode prend comme argument un flot de données ou Stream dans lequel une représentation en chaîne de caractères ou 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) écrite :

Par exemple, la classe Browser ne redéfinit pas le méthode printOn: et envoyer le message printString à une instance exécute la méthode définie dans Object.

---

```
Browser new printString  →  'a Browser'
```

---

La classe TTCFont montre un exemple de spécialisation de printOn:. Elle imprime le nom de la classe suivie par le nom de la famille, la taille et le nom de la sous-famille de la police, comme le montre le code ci-dessous qui imprime une instance de cette classe.

---

Méthode 8.1 – *printOn : redefinition.*

---

```
TTCFont»printOn: aStream
  aStream nextPutAll: 'TTCFont(';
  nextPutAll: self familyName; space;
  print: self pointSize; space;
  nextPutAll: self subfamilyName;
  nextPut: $)
```

---

```
TTCFont allInstances anyOne printString  →  'TTCFont(BitstreamVeraSans
6 Bold)'
```

---

Notez que le message printOn: n'est pas le même que storeOn:. Le message storeOn: puts on its argument stream an expression that can be used to recreate the receiver. Cette expression est évaluée quand le flot de données 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 comme expression auto-évaluée (self-evaluating?).

**Un mot à propos des représentations et des représentations auto-évaluées.** 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 valeur. Par exemple, la valeur de l'objet `true` est lui-même, *c-à-d.* l'objet `true`. Nous appelons de tels objets *objets auto-évalués*. Vous pouvez voir une version *imprimée* de la valeur d'un objet quand vous imprimez l'objet dans un espace de travail (workspace). Voici quelques exemples d'expressions auto-évaluées.

---

<code>true</code>	→	<code>true</code>
<code>3@4</code>	→	<code>3@4</code>
<code>\$a</code>	→	<code>\$a</code>
<code> #(1 2 3)</code>	→	<code> #(1 2 3)</code>

---

Notez que certains objets comme les tableaux peuvent être 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. Dans Squeak 3.9, un mécanisme a été introduit (via le message `isSelfEvaluating`) pour imprimer des collections dans leur forme auto-évaluée autant que possible et ceci est particulièrement vrai pour les *brace arrays*. L'exemple suivant montre qu'un tableau dynamique est auto-évalué seulement si ses éléments sont :

---

<code>{10@10 . 100@100}</code>	→	<code>{10@10 . 100@100}</code>
<code>{Browser new . 100@100}</code>	→	<code>an Array(a Browser 100@100)</code>

---

Rappelez-vous que les tableaux littéraux ne peuvent contenir que des littéraux. Ainsi le tableau suivant ne contient pas deux points mais six éléments littéraux.

---

<code> #(10@10 100@100)</code>	→	<code> #(10 #@ 10 100 #@ 100)</code>
--------------------------------	---	--------------------------------------

---

Beaucoup de spécialisations de méthodes `printOn:` implémentent le comportement d'auto-évaluation. Les implémentations de `Point>>printOn:` et `Interval>>printOn:` sont auto-évaluées.

### Méthode 8.2 – *Self-evaluation of Point*

---

`Point>>printOn: aStream`

*"The receiver prints on aStream in terms of infix notation."*

`x printOn: aStream.`

`aStream nextPut: $@.`

---

```
y printOn: aStream
```

---



---

### Méthode 8.3 – *Self-evaluation of 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)  "intervals are self-evaluating"
```

---

## 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 d'objets teste l'identité d'objets :

---

### Méthode 8.4 –

---

```
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  "meme valeur"
```

```
(1 + 2 i) == (1 + 2 i) →  false "mais objets differents"
```

---

Ceci fonctionne parce que Complex surcharge = comme suit :

Méthode 8.5 – *Égalité de 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 d'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 instances 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 `hash` :

Méthode 8.6 – *Hash doit être réimplémentée pour les nombres complexes*


---

```
Complex»hash
  "Hash is reimplemented because = is implemented."
  ↑ real hash bitXor: imaginary hash.
```

---

Alors que vous devriez surcharger `=` et `hash` ensemble, vous ne devriez *jamais* surcharger `==`. (La sémantique de l'indentité d'objets est la même pour toutes les classes) `==` est une méthode primitive de `ProtoObject`.

Notez que Squeak a certains comportements étranges comparé à d'autres Smalltalks : par exemple un symbole et une chaîne de caractères peuvent être égaux si the string associated with the symbol is equal to the string. (Nous considérons ce comportement comme un bug, pas comme un feature.)

---

```
#'lulu' = 'lulu'  → true
'lulu' = #'lulu' → true
```

---

## Appartenance à une classe

Plusieurs méthodes vous permettent de demander 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 d'une classe spécifique :

---

```
1 isMemberOf: SmallInteger → true  "doit etre precisement cette classe"
1 isMemberOf: Integer      → false
1 isMemberOf: Number       → false
1 isMemberOf: Object       → false
```

---

Puisque Smalltalk est écrit en Smalltalk, you can really navigate through its structure using the right combination of superclass and class messages (see le chapitre 11).

**isKindOf:** `Object>isKindOf:` answers whether the receiver's class is either the same as, or a subclass of the argument class.

---

```
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` which is a `Fraction` is a kind of `Number`, since the class `Number` is a superclass of the class `Fraction`, but `1/3` is not a `Integer`.

**respondsTo:** `Object>respondsTo:` answers whether the receiver understands the message selector given as an argument.

---

```
1 respondsTo: #, → false
```

---

Normally it is a bad idea to query an object for its class, or to ask it which messages it understands. Instead of making decisions based

on the class of object, you should simply send a message to the object and let it decide (*c-à-d.* on the basis of its class) how it should behave.

## Copying

Copying objects introduces some subtle issues. Since instance variables are accessed by reference, a *shallow copy* of an object would share its references to instance variables with the original object :

---

```

a1 := { { 'harry' } }.
a1  →  #(#('harry'))
a2 := a1 shallowCopy.
a2  →  #(#('harry'))
(a1 at: 1) at: 1 put: 'sally'.
a1  →  #(#('sally'))
a2  →  #(#('sally'))  "the subarray is shared"

```

---

`Object>shallowCopy` is a primitive method that creates a shallow copy of an object. Since `a2` is only a shallow copy of `a1`, the two arrays share a reference to the nested Array that they contain.

`Object>shallowCopy` is the “public interface” to `Object>copy` and should be overridden if instances are unique. This is the case, for example, with the classes `Boolean`, `Character`, `SmallInteger`, `Symbol` and `UndefinedObject`.

`Object>copyTwoLevel` does the obvious thing when a simple shallow copy does not suffice :

---

```

a1 := { { { 'harry' } } }.
a2 := a1 copyTwoLevel.
(a1 at: 1) at: 1 put: 'sally'.
a1  →  #(#('sally'))
a2  →  #(#('harry'))  "fully independent state"

```

---

`Object>deepCopy` makes an arbitrarily deep copy of an object.

---

```

a1 := { { { { 'harry' } } } }.
a2 := a1 deepCopy.
(a1 at: 1) at: 1 put: 'sally'.
a1  →  #(#('sally'))

```

---

```
a2  →  #(#(#('harry')))
```

---

The problem with `deepCopy` is that it will not terminate when applied to a mutually recursive structure :

---

```
a1 := { 'harry' }.
a2 := { a1 }.
a1 at: 1 put: a2.
a1 deepCopy  →  ... does not terminate !
```

---

Although it is possible to override `deepCopy` to do the right thing, `Object»copy` offers a better solution :

---

#### Méthode 8.7 – Copying objects as a template method

---

`Object»copy`

*"Answer another instance just like the receiver. Subclasses typically override `postCopy`; they typically do not override `shallowCopy`."*

↑ `self shallowCopy postCopy`

---

You should override `postCopy` to copy any instance variables that should not be shared. `postCopy` should always do a super `postCopy`.

## Debugging

The most important method here is `halt`. In order to set a breakpoint in a method, simply insert the message `send self halt` at some point in the body of the method. When this message is sent, execution will be interrupted and a debugger will open to this point in your program. (See le chapitre 6 for more details about the debugger.)

The next most important message is `assert:`, which takes a block as its argument. If the block returns `true`, execution continues. Otherwise an exception will be raised. If this exception is not otherwise caught, the debugger will open to this point in the execution. `assert:` is especially useful to support *design by contract*. The most typical usage is to check non-trivial pre-conditions to public methods of objects. `Stack»pop` could easily have been implemented as follows :

Méthode 8.8 – *Checking a pre-condition*


---

```
Stack»pop
  "Return the first element and remove it from the stack."
  self assert: [ self isEmpty not ].
  ↑self linkedList removeFirst element
```

---

Do not confuse `Object»assert:` with `TestCase»assert:`, which occurs in the SUnit testing framework (see le chapitre 7). While the former expects a block as its argument<sup>1</sup>, the latter expects a Boolean. Although both are useful for debugging, they each serve a very different intent.

## Error handling

This protocol contains several methods useful for signaling run-time errors.

Sending `self deprecated: anExplanationString` signals that the current method should no longer be used, if deprecation has been turned on in the *debug* protocol of the preference browser. The String argument should offer an alternative.

---

```
1 dolfNotNil: [ :arg | arg printString, ' is not nil' ]
   → SmallInteger(Object)»dolfNotNil : has been deprecated. use
   ifNotNilDo :
```

---

`doesNotUnderstand:` is sent whenever message lookup fails. The default implementation, *c-à-d.* `Object»doesNotUnderstand:` will trigger the debugger at this point. It may be useful to override `doesNotUnderstand:` to provide some other behaviour.

`Object»error` and `Object»error:` are generic methods that can be used to raise exceptions. (Generally it is better to raise your own custom exceptions, so you can distinguish errors arising from your code from those coming from kernel classes.)

Abstract methods in Smalltalk are implemented by convention with the body `self subclassResponsibility`. Should an abstract class be instantiated by accident, then calls to abstract methods will result in `Object»subclassResponsibility` being evaluated.

---

<sup>1</sup> Actually, it will take any argument that understands value, including a Boolean.



---

### Méthode 8.9 – Signaling that a method is abstract

---

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

---

Magnitude, Number and Boolean are classical examples of abstract classes that we shall see shortly in this chapter.

---

Number new + 1     $\longrightarrow$     Error : My subclass should have overridden #+

---

self shouldNotImplement is sent by convention to signal that an inherited method is not appropriate for this subclass. This is generally a sign that something is not quite right with the design of the class hierarchy. Due to the limitations of single inheritance, however, sometimes it is very hard to avoid such workarounds.

A typical example is Collection»remove: which is inherited by Dictionary but flagged as not implemented. (A Dictionary provides removeKey: instead.)

## Testing

The *testing* methods have nothing to do with SUnit testing! A testing method is one that lets you ask a question about the state of the receiver and returns a Boolean.

Numerous testing methods are provided by Object. We have already seen isComplex. Others include isArray, isBoolean, isBlock, isCollection and so on. Generally such methods are to be avoided since querying an object for its class is a form of violation of encapsulation. Instead of testing an object for its class, one should simply send a request and let the object decide how to handle it.

Nevertheless some of these testing methods are undeniably useful. The most useful are probably ProtoObject»isNil and Object»notNil (though the Null Object<sup>2</sup> design pattern can obviate the need for even these methods).

---

<sup>2</sup>?, .

## Initialize release

A final key method that occurs not in `Object` but in `ProtoObject` is `initialize`.

---

Méthode 8.10 – `initialize` *as an empty hook method*

---

`ProtoObject`»`initialize`

*"Subclasses should redefine this method to perform initializations on instance creation"*

---

The reason this is important is that in Squeak as of version 3.9, the default `new` method defined for every class in the system will send `initialize` to newly created instances.

---

Méthode 8.11 – `new` *as a class-side template method*

---

`Behavior`»`new`

*"Answer a new initialized instance of the receiver (which is a class) with no indexable variables. Fail if the class is indexable."*

↑ `self basicNew initialize`

---

This means that simply by overriding the `initialize` hook method, new instances of your class will automatically be initialized. The `initialize` method should normally perform a `super initialize` to establish the class invariant for any inherited instance variables. Note that this is *not* standard behavior in other Smalltalks.

## 8.2 Numbers

Remarkably, numbers in Smalltalk are not primitive data values but true objects. Of course numbers are implemented efficiently in the virtual machine, but the `Number` hierarchy is as perfectly accessible and extensible as any other portion of the Smalltalk class hierarchy.

Numbers are found in the *Kernel-Numbers* category. The abstract root of this hierarchy is `Magnitude`, which represents all kinds of classes supporting comparison operators. `Number` adds various arithmetic

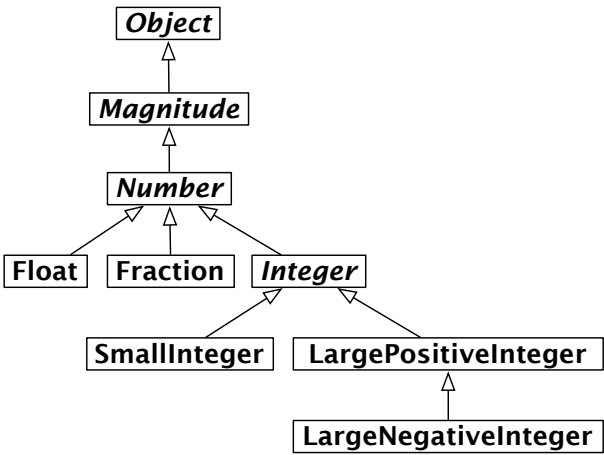


FIG. 8.1 – The Number Hierarchy

and other operators as mostly abstract methods. Float and Fraction represent, respectively, floating point numbers and fractional values. Integer is also abstract, thus distinguishing between subclasses SmallInteger, LargePositiveInteger and LargeNegativeInteger. For the most part users do not need to be aware of the difference between the three Integer classes, as values are automatically converted as needed.

Magnitude

Magnitude is the parent not only of the Number classes, but also of other classes supporting comparison operations, such as Character, Duration and Timespan. (Complex numbers are not comparable, and so do not inherit from Number.)

Methods < and = are abstract. The remaining operators are generically defined. For example :

Méthode 8.12 – Abstract comparison methods

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

Similarly, Number defines +, -, \* and / to be abstract, but all other arithmetic operators are generically defined.

All Number objects support various *converting* operators, such as asFloat and asInteger. There are also numerous *shortcut constructor methods*, such as i, which converts a Number to an instance of Complex with a zero real component, and others which generate Duration s, such as hour, day and week.

Numbers directly support common *math functions* such as sin, log, raiseTo:, squared, sqrt and so on.

Number»printOn: is implemented in terms of the abstract method Number»printOn:base:. (The default base is 10.)

Testing methods include even, odd, positive and negative. Unsurprisingly Number overrides isNumber . More interesting, isInfinite is defined to return false.

*Truncation* methods include floor, ceiling, integerPart, fractionPart and so on.

1 + 2.5	→	3.5	"Addition of two numbers"
3.4 * 5	→	17.0	"Multiplication of two numbers"
8 / 2	→	4	"Division of two numbers"
10 - 8.3	→	1.7	"Subtraction of two numbers"
12 = 11	→	false	"Equality between two numbers"
12 ~= 11	→	true	"Test if two numbers are different"
12 > 9	→	true	"Greater than"
12 >= 10	→	true	"Greater or equal than"
12 < 10	→	false	"Smaller than"
100@10	→	100@10	"Point creation"


The following example works surprisingly well in Smalltalk :

---

1000 factorial / 999 factorial     $\longrightarrow$     1000

---

Note that 1000 factorial is really calculated which in many other languages can be quite difficult to compute. This is an excellent example of automatic coercion and exact handling of a number.

 *Try to display the result of 1000 factorial. It takes more time to display it than to calculate it !*

## Float

Float implements the abstract Number methods for floating point numbers.

More interestingly, Float class (*c-à-d.* the class-side of Float) provides methods to return the following *constants* : e, infinity, nan and pi.

---

Float pi	$\longrightarrow$	3.141592653589793
Float infinity	$\longrightarrow$	Infinity
Float infinity isInfinite	$\longrightarrow$	true

---

## Fraction

Fractions are represented by instance variables for the numerator and denominator, which should be Integers. Fractions are normally created by Integer division (rather than using the constructor method Fraction>numerator:denominator:) :

---

6/8	$\longrightarrow$	(3/4)
(6/8) class	$\longrightarrow$	Fraction

---

Multiplying a Fraction by an Integer or another Fraction may yield an Integer :

---

6/8 \* 4     $\longrightarrow$     3

---

## Integer

Integer is the abstract parent of three concrete integer implementations. In addition to providing concrete implementations of many abstract Number methods, it also adds a few methods specific to integers, such as factorial, atRandom, isPrime, gcd: and many others.

SmallInteger is special in that its instances are represented compactly — instead of being stored as a reference, a SmallInteger is represented directly using the bits that would otherwise be used to hold a reference. The first bit of an object reference indicates whether the object is a SmallInteger or not.

The class methods minVal and maxVal tell us the range of a SmallInteger :

---

SmallInteger maxVal = ((2 raisedTo: 30) - 1)	→	true
SmallInteger minVal = (2 raisedTo: 30) negated	→	true

---

When a SmallInteger goes out of this range, it is automatically converted to a LargePositiveInteger or a LargeNegativeInteger, as needed :

---

(SmallInteger maxVal + 1) class	→	LargePositiveInteger
(SmallInteger minVal - 1) class	→	LargeNegativeInteger

---

Large integers are similarly converted back to small integers when appropriate.

As in most programming languages, integers can be useful for specifying iterative behavior. There is a dedicated method timesRepeat: for evaluating a block repeatedly. We have already seen a similar example in le chapitre 3 :

---

```
n := 2.
3 timesRepeat: [ n := n*n ].
n → 256
```

---

## 8.3 Characters

Character is defined in the *Collections-Strings* category as a subclass of Magnitude. Printable characters are represented in Squeak as  $\$(char)$ . For example :

---

```
$a < $b    →    true
```

---

Non-printing characters can be generated by various class methods. Character class »value: takes the Unicode (or ASCII) integer value as argument and returns the corresponding character. The protocol *accessing untypeable characters* contains a number of convenience constructor methods such as backspace, cr, escape, euro, space, tab, and so on.

---

```
Character space = (Character value: Character space asciiValue)    →    true
```

---

The printOn: method is clever enough to know which of the three ways to generate characters offers the most appropriate representation :

---

```
Character value: 1    →    Character value: 1
Character value: 32    →    Character space
Character value: 97    →    $a
```

---

Various convenient *testing* methods are built in : isAlphaNumeric, isCharacter, isDigit, isLowercase, isVowel, and so on.

To convert a Character to the string containing just that character, send asString. In this case asString and printString yield different results :

---

```
$a asString    →    'a'
$a             →    $a
$a printString →    '$a'
```

---

Every ascii Character is a unique instance, stored in the class variable CharacterTable :

---

```
(Character value: 97) == $a    →    true
```

---

Characters outside the range 0 to 255 are not unique, however :

---

Character characterTable size	→	256
(Character value: 500) == (Character value: 500)	→	false

---

# 8.4 Strings

The String class is also defined in the category *Collections-Strings*. A String is an indexed Collection that holds only Characters.

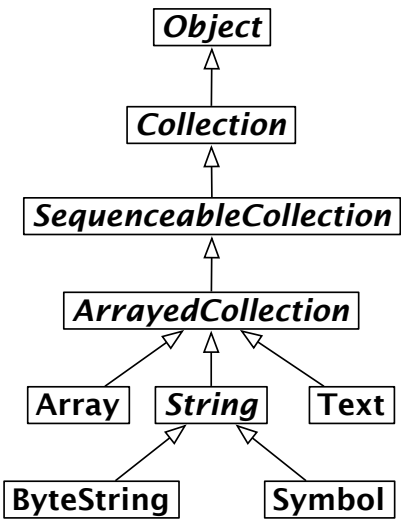


FIG. 8.2 – The String Hierarchy

In fact, String is abstract and Squeak Strings are actually instances of the concrete class ByteString.

---

'hello world' class	→	ByteString
---------------------	---	------------

---

The other important subclass of String is Symbol. The key difference is that there is only ever a single instance of Symbol with a given value. (This is sometimes called “the unique instance property”). In contrast,



two separately constructed Strings that happen to contain the same sequence of characters will often be different objects.

---

```
'hel','lo' == 'hello'  →  false
```

---

---

```
('hel','lo') asSymbol == #hello  →  true
```

---

Another important difference is that a String is mutable, whereas a Symbol is immutable.

---

```
'hello' at: 2 put: $u; yourself  →  'hullo'
```

---

---

```
#hello at: 2 put: $u  →  error
```

---

It is easy to forget that since strings are collections, they understand the same messages that other collections do :

---

```
#hello indexOf: $o  →  5
```

---

Although String does not inherit from Magnitude, it does support the usual *comparing* methods, <, = and so on. In addition, String#match: is useful for some basic glob-style pattern-matching :

---

```
'*or*' match: 'zorro'  →  true
```

---

Should you need more advanced support for regular expressions, there are a number of third party implementations available, such as Vassili Bykov's Regex package.

Strings support rather a large number of conversion methods. Many of these are shortcut constructor methods for other classes, such as asDate, asFileName and so on. There are also a number of useful methods for converting a string to another string, such as capitalized and translateToLowercase.

For more on strings and collections, see le chapitre 9.

## 8.5 Booleans

The class `Boolean` offers a fascinating insight into how much of the Smalltalk language has been pushed into the class library. `Boolean` is the abstract superclass of the Singleton classes `True` and `False`.

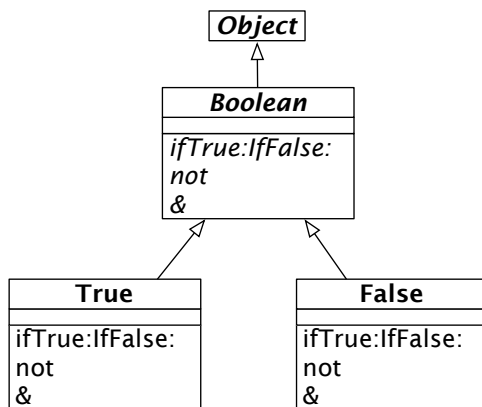


FIG. 8.3 – The Boolean Hierarchy

Most of the behaviour of Booleans can be understood by considering the method `ifTrue:ifFalse:`, which takes two `Blocks` as arguments.

---

(4 factorial > 20) ifTrue: [ 'bigger' ] ifFalse: [ 'smaller' ] → 'bigger'

---

The method is abstract in `Boolean`. The implementations in its concrete subclasses are both trivial :

---

### Méthode 8.13 – Implementations of `ifTrue:ifFalse:`

---

`True`»ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock  
 ↑trueAlternativeBlock value

`False`»ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock  
 ↑falseAlternativeBlock value

---

In fact, this is the essence of OOP : when a message is sent to an object, the object itself determines which method will be used to

respond. In this case an instance of `True` simply evaluates the *true* alternative, while an instance of `False` evaluates the *false* alternative. All the abstract Boolean methods are implemented in this way for `True` and `False`. For example :

Méthode 8.14 – *Implementing negation*

---

```
True»not
  "Negation--answer false since the receiver is true."
↑false
```

---


Booleans offer several useful convenience methods, such as `ifTrue;`, `ifFalse;`, `ifFalse;ifTrue`. You also have the choice between eager and lazy conjunctions and disjunctions.

---

<code>(1&gt;2) &amp; (3&lt;4)</code>	→	<code>false</code>	<i>"must evaluate both sides"</i>
<code>(1&gt;2) and: [ 3&lt;4 ]</code>	→	<code>false</code>	<i>"only evaluate receiver"</i>
<code>(1&gt;2) and: [ (1/0) &gt; 0 ]</code>	→	<code>false</code>	<i>"argument block is never evaluated, so no exception"</i>

---

In the first example, both Boolean subexpressions are evaluated, since `&` takes a Boolean argument. In the second and third examples, only the first is evaluated, since `and:` expects a `Block` as its argument. The `Block` is evaluated only if the first argument is true.

 Try to imagine how `and:` and `or:` are implemented. Check the implementations in `Boolean`, `True` and `False`.

## 8.6 Chapter summary

- If you override `=` then you should override `hash` as well.
- Override `postCopy` to correctly implement copying for your objects.
- Send `self halt` to set a breakpoint.
- Return `self subclassResponsibility` to make a method abstract.
- To give an object a `String` representation you should override `printOn:`.
- Override the hook method `initialize` to properly initialize instances.

- Number methods automatically convert between Floats, Fractions and Integers.
- Fractions truly represent rational numbers rather than floats.
- Characters are unique instances.
- Strings are mutable ; Symbols are not. Take care not to mutate string literals, however !
- Symbols are unique ; Strings are not.
- Strings and Symbols are Collections and therefore support the usual Collection methods.

## Chapitre 9

# Les collections

### 9.1 Introduction

Les classes de collection forment un groupe de sous-classes de *Collection* et de *Streams* (pour flux de données) faiblement couplées destiné à un usage générique. Ce groupe de classes mentionné dans la bible de Smalltalk nommée “Blue Book”<sup>1</sup> (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.

En Squeak, les classes abstraites *Collection* et *Stream* disposent respectivement de 98 et de 39 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 applications et ne sont par conséquent pas organisées dans la catégorie “Collections”.

Dans ce chapitre, nous réunirons *Collection* et ses 37 sous-classes aussi présentes dans les catégories-système de la forme *Collections-\**

---

<sup>1</sup>2,.

sous le terme de “hiérarchie de collection” et Stream et ses 10 sous-classes de la catégorie *Collections-Streams* sous celui de “hiérarchie de stream”. La liste complète apparaît sur la figure 9.1. Ces 49 classes répondent à 794 messages définissant un total de 1236 méthodes !

Dans ce chapitre, nous nous attarderons principalement sur un sous-ensemble de classes de collection montré sur la figure 9.2. Les flux de données ou *streams* seront abordés séparément dans le chapitre 10.

## 9.2 Les variétés de collections

Pour faire bon usage des classes de collection, 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. Cependant Smalltalk-80 a adopté la programmation basée sur les collections comme précept 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 des enregistrements d’étudiants appelé *students* (pour étudiant, 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 dans une boucle. Mais le programmeur en Smalltalk écrira :

---

```
students select: [ :each | each gpa < threshold ]
```

---

ce qui donnera une nouvelle collection contenant précisément ces éléments de *students* (étudiants) pour lesquels la fonction entre crochets

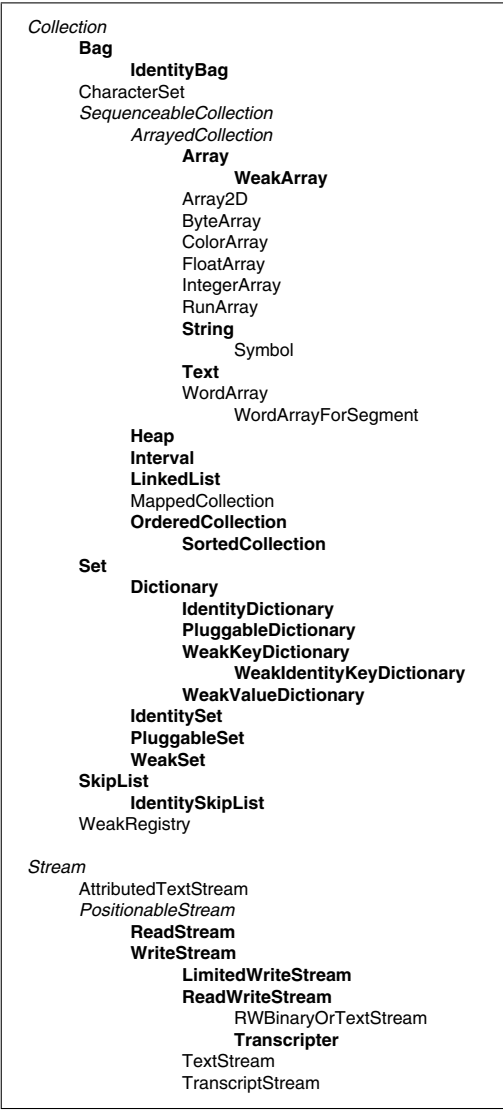


FIG. 9.1 – Les classes de collection dans Squeak. L’indentation indique la hiérarchie : Les classes *en italique* sont abstraites.

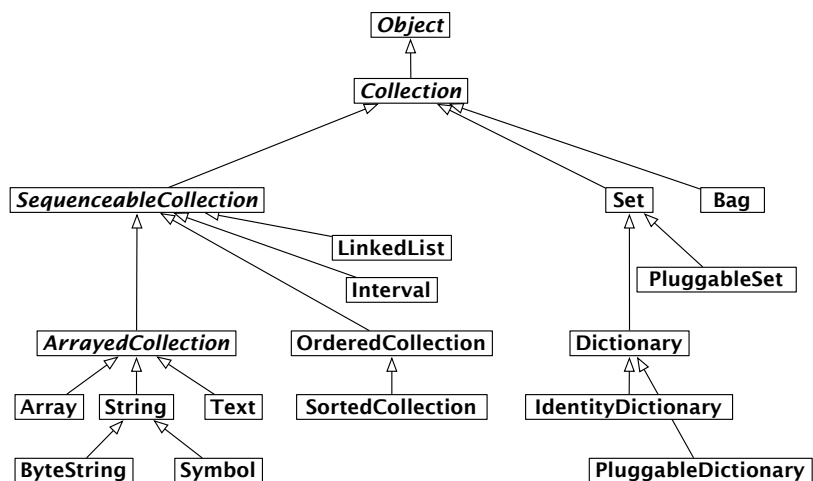


FIG. 9.2 – Certaines des classes majeures de collection de Squeak.

renvoie une réponse positive *c-à-d.* `true`<sup>2</sup>. 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 les protocoles bien définis pour tester l'appartenance et énumérer les éléments. *Toutes* les collections accepte les messages de la catégorie des tests nommée *testing* tels includes: (test d'inclusion), `isEmpty` (test de

<sup>2</sup>L'expression entre crochets (brackets en anglais) peut être vue comme une  $\lambda$  expression définissant une fonction anonyme  $\lambda x.x \text{ gpa} < \text{threshold}$ .



Protocole	Méthodes
<i>accessing</i>	size , capacity , at: <i>anIndex</i> , at: <i>anIndex</i> put: <i>anElement</i>
<i>testing</i>	isEmpty , includes: <i>anElement</i> , contains: <i>aBlock</i> , occurrencesOf: <i>anElement</i>
<i>adding</i>	add: <i>anElement</i> , addAll: <i>aCollection</i>
<i>removing</i>	remove: <i>anElement</i> , remove: <i>anElement</i> ifAbsent: <i>aBlock</i> , removeAll: <i>aCollection</i>
<i>enumerating</i>	do: <i>aBlock</i> , collect: <i>aBlock</i> , select: <i>aBlock</i> , reject: <i>aBlock</i> , detect: <i>aBlock</i> , detect: <i>aBlock</i> ifNone: <i>aNoneBlock</i> , inject: <i>aValue</i> into: <i>aBinaryBlock</i>
<i>converting</i>	asBag , asSet , asOrderedCollection , asSortedCollection , asArray , asSortedCollection: <i>aBlock</i>
<i>creation</i>	with: <i>anElement</i> , with:with: , with:with:with: , with:with:with:with: , withAll: <i>aCollection</i>

FIG. 9.3 – Les protocoles standards de collection

virginité) et occurrencesOf: (test d’occurences 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: (détection 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 tant l’omniprésence de ce protocole que sa diversité qui le rend si puissant.

La figure 9.3 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 Collection.

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 une premier

élément dit *first* et progresse dans un ordre bien défini jusqu'au dernier élément dit *last*. Les instances de *Set*, *Bag* (ou multien-semble) 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 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-modifiables 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 du tableau *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

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

FIG. 9.4 – Certaines classes de collection rangées selon leur technique d’implémentation.

(IdentityDictionary, IdentitySet et IdentityBag) utilisent la méthode == qui teste si les arguments sont le même objet et les variantes Pluggable emploie 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 suffisante ; nous devons aussi regarder les classes de collection selon leur implémentation. Comme nous le montre la figure 9.4, cinq techniques d’implémentations 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 par delà les 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 un tableau 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 représentées 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 au ramasse-miettes ou *garbage collector*. La machine virtuelle Squeak 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 Pugh<sup>3</sup>.

## 9.4 Exemples de classes importantes

Nous présentons maintenant les classes de collection 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`,

---

<sup>3</sup>?, .

`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 l'interprète pour convenir à sa sémantique. 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 collection 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 la taille `anInteger` dont les éléments seront tous nulles *c-à-d.* de valeur `nil`. `with: anObject` crée une collection et ajoute `anObject` à la collection créée. Les collections réalisent cela de différente manière.

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 la taille `anInteger`.  
`Array new: 5` crée un tableau de la 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 (ou "literals") qui doivent être connus quand l'expression est compilée et non lorsqu'elle est exécutée. Le code suivant crée un tableau de la 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 base `#()` a pour conséquence le fait que le compilateur interprète littéralement les expressions contenues dans le tableau. L'expression est scannée 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.

---

```
{ 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
```

---

Sois 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 d'objet (*c-à-d.* elle est vierge), 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 être effacé 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 un arrangement de nombres. Par exemple, l'intervalle de chiffres 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 accédés 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 considérés comme étant la même si elles retournent *true* lorsqu'elles sont comparées par `=`. Un 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 collection est basée sur une héritage et pas 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 la même, 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 ou 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 fois pour toute 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 Symbol

s (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 le chapitre 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 accédés par le message de recherche par indice `at:`. Les objets à inclure dans `Set` devraient 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 duplicata :

---

```
{ 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:`. Premièrement, le receveur est 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 *iterators* (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 voulez trier des objets qui ne définissent pas la méthode <= ou que vous voulez trier selon une 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 créé 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 `WideString`<sup>4</sup> en 32 bits. Pour garder les choses simples, 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*).

---

```
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.

---

<sup>4</sup>*Wide* a le sens : étendu

---

```
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 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 aussi celles de 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 ou *String matching*** 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
```

---

Il existe aussi des techniques d'appariements plus avancés par filtre offrant les même possibilités que Perl mais elles ne sont pas in-



cluses par défaut dans l'image standard<sup>5</sup>. seeindexregular expression packagepaquetage d'expressions régulières

**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.
'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: {'Squeak' . 'extra'}    → 'Squeak 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: 'Squeak' with: 'extra' → 'Squeak est
extra'
'<2s> est <1s>' expandMacrosWith: 'Squeak' with: 'extra' → 'extra est
Squeak'
'<1p> ou <1s>' expandMacrosWith: 'Squeak' with: 'extra' → '"Squeak" ou
Squeak'
'<1?Quentin:Thibaut> joue' expandMacrosWith: true    → 'Quentin joue'
'<1?Quentin:Thibaut> joue' expandMacrosWith: false   → 'Thibaut joue'
```

---

<sup>5</sup>Nous vous recommandons fortement le paquetage d'expressions régulières ou *regular expression package* de Vassili Bykov, disponible à [www.squeaksource.com/Regex.html](http://www.squeaksource.com/Regex.html).

**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, et 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 bas niveau comme `to:do:` qui évalue un bloc avec un argument référant le parcours entre le nombre initial et final, la hiérarchie de collection `Smalltalk` offre de nombreux itérateurs 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 contenu 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"
```

---

## Collectionner 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 numérique 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 générons 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 !

---

```
(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 affirme le test passé en bloc-argument. (`isVowel` retourne vrai *c-à-d.* `true` si le receveur est une voyelle <sup>6</sup>.)

---

```
'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 | '*java*' match: each asString] ifNone: [ nil ]  
→ nil
```

---

## Accumuler les résultats avec inject:into:

Les langages de programmation fonctionnelle offre souvent une fonction d'ordre supérieure appelée *fold* ou *reduce* pour accumuler un résultat en appliquant un opérateur binaire de manière itérative sur tous les éléments d'une collection. Squeak 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. Nous pouvons écrire cette expression en Squeak 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 les factoriels :

---

<sup>6</sup>Note du traducteur : les voyelles accentuées ne sont pas pas considérées par défaut comme des voyelles ; Smalltalk-80 a la même tare que la plupart des vieux langages de programmation nés dans la culture anglosaxonne.

---

```
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 ]  →
14
```

---

**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 n'est pas affecté par 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 dans laquelle vous itérez.

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.

---

```
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 devait avoir été filtré !

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 Set ainsi que d'autres phénomènes plus étrange. Une solution proposée par Kent Beck est d'utiliser `xor` pour redéfinir hash. Supposons que nous voulons que deux livres soient considérés comme égaux si leurs titres et leurs auteurs sont les mêmes. Alors nous redéfinirions non seulement = mais aussi hash comme suit :



## Méthode 9.1 – Redéfinir = et hash .

---

Book»= aBook

self class = aBook class iffFalse: [↑ false].

↑ title = aBook title and: [ authors = aBook authors]

Book»hash

↑ title hash xor: authors hash

---

Un autre problème terrifiant 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 au moins que vous n'aimez vraiment le débogage !

## 9.7 Résumé du chapitre

La hiérarchie de collection 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 enregistre des associations clé-valeur et les ensembles *Set* ou *Bag* qui sont deux 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 *Array* 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 *Symbol* sont utilisés comme clés

ou que la correspondance soit établie sur 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 le traitement uniforme des collections.
- Ne jamais effacer un élément d'une collection que vous parcourez itérativement. Si vous la devez 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

# Stream : les flux de données

Les flux de données ou *streams* sont utilisés pour itérer dans une séquence d'éléments comme des collections, des fichiers et des flux réseau. Les *streams* peuvent être lisible ou inscriptible 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 une séquence 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 plus part 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`.

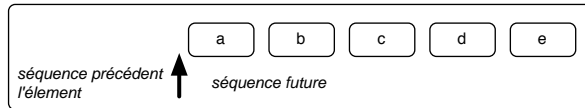


FIG. 10.1 – Un flux de données positionné à son origine.

Lire un élément correspond 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.

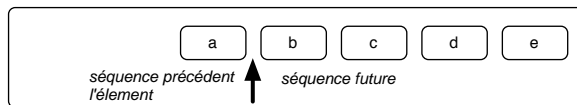


FIG. 10.2 – Le même flux de données après l'exécution de la méthode `next` : le caractère `a` est "dans le passé" alors que `b`, `c`, `d` and `e` sont "dans le futur".

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`.

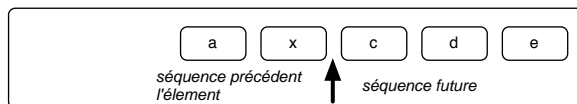


FIG. 10.3 – Le même flux de données après avoir écrit un `x`.

## 10.2 Streams contre Collections

Le protocole de collection supporte le stockage, l'effacement et l'énumération des éléments d'une collection mais il ne permet pas que ces opérations soient combinés 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 continue et laquelle ne le fait. De telles procédures requiert 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 dans*<sup>1</sup> une collection. Par exemple, le code suivant crée un *stream* sur un intervalle puis y lit deux éléments.

---

```
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 supporte les protocoles de lecture et d'écriture.

Le principal problème de `WriteStream` et de `ReadWriteStream` est que, dans Squeak, ils ne supportent que les tableaux et les chaînes de caractères. Cette limitation est en cours de disparation grâce au développement d'une nouvelle librairie nommée *Nile* mais en attendant, vous obtiendrez une erreur si vous essayez d'utiliser les *streams* avec un autre type de collection :

---

<sup>1</sup>En anglais, nous dirions "stream over".

---

```
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, séparés par un retour-chariot et enfin ferme le fichier.

---

```
StandardFileStream
  fileName: '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.

### 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 utilisés 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'index, 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
```

---

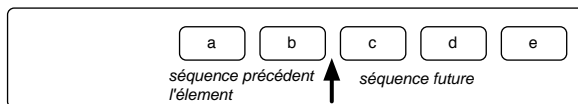


FIG. 10.4 – Un flux de données à la position 2.

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 la 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.   →      "le flux est après d"
stream position.  →      4
stream skip: -2.  →      2  "le flux est après b"
stream position.  →      2
stream reset.     →
stream position.  →      0
stream skipTo: $e. →      "le flux est après e"
stream next.     →  $f
```



```
stream contents.  —→  'abcdef'
```

---

Comme vous pouvez le voir, la lettre e a été sauté.

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 paramètre 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 274) 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 flex 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 appellerons 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.

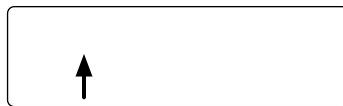


FIG. 10.5 – Un nouvel historique est vide. Rien n'est affiché dans le navigateur web.

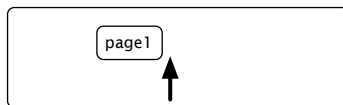


FIG. 10.6 – L'utilisateur ouvre la page 1.

Ce comportement peut être programmé avec un `ReadWriteStream`.



FIG. 10.7 – L'utilisateur clique sur un lien vers la page 2.



FIG. 10.8 – L'utilisateur clique sur un lien vers la page 3.



FIG. 10.9 – L'utilisateur clique sur le bouton “Retour” (Back). Il visite désormais la page 2 à nouveau.

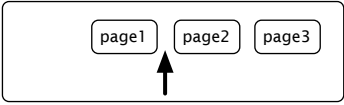


FIG. 10.10 – L'utilisateur clique sur le bouton “Retour” (Back). La page 1 est affichée maintenant.

---

Object subclass: #History  
instanceVariableNames: 'stream'  
classVariableNames: "  
poolDictionaries: "  
category: 'SBE-Streams'

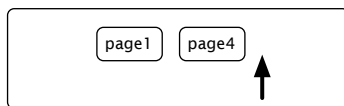


FIG. 10.11 – Depuis la page 1, l'utilisateur clique sur un lien vers la page 4. L'historique oublie les pages 2 et 3.

```
History>>initialize
  super initialize.
  stream := ReadWriteStream on: Array new.
```

---

Nous n'avons rien de compliqué ici ; nous définissons une nouvelle classe qui contient une *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 back
```

```
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*: (traductible 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 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 collections 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 répond 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 existante avec le nom donné.



Vous devez vous remémorer de fermer le *stream* sur 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 premièrement 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 streams 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. → a ByteArray(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 graphisme 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.

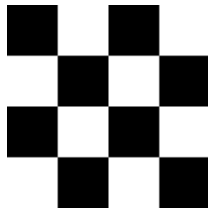


FIG. 10.12 – Un échiquier 4 par 4 que vous pouvez dessiner en utilisant des *streams* binaires.

## 10.5 Résumé du chapitre

Par rapport aux collections, les flux de données ou *streams* offre une bien meilleur façon de lire et 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 streams peuvent être soit lisibles, soit inscriptibles, soit à la fois lisible et inscriptible.
- 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.

Troisième partie

## **Squeak avancé**



## Chapitre 11

# Classes et méta-classes

Comme nous l'avons vu dans le chapitre 5, en Smalltalk, tout est un 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 capture 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, quelque soit la situation.

### 11.1 Les règles pour les classes et les méta-classes

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 un 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 méta-classe est automatiquement créée pour chaque nouvelle classe. La plupart du temps, vous n'avez pas besoin de vous soucier ou de penser aux méta-classes. Cependant, chaque fois que vous utilisez le System 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 méta-classe sont deux classes inséparables, même si la première est une instance de la seconde. Pour expliquer correctement les classes et les méta-classes, nous devons étendre les règles du chapitre 5 en ajoutant les règles suivantes :

**Règle 6.** Toute classe est une instance d'une méta-classe.

**Règle 7.** La hiérarchie des méta-classes est parallèle à celle des classes.

**Règle 8.** Toute méta-classe hérite de Class et de Behavior.

**Règle 9.** Toute méta-classe est une instance de Metaclass.

**Règle 10.** La méta-classe 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.



## 11.2 Retour sur le modèle objet de Smalltalk

Puisque tout est un 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 figure 11.1.

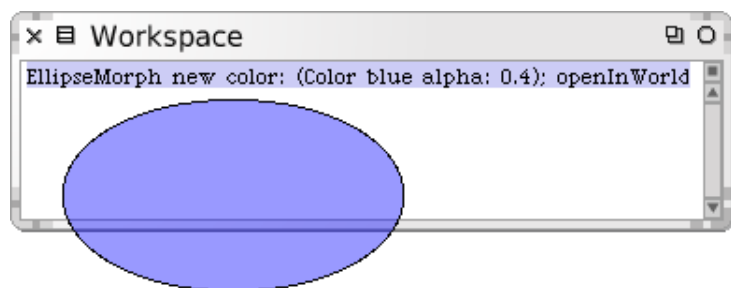


FIG. 11.1 – Une ellipse translucide.

D'après la Règle 3, tout 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), on peut 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 un objet bien que certains de ces objets soient aussi des classes.

La recherche de méthodes suit la chaîne d'héritage (Règle 5), donc quand on envoie 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 11.2.

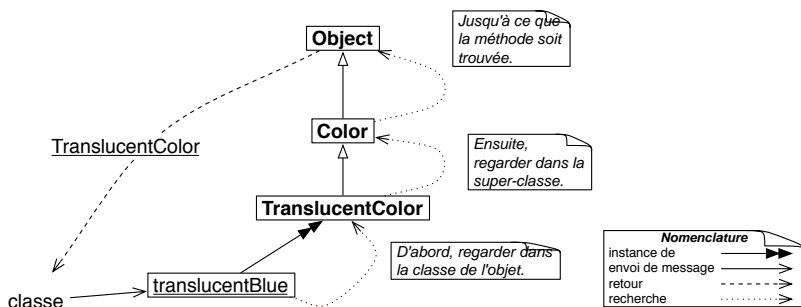


FIG. 11.2 – Envoyer un message à une couleur translucide.

Cette figure capture l'essence de la relation *est-un(e)*. Notre objet bleu translucide *est une* instance de TranslucentColor, mais on peut 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 :

---

translucentBlue := Color blue alpha: 0.4.		
translucentBlue isKindOf: TranslucentColor	→	true
translucentBlue isKindOf: Color	→	true
translucentBlue isKindOf: Object	→	true

---

### 11.3 Toute classe est une instance d'une méta-classe

Comme nous l'avons mentionné dans la section 11.1, les classes dont les instances sont aussi des classes sont appelées des méta-classes.

**Les méta-classes sont implicites.** Les méta-classes 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 méta-classe implicite est créée pour chaque classe que vous créez donc chaque méta-classe n'a qu'une seule instance.

Alors que les classes ordinaires sont nommées par des variables globales, les méta-classes sont anonymes. Cependant, on peut 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 11.3 montre que chaque classe est une instance de sa méta-classe (anonyme).

Le fait que les classes soient aussi des objets facilite leur interrogation par envoi de messages. Voyons cela :

---

Color subclasses	→	{TranslucentColor}
TranslucentColor subclasses	→	#()

---

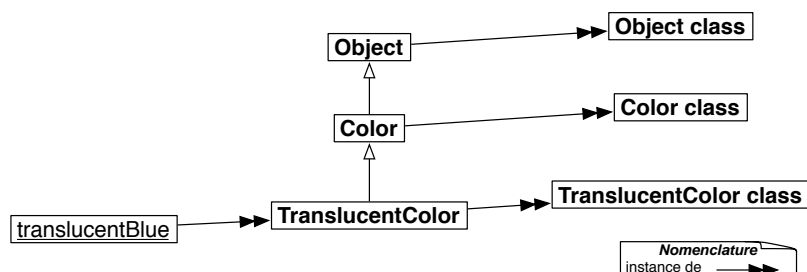


FIG. 11.3 – Les méta-classes de TranslucentColor et ses super-classes.

TranslucentColor allSuperclasses	→	an OrderedCollection(Color Object ProtoObject)
TranslucentColor instVarNames	→	#('alpha')
TranslucentColor allInstVarNames	→	#('rgb' 'cachedDepth' 'cachedBitPattern' 'alpha')
TranslucentColor selectors	→	an IdentitySet(#alpha: #asNontranslucentColor #privateAlpha #pixelValueForDepth: #isOpaque #isTranslucentColor #storeOn: #pixelWordForDepth: #scaledPixelValue32 #alpha #bitPatternForDepth: #hash #convertToCurrentVersion:refStream: #isTransparent #isTranslucent #setRgb:alpha: #balancedPatternForDepth: #storeArrayValuesOn:)

## 11.4 La hiérarchie des méta-classes est parallèle à celle des classes

La Règle 7 dit que la super-classe d'une méta-classe ne peut pas être une classe arbitraire : elle est contrainte à être la méta-classe de la super-classe de l'unique instance de cette méta-classe.

TranslucentColor class superclass	→	Color class
TranslucentColor superclass class	→	Color class

C'est ce que nous voulons dire par le fait que la hiérarchie des méta-classes est parallèle à la hiérarchie des classes ; la figure 11.4 montre comment cela fonctionne pour la hiérarchie de TranslucentColor.

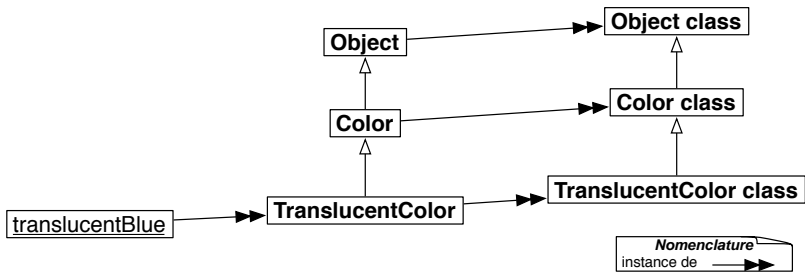


FIG. 11.4 – La hiérarchie des méta-classes est parallèle à la hiérarchie des classes.

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 le chaîne d’héritage.

Ainsi, les messages envoyés à des classes doivent suivre la chaîne d’héritage des méta-classes. 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 méta-classes jusqu’à trouver dans Color class (see la figure 11.5).

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 !

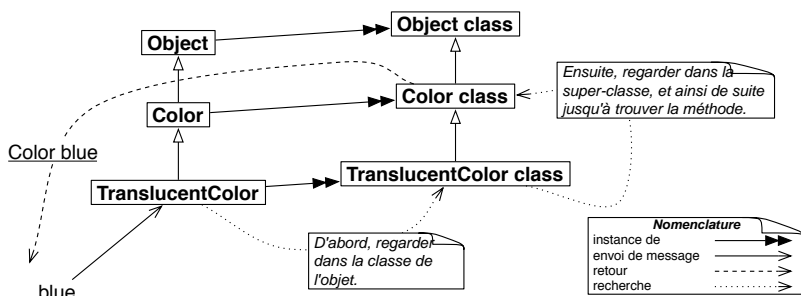


FIG. 11.5 – Le traitement des messages pour les classes est le même que pour les objets ordinaires.

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 ce message, mais si vous avez une bonne raison pour faire cela, il n'y a rien qui vous empêche d'ajouter une méthode `new` à une classe qui n'est pas une méta-classe.

Comme les classes sont des objets, on peut aussi les inspecter.

 *Inspectez Color blue et Color.*

Notons que dans un cas, vous inspectez une instance de `Color` et que 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 montré dans la figure 11.6.

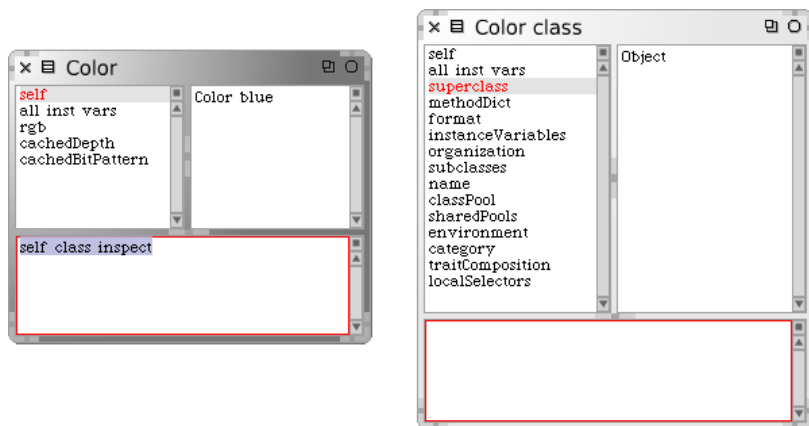


FIG. 11.6 – Les classes sont aussi des objets.

## 11.5 Toute méta-classe hérite de Class et de Behavior

Toute méta-classe *est-une* classe, donc hérite de Class. À son tour, Class hérite de ses super-classes, ClassDescription et Behavior. En Small-talk, puisque tout *est-un* objet, ces classes héritent finalement toutes de Object. Nous pouvons voir le schéma complet dans la figure 11.7.

**Où est défini new ?** Pour comprendre l'importance du fait que les méta-classes héritent de Class et de Behavior, demandons-nous où est défini new et comment cette définition est trouvée. Quand le message new est envoyé à une classe, il est recherché dans sa chaîne de méta-classes et finalement dans ses super-classes Class, ClassDescription et Behavior comme montré dans la figure 11.8.

La question « Où est défini new ? » est cruciale. new est défini en premier dans la classe Behavior et peut être redéfini dans ses sous-classes, ce qui inclut toutes les méta-classes des classes que nous avons

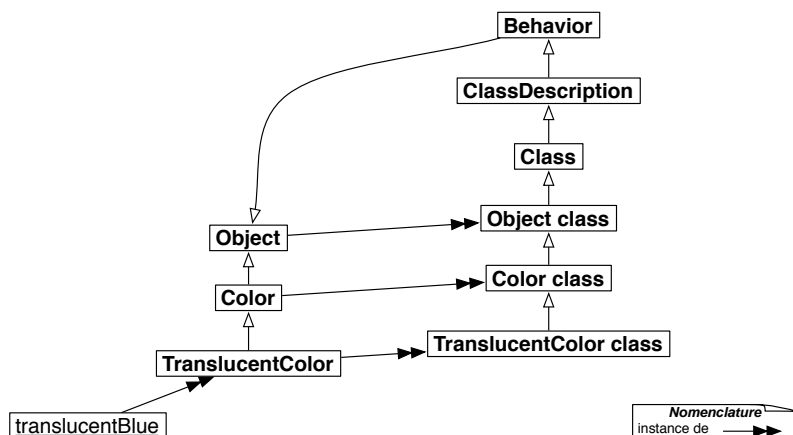


FIG. 11.7 – Les méta-classes héritent de Class et de Behavior

définies, si cela est nécessaire. Maintenant, quand un message `new` est envoyé à une classe, il est recherché, comme d'habitude, dans la méta-classe 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`"

---

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` créer un tableau de 4 éléments. Vous ne trouverez pas la définition de cette méthode dans `Array` ni dans aucune de ses super-



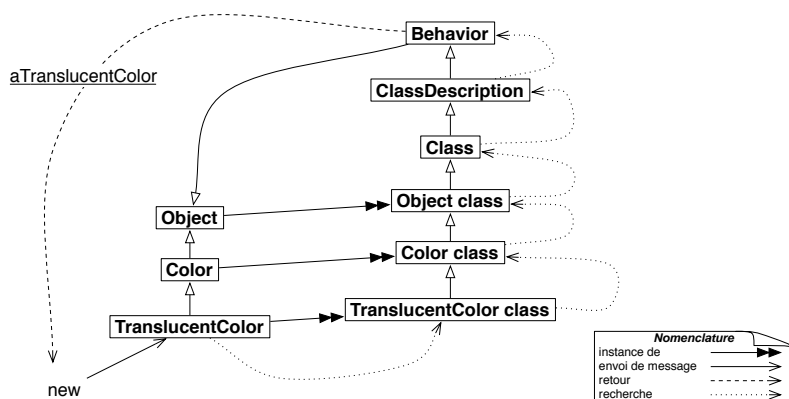


FIG. 11.8 – `new` est un message ordinaire recherché dans la chaîne des méta-classes.

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 méta-classes doivent finalement hériter de Class.

## 11.6 Toute méta-classe est une instance de Metaclass

Les méta-classes sont aussi des objets ; elles sont des instances de la classe Metaclass comme montré dans la figure 11.9. Les instances de la classe Metaclass sont les méta-classes anonymes ; chacune ayant exactement une unique instance qui est une classe.

Metaclass représente le comportement commun des méta-classes. Elle fournit des méthodes pour la création d'instance (subclassOf: ) 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).

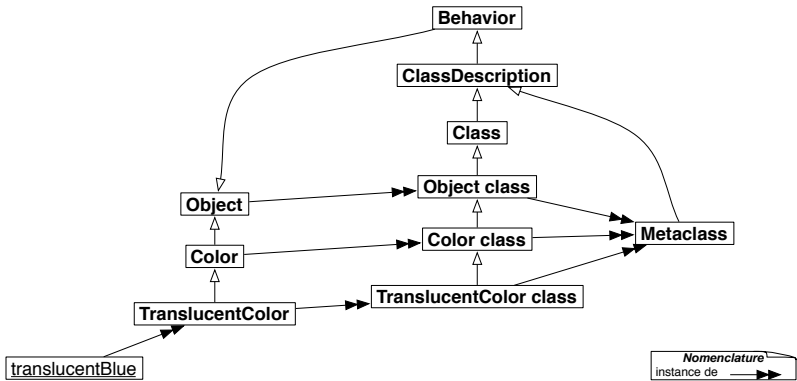


FIG. 11.9 – Toute méta-classe est une Metaclass .

## 11.7 La méta-classe 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 méta-classe, donc forcément une instance de Metaclass, exactement comme toutes les autres méta-classes dans le système (voir la figure 11.10).

La figure montre que toutes les méta-classes sont des instances de Metaclass, ce qui inclut aussi la méta-classe de Metaclass. Si vous comparez les figures 11.9 et 11.10, vous verrez comment la hiérarchie des méta-classes 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 11.10 est correcte. (En réalité, vous verrez que nous avons dit un pieux mensonge — Object class superclass → ProtoObject class, et non Class. En Squeak, il faut aller une super-classe plus haut dans la hiérarchie pour atteindre Class.)

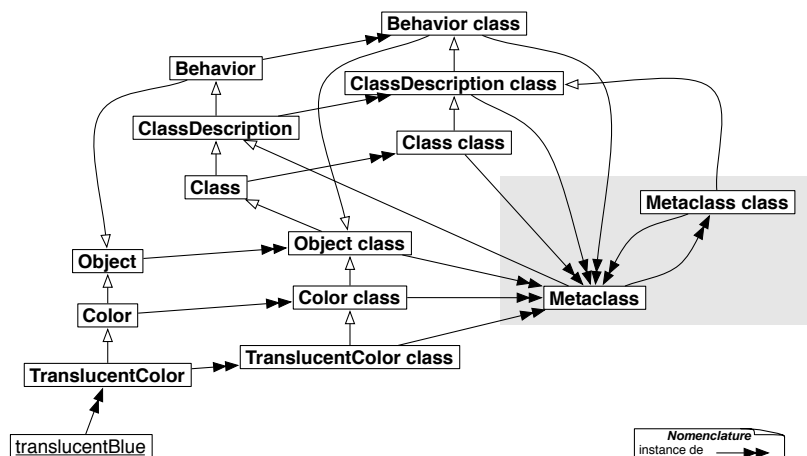


FIG. 11.10 – Toutes les méta-classes sont des instances de la classe Metaclass , même la méta-classe de Metaclass .

#### Exemple 11.1 – La hiérarchie des classes

TranslucentColor superclass	→	Color
Color superclass	→	Object

#### Exemple 11.2 – La hiérarchie parallèle des méta-classes

TranslucentColor class superclass	→	Color class
Color class superclass	→	Object class
Object class superclass superclass	→	Class "Attention : saute ProtoObject class"
Class superclass	→	ClassDescription
ClassDescription superclass	→	Behavior
Behavior superclass	→	Object

#### Exemple 11.3 – Les instances de Metaclass

TranslucentColor class class	→	Metaclass
Color class class	→	Metaclass
Object class class	→	Metaclass
Behavior class class	→	Metaclass

Exemple 11.4 – Metaclass class *est une* Metaclass

---

Metaclass class class	→	Metaclass
Metaclass superclass	→	ClassDescription

---

11.8 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 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 méta-classe. Les méta-classes sont implicites. Une méta-classe est créée automatiquement à chaque fois que vous créez une classe ; cette dernière étant sa seule instance.
2. La hiérarchie des méta-classes 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 méta-classes.
3. Toute méta-classe hérite de Class et de Behavior. Toute classe *est une* Class. Puisque les méta-classes sont aussi des classes, elles doivent hériter de Class. Behavior fournit un comportement commun à toutes les entités ayant des instances.
4. Toute méta-classe est une instance de Metaclass. ClassDescription fournit tout ce qui est commun à Class et à Metaclass.
5. La méta-classe de Metaclass est une instance de Metaclass. La relation *instance-de* forme une boucle fermée, donc Metaclass class class → Metaclass.



Quatrième partie

## **Annexes**





## Annexe A

# Foire Aux Questions

### A.1 Prémices

**FAQ 1** *Où puis-je trouver la dernière version de Squeak*

**Réponse** [ftp.squeak.org/current\\_development](http://ftp.squeak.org/current_development)

**FAQ 2** *Où est “l’Image de Développement” de Squeak ?*

**Réponse** [www.squeaksource.com/ImageForDevelopers](http://www.squeaksource.com/ImageForDevelopers) Ceci une image préparée spécifiquement pour les développeurs. Elle contient de multiples paquetages pré-installés pour les développeurs.

### 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]  →  'hle0'
```

---

## A.3 Naviguer dans le système

**FAQ 5** *Comment puis-je chercher une classe ?*

**Réponse** CMD-b (pour *browse c-à-d.* parcourir à l'aide du navigateur de classe) via le nom de la classe ou CMD-f dans le panneau de navigation des catégories de classes.

**FAQ 6** *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 7** *Comment puis-je naviguer dans toute la hiérarchie des messages de toutes les superclasses send ?*

### 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 8** *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 9** *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 10** *Comment puis-je créer une vue de l'arbre syntaxique abstrait ou AST d'une expression ?*

**Réponse** Charger le paquetage AST depuis squeaksource.com. Ensuite évaluer :

---

```
(RBParser parseExpression: '3+4') explore
```

---

(D'autre part *explorer le.*)

**FAQ 11** *Comment puis-je trouver tout les traits dans le système ?*

**Réponse**

---

```
Smalltalk allTraits
```

---

**FAQ 12** *Comment puis-je trouver quelles classes utilisent les traits ?*

**Réponse**

---

Smalltalk allClasses select: [:each | each hasTraitComposition ]

---

## A.4 Utilisation de Monticello et de Squeak-Source

**FAQ 13** *Comment puis-je charger un projet du Squeaksource ?*

**Réponse**

1. Trouvez le projet que vous souhaitez sur [squeaksource.com](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 14** *Comment puis-je créer un projet SqueakSource ?*

**Réponse**

1. Allez à [squeaksource.com](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 15** *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 insère dans votre package.

## A.5 Outils

**FAQ 16** *Comment puis-je ouvrir de manière pragmatique le SUnit TestRunner ?*

**Réponse** Évaluez TestRunner open.

**FAQ 17** *Où puis-je trouver le Refactoring Browser ?*

**Réponse** Chargez le paquetage AST puis le moteur de refactorisation sur le site squeaksource.com :

[www.squeaksource.com/AST](http://www.squeaksource.com/AST)

[www.squeaksource.com/RefactoringEngine](http://www.squeaksource.com/RefactoringEngine)

**FAQ 18** *Comment puis-je enregistrer le navigateur comme navigateur par défaut ?*

**Réponse** Cliquez sur l'icône du menu situé en haut à gauche de la fenêtre Browser à côté de la croix de destruction de la fenêtre. 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 19** *Comment puis-je travailler avec les expressions régulières ?*

**Réponse** Chargez le paquetage de RegEx de Vassili Bykov à l'adresse :

[www.squeaksource.com/Regex.html](http://www.squeaksource.com/Regex.html)

**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) <sup>1</sup> Smalltalk. Vous devrez installer au moins SmaCC-lr.13. Chargez-le depuis [www.squeaksource.com/SmaCCDevelopment.html](http://www.squeaksource.com/SmaCCDevelopment.html). Il y a un bon tutoriel en ligne à l'adresse : [www.refactory.com/Software/SmaCC/Tutorial.html](http://www.refactory.com/Software/SmaCC/Tutorial.html)

**FAQ 22** *Quels paquetages devrais-je charger depuis SqueakSource Smacc-Development pour écrire un analyseur grammatical ?*

**Réponse** Chargez la dernière version de SmaCCDev — le lanceur de programme est déjà actif. (Attention : SmaCC-Development est destiné à la version 3.8 de Squeak)

---

<sup>1</sup> En anglais, Compiler-Compiler.



