

Cours complet Pharo par l'exemple

Par Andrew Black - Stéphane Ducasse - Oscar Nierstrasz -
Damien Pollet - Damien Cassou - Marcus Denker - Martial Boniou
(traducteur) - René Mages (traducteur) - Serge Stinckwich (traducteur)

Date de publication : 9 juin 2021

Dernière mise à jour : 4 août 2021

TOUT PUBLIC

Ce cours se charge de vous apprendre le langage de programmation Pharo. Pharo est une implémentation moderne, libre et complète du langage de programmation Smalltalk et de son environnement. Pharo est un fork (1) de Squeak (2) , une réécriture de l'environnement Smalltalk-80 original.

Commentez

Préface.....	4
Qu'est-ce que Pharo ?.....	4
Qui devrait lire ce cours ?.....	4
Un petit conseil.....	4
Un livre ouvert.....	5
1 - Chapitre 1 - Une visite de Pharo.....	5
1-1 - Premiers pas.....	5
1-2 - Le menu World.....	9
1-3 - Envoyer des messages.....	10
1-4 - Enregistrer, quitter et redémarrer une session Pharo.....	11
1-5 - Les fenêtres Workspace et Transcript.....	12
1-6 - Les raccourcis-clavier.....	13
1-7 - Le navigateur de classes Class Browser.....	15
1-8 - Trouver des classes.....	16
1-9 - Trouver des méthodes.....	18
1-10 - Définir une nouvelle méthode.....	20
1-11 - Résumé du chapitre.....	24
2 - Chapitre 2 - Une première application.....	25
2-1 - Le jeu Lights Out.....	25
2-2 - Créer un nouveau paquetage.....	26
2-3 - Définir la classe LOCell.....	26
2-3-1 - À propos des catégories et des paquetages.....	26
2-3-2 - Créer une nouvelle classe.....	27
2-4 - Ajouter des méthodes à la classe.....	28
2-5 - Inspecter un objet.....	29
2-6 - Définir la classe LOGame.....	30
2-7 - Organiser les méthodes en protocoles.....	32
2-8 - Essayons notre code.....	35
2-9 - Sauvegarder et partager le code Smalltalk.....	38
2-9-1 - Les paquetages Monticello.....	38
2-9-2 - Sauvegarder et charger du code avec Monticello.....	39
2-9-3 - SqueakSource : un SourceForge pour Pharo.....	39
2-10 - Résumé du chapitre.....	41
3 - Chapitre 3 - Un résumé de la syntaxe.....	41
3-1 - Les éléments syntaxiques.....	42
3-2 - Les pseudovariables.....	44
3-3 - Les envois de messages.....	44
3-4 - Syntaxe relative aux méthodes.....	45
3-5 - La syntaxe des blocs.....	46
3-6 - Conditions et itérations.....	47
3-7 - Primitives et Pragmas.....	48
3-8 - Résumé du chapitre.....	48
4 - Chapitre 4 - Comprendre la syntaxe des messages.....	49
4-1 - Identifier les messages.....	49
4-2 - Trois sortes de messages.....	51
4-2-1 - Messages unaires.....	51
4-2-2 - Messages binaires.....	51
4-2-3 - Messages à mots-clefs.....	51
4-3 - Composition de messages.....	52
4-3-1 - Unaire > Binaire > Mots-clefs.....	52
4-3-2 - Les parenthèses en premier.....	54
4-3-3 - De gauche à droite.....	54
4-3-4 - Incohérences arithmétiques.....	55
4-4 - Quelques astuces pour identifier les messages à mots-clefs.....	57
4-4-1 - Des parenthèses ou pas ?.....	57
4-4-2 - Quand utiliser les [] ou les () ?.....	57
4-5 - Séquences d'expression.....	58
4-6 - Cascades de messages.....	58

4-7 - Résumé du chapitre.....	58
-------------------------------	----

Préface

Qu'est-ce que Pharo ?

Alors que Squeak fut développé principalement en tant que plateforme pour le développement de logiciels éducatifs expérimentaux, Pharo tend à offrir une plateforme, à la fois, open source et épurée pour le développement de logiciels professionnels et aussi, stable et robuste pour la recherche et le développement dans le domaine des langages et environnements dynamiques. Pharo est l'implémentation Smalltalk de référence de Seaside : le framework (3) (dit aussi « cadre d'applications ») destiné au développement web. Pharo résout les problèmes de licence inhérents à Squeak. Contrairement aux versions précédentes de Squeak, le noyau de Pharo ne contient que du code sous licence MIT. Le projet Pharo a débuté en mars 2008 depuis un fork de la version 3.9 de Squeak et la première version 1.0 bêta a été publiée le 31 juillet 2009. Bien que dépourvu de nombreux paquetages présents dans Squeak, Pharo est fourni avec beaucoup de fonctionnalités optionnelles dans Squeak. Par exemple, les fontes TrueType sont incluses dans Pharo. Pharo dispose aussi du support pour de véritables fermetures lexicales ou block closures. Les éléments d'interface utilisateur ont été revus et simplifiés.

Pharo est extrêmement portable — même sa machine virtuelle est entièrement écrite en Smalltalk, ce qui facilite son débogage, son analyse et les modifications à venir. Pharo est le véhicule d'un ensemble de projets innovants, des applications multimédias et éducatives aux environnements de développement pour le web. Il est important de préciser le fait suivant concernant Pharo : Pharo ne devrait pas être qu'une simple copie du passé, mais véritablement une réinvention de Smalltalk. Pour autant, les approches où l'on fait table rase du passé fonctionnent rarement. Au contraire, Pharo encourage les changements évolutifs et incrémentaux. Nous voulons qu'il soit possible d'expérimenter grâce à de nouvelles fonctionnalités et bibliothèques. Par évolution, nous disons que Pharo tolère les erreurs et n'a pas pour objectif de devenir la prochaine solution de rêve d'un bond — même si nous le désirons. Pharo favorisera toutes les évolutions à caractère incrémental. Le succès de Pharo dépend des contributions de sa communauté.

Qui devrait lire ce cours ?

Ce cours est dérivé de l'ouvrage « Squeak par l'exemple (4) », une introduction à Squeak éditée sous licence libre. Il a néanmoins été librement adapté pour refléter les différences qui existent entre Pharo et Squeak. Il présente différents aspects de Pharo, en commençant par les concepts de base et en poursuivant vers des sujets plus avancés. Ce cours ne vous apprendra pas à programmer. Le lecteur doit avoir quelques notions concernant les langages de programmation. Quelques connaissances sur la programmation orientée objet seront utiles. Ce cours introduit l'environnement de programmation, le langage et les outils de Pharo. Vous serez confronté à de nombreuses bonnes pratiques de Smalltalk, mais l'accent sera mis plus particulièrement sur les aspects techniques et non sur la conception orientée objet. Nous vous présenterons, autant que possible, un grand nombre d'exemples (nous avons été inspirés par l'excellent s de Alec Sharp sur Smalltalk (5)). Il y a plusieurs autres livres sur Smalltalk disponibles gratuitement sur le web, mais aucun d'entre eux ne se concentre sur Pharo. Voyez par exemple : <http://stephane.ducasse.free.fr/FreeBooks.html>.

Un petit conseil

Ne soyez pas frustré par des éléments de Smalltalk que vous ne comprenez pas immédiatement. Vous n'avez pas tout à connaître ! Alan Knight exprime ce principe comme suit (6) .



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

Un livre ouvert

Ce cours est ouvert dans plusieurs sens :

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

Exemples et exercices : nous utilisons deux conventions typographiques. Nous avons essayé de fournir autant d'exemples que possible. Il y en a notamment plusieurs avec des fragments de code qui peuvent être évalués. Nous utilisons le symbole \rightarrow afin d'indiquer le résultat qui peut être obtenu en sélectionnant l'expression et en utilisant l'option print it du menu contextuel : `3 + 4` \rightarrow 7 "Si vous sélectionnez 3+4 et 'print it', 7 s'affichera". Si vous voulez découvrir Pharo en vous amusant avec ces morceaux de code, sachez que vous pouvez charger un fichier texte avec la totalité des codes d'exemples sur le site web du livre : <http://PharoByExample.org/fr>. La deuxième convention que nous utilisons est l'icône




pour vous indiquer que vous avez quelque chose à faire :



Avancez et lisez le prochain chapitre

1 - Chapitre 1 - Une visite de Pharo

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

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

1-1 - Premiers pas

Pharo est librement disponible au téléchargement depuis la page <http://pharo-project.org/pharo-download> du site web de Pharo. Pour bien démarrer, il vous faudra trois archives : une archive image (contenant deux fichiers, l'image proprement dite et le fichier de *changes*) disponible dans le paragraphe *Pharo 1.* image*, un fichier nommé *sources* disponible dans le paragraphe *Sources file* et enfin, un programme exécutable appelé *machine virtuelle* selon votre système d'exploitation dans le paragraphe *Virtual Machines*.

Dans le cas du présent cours, nous aurons seulement besoin de télécharger une archive unique contenant tout le nécessaire. Sachez que si vous avez déjà une autre version de Pharo qui fonctionne sur votre machine, la plupart des exemples d'introduction de ce cours fonctionneront, mais, en raison de subtils changements dans l'interface et les outils proposés dans une version actuelle de Pharo, nous vous recommandons le téléchargement du fichier « Pharo-1.1-OneClick » disponible ici <http://gforge.inria.fr/frs/download.php/27303/Pharo-1.1-OneClick.zip> : vous aurez alors une image en parfait accord avec le cours.



Depuis le site <http://gforge.inria.fr/frs/download.php/27303/Pharo-1.1-OneClick.zip>, téléchargez et décompressez l'archive Pharo sur votre ordinateur.

Le dossier résultant de la décompression de l'archive contient quatre fichiers importants : la machine virtuelle selon votre système d'exploitation, l'image, le fichier « sources » et le fichier « changes ». Si vous êtes utilisateur de Mac OS X, ne vous inquiétez pas de ne voir qu'un seul fichier ; il s'agit d'un exécutable en bundle que vous pouvez explorer en cliquant dessus avec le bouton droit et en choisissant l'option du menu contextuel dite « Afficher le contenu du paquet ».

Avec la machine virtuelle, vous devriez donc avoir quatre fichiers tels que nous pouvons le voir sur la figure 1.1. Ainsi Pharo se compose :

- D'une *machine virtuelle* (abrégée en VM pour virtual machine) : c'est la seule partie de l'environnement qui est particulière à chaque couple système d'exploitation et processeur. Des machines virtuelles précompilées sont disponibles pour la plupart des systèmes (Linux, Mac OS X, Win32). Dans la figure 1.1, vous pouvez voir que la machine virtuelle pour la plateforme Windows est appelée *pharo.exe*. En naviguant dans le répertoire Contents/Linux, les utilisateurs trouveront un fichier binaire nommé *squeakvm* (7) : il s'agit de la machine virtuelle qui est appelée grâce au script shell *pharo.sh* ;
- Du fichier *source* : il contient le code source du système Pharo. Ce fichier ne change pas très fréquemment. Dans la figure 1.1, il correspond au fichier *PharoV10.sources* ;
- De l'*image système* : il s'agit d'un cliché d'un système Pharo en fonctionnement, figé à un instant donné. Il est composé de deux fichiers : le premier nommé avec l'extension *.image* contient l'état de tous les objets du système dont les classes et les méthodes (qui sont aussi des objets). Le second avec l'extension *.changes* contient le journal de toutes les modifications apportées au code source du système (contenu dans le fichier source). Dans la figure 1.1, ces fichiers sont appelés *PBE.image* et *PBE.changes*.

Ces trois derniers fichiers résident discrètement dans le répertoire *Contents/Resources*. Pendant que vous travaillez avec Pharo, les fichiers *.image* et *.changes* sont modifiés ; si vous êtes amenés à utiliser d'autres images, vous devez donc vous assurer qu'ils sont accessibles en écriture et qu'ils sont toujours ensemble, c.-à-d. dans le même dossier. Ne tentez pas de les modifier avec un éditeur de texte, Pharo les utilise pour stocker vos objets de travail et vos changements dans le code source. Faire une copie de sauvegarde de vos images téléchargées et de vos fichiers changes est une bonne idée ; vous pourrez ainsi toujours démarrer à partir d'une image propre et y recharger votre code. Les fichiers source et l'exécutable de la VM peuvent être en lecture seule — il est donc possible de les partager entre plusieurs utilisateurs.

Lancement. Pour lancer Pharo :

- si vous êtes sous Windows, cliquez sur *pharo.exe* à la racine du répertoire PBE-1.0-OneClick.app. Le fichier *pharo.ini* contient diverses options de lancement telles que *ImageFile* permettant de pointer vers une image particulière. Veillez à ne pas toucher ou déplacer ce fichier ;
- si vous êtes sous Linux, vous pouvez au choix cliquer sur *pharo.sh* ou lancer depuis votre terminal la commande *pharo.sh* depuis la racine du répertoire **PBE-1.0-OneClick.app**. Si vous ouvrez le script *shell* avec un éditeur, vous verrez que la commande exécute la machine virtuelle avec l'image *PBE.image* du répertoire **Contents/Resources**.
- si vous êtes sous Mac OS X, cliquez sur le fichier *PBE-1.0-OneClick* (ou *PBE-1.0-OneClick.app* suivant votre configuration). En affichant le contenu du paquet, vous avez dû voir le fichier de propriétés *Info.plist* à la racine du répertoire **Contents**. C'est là que la magie opère. Si vous ouvrez ce dernier fichier avec le programme *Property List Editor*, vous verrez que notre application *PBE-1.0-OneClick* cache le lancement d'une machine nommée *Squeak VM Opt* sur le fichier *PBE.image*.



FIGURE 1.2 – Une image PBE fraîchement démarrée.

Ainsi, cette archive dite « *OneClick* » (c.-à-d. un *clic*) nous évite de faire un *glissé-déposé* de notre image *PBE.image* sur le programme exécutable de notre machine virtuelle ou d'écrire notre propre script de lancement : tout se passe en un clic de souris.

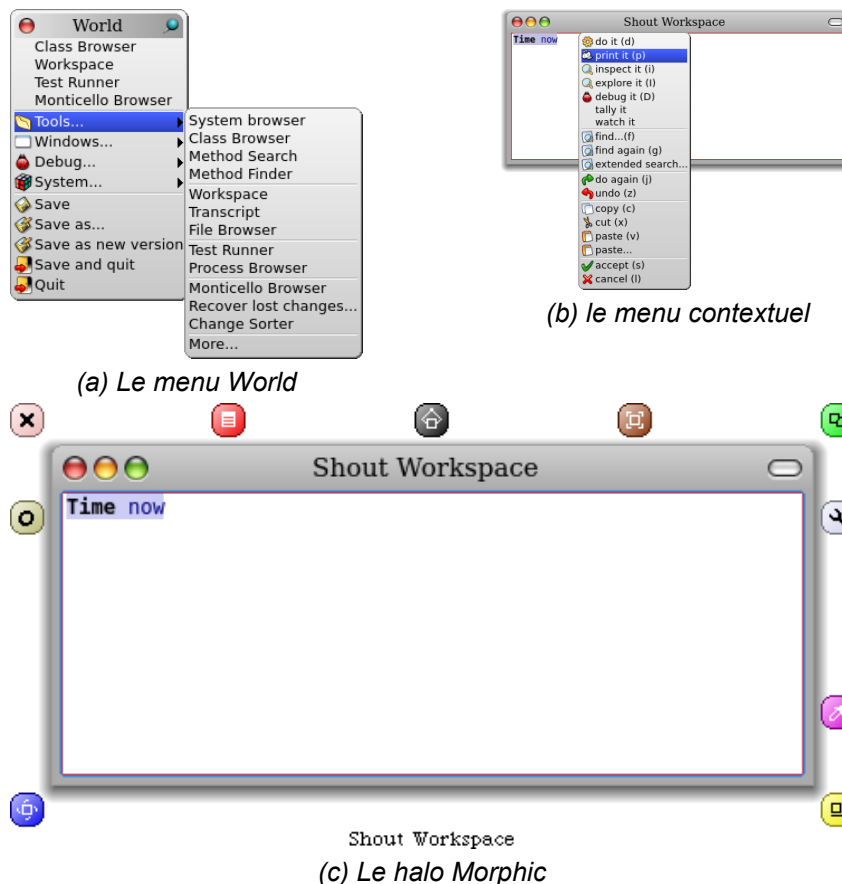
Une fois lancé, Pharo vous présente une large fenêtre qui peut contenir des espaces de travail nommés *Workspace* (voir la figure 1.2). Vous pourriez remarquer une barre de menus, mais Pharo emploie principalement des menus contextuels.



Lancez Pharo. Vous pouvez fermer les fenêtres déjà ouvertes en cliquant sur la bulle rouge dans le coin supérieur gauche des fenêtres.

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

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



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

Smalltalk a été conçu à l'origine pour être utilisé avec une souris à trois boutons. Si votre souris en a moins, vous pourrez utiliser des touches du clavier en complément de la souris pour simuler les boutons manquants. Une souris à deux boutons fonctionne bien avec Pharo, mais si la vôtre n'a qu'un seul bouton vous devriez songer à adopter un modèle récent avec une molette qui fera office de troisième bouton : votre travail avec Pharo n'en sera que plus agréable.

Pharo évite les termes « clic gauche » ou « clic droit », car leurs effets peuvent varier selon les systèmes, le matériel ou les réglages utilisateur. Originellement, Smalltalk introduit des couleurs pour définir les différents boutons de souris (8). Puisque de nombreux utilisateurs utiliseront diverses touches de modifications (*Ctrl*, *Alt*, *Meta* etc) pour réaliser les mêmes actions, nous utiliserons plutôt les termes suivants :

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

clic d'action : c'est le second bouton le plus utilisé ; il est utilisé pour afficher un menu contextuel c.-à-d. un menu qui fournit différentes actions dépendantes de la position de la souris comme le montre la figure 1.3 (b). Si vous n'avez pas de souris à multiples boutons, vous configurerez normalement la touche de modifications *Ctrl* pour effectuer cette même action avec votre unique bouton de souris ; nous utiliserons l'expression « cliquer avec le bouton d'action » (9) .

meta-clic : vous pouvez finalement *meta-cliquer* sur un objet affiché dans l'image pour activer le « halo Morhic » qui est une constellation d'icônes autour de l'objet actif à l'écran ; chaque icône représentant une poignée de contrôle permettant des actions telles que *changer la taille* ou *faire pivoter l'objet*, comme vous pouvez le voir sur la figure 1.3 (c) (10) . En survolant lentement une icône avec le pointeur de votre souris, une bulle d'aide en affichera un descriptif

de sa fonction. Dans Pharo, *meta-cliquer* dépend de votre système d'exploitation : soit vous devez maintenir *SHIFT Ctrl* soit *SHIFT Option* tout en cliquant.

- 1 Saisissez **Time now** (expression retournant l'heure actuelle) dans le Workspace, puis cliquez avec le bouton d'action dans le *Workspace* et sélectionnez **print it** (en français, « imprimez-le ») dans le menu qui apparaît.

Nous recommandons aux droitiers de configurer leur souris pour cliquer avec le bouton gauche (qui devient donc le bouton de clic), cliquer avec le bouton d'action avec le bouton droit et meta-cliquer avec la molette de défilement cliquable, si elle est disponible. Si vous utilisez un Macintosh avec une souris à un bouton, vous pouvez simuler le second bouton en maintenant \mathbb{X} la touche enfoncée en cliquant. Cependant, si vous prévoyez d'utiliser Pharo souvent, nous vous recommandons d'investir dans un modèle à deux boutons au minimum.

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

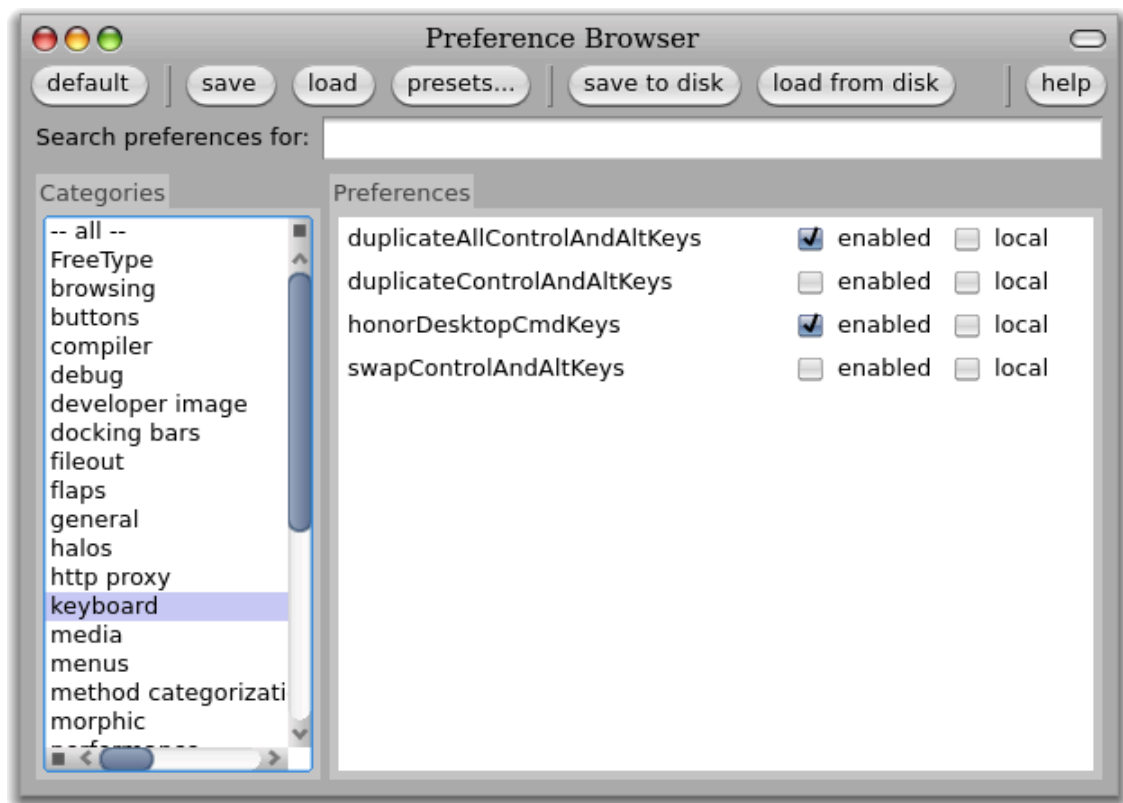


FIGURE 1.4 – Le Preference Browser.

1-2 - Le menu World

- 1 Cliquez dans l'arrière-plan de Pharo.

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

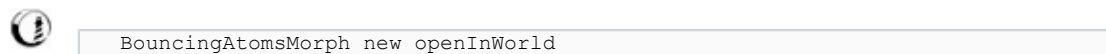
Le menu World vous offre un moyen simple d'accéder à la plupart des outils disponibles dans Pharo.

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

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

1-3 - Envoyer des messages

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



- Maintenant, cliquez avec le bouton d'action. Un menu devrait apparaître. Sélectionnez l'option **do it (d)** (en français, « faites-le ! ») comme le montre la figure 1.5.

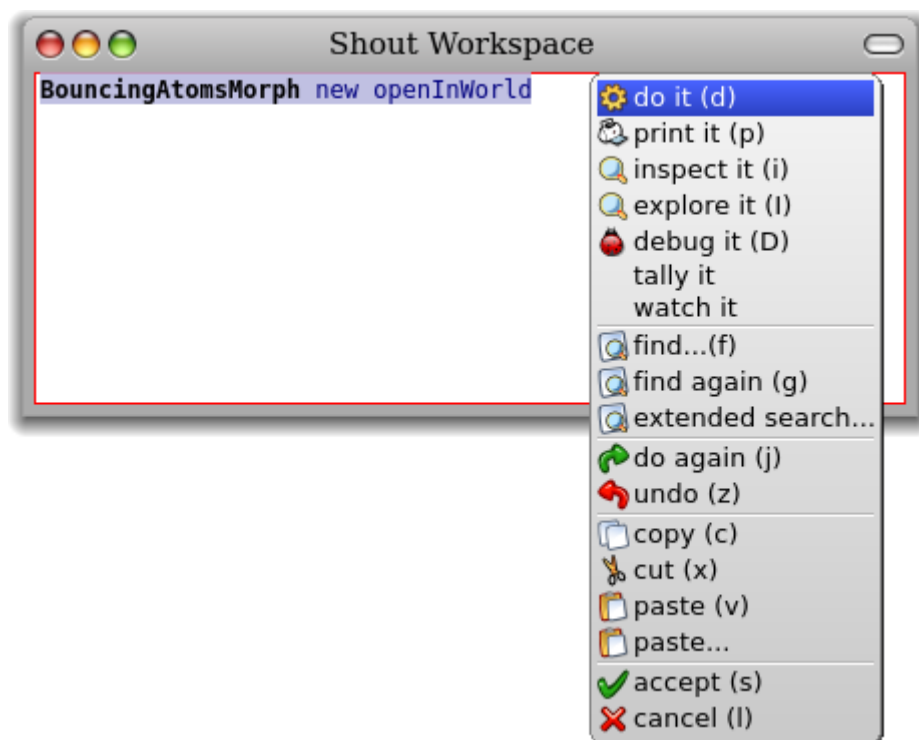


FIGURE 1.5 – Évaluer une expression avec **do it**.

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

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

Si vous discutez avec des habitués de Smalltalk, vous constaterez rapidement qu'ils n'emploient généralement pas les expressions comme « faire appel à une opération » ou « invoquer une méthode » : ils diront « envoyer un message ».

Ceci reflète l'idée que les objets sont responsables de leurs propres actions. Vous ne *direz* jamais à un objet quoi faire — vous lui *demanderez* poliment de faire quelque chose en lui envoyant un message. C'est l'objet, et non pas vous, qui choisit la méthode appropriée pour répondre à votre message.

1-4 - Enregistrer, quitter et redémarrer une session Pharo.



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

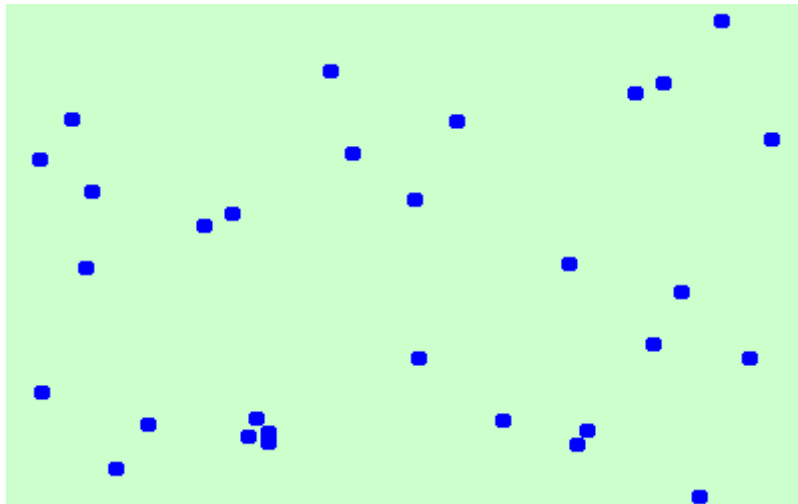


FIGURE 1.6 – Une instance de BouncingAtomsMorph.



Sélectionnez World ► Save and quit pour sauvegarder votre image et quitter Pharo.

Les fichiers « PBE.image » et « PBE.changes » contenus dans votre dossier Contents/Resources ont changé. Ces fichiers représentent l'image « vivante » de votre session Pharo au moment qui précédait votre enregistrement avec Save and quit. Ces deux fichiers peuvent être copiés à votre convenance dans les dossiers de votre disque pour y être utilisés plus tard : il faudra veiller à ce que le fichier « sources » soit présent et que l'exécutable de la machine virtuelle soit informé de la nouvelle localisation de notre image. Pour le cas du présent cours, il n'est pas souhaitable de toucher à ces fichiers ,mais si vous voulez en savoir plus sur la possibilité de préserver l'image actuelle et de changer d'image en utilisant l'option Save as..., rendez-vous dans la FAQ 4, p. 348.



Relancez Pharo en cliquant sur l'icône de votre programme (en fonction de votre système d'exploitation).

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


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

En quittant Pharo, vous sauvegardez un instantané de tous vos objets. En sauvegardant par World ► Save, vous remplacerez l'image courante par l'instantané de votre session comme nous l'avons fait grâce à Save and quit, mais sans quitter le programme.

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


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

Vous pourriez penser que l'utilisation d'une image est incontournable pour stocker et gérer des projets, mais comme nous le verrons bientôt il existe des outils plus adaptés pour gérer le code et travailler en équipe sur des projets. Les images sont très utiles, mais nous les considérons comme une pratique un peu dépassée et fragile pour diffuser et partager vos projets alors qu'il existe des outils tels que Monticello qui proposent de bien meilleurs moyens de suivre les évolutions du code et de le partager entre plusieurs développeurs.

 Meta-cliquez (en utilisant les touches de modifications appropriées conjointement avec votre souris) sur la fenêtre d'atomes rebondissants (11) .

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

1-5 - Les fenêtres Workspace et Transcript

 Fermez toutes fenêtres actuellement ouvertes. Ouvrez un Transcript (grâce au menu World ► Tools) et un Workspace. Positionnez et redimensionnez le Transcript et le Workspace pour que ce dernier recouvre le Transcript.

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

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

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


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

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

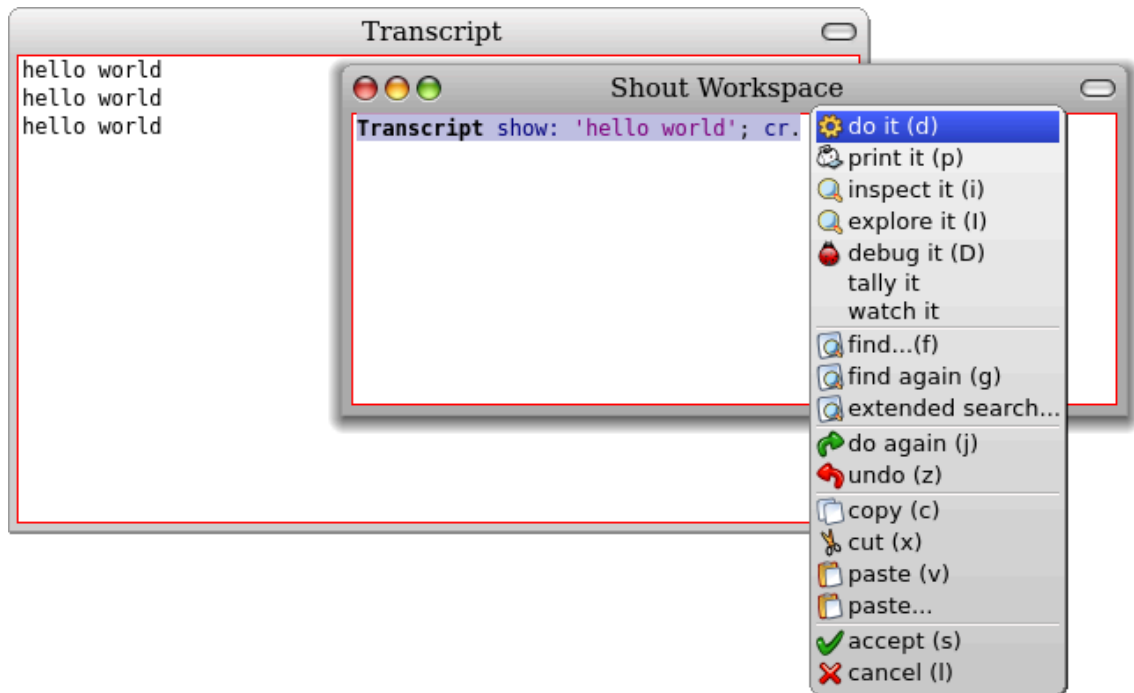


```
Transcript show: 'hello world'; cr.
```

Expérimentez la sélection en double-cliquant dans l'espace de travail à différents points dans le texte que vous venez de saisir. Remarquez comment un mot entier ou tout un texte est sélectionné selon que vous cliquez sur un mot, à la fin d'une chaîne de caractères ou à la fin d'une expression entière.

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

Notez que le texte « hello world » (12) apparaît dans la fenêtre Transcript (voir la figure 1.5). Refaites un `do it (d)` (Le (d) dans l'option de menu `do it (d)` vous indique que le raccourci-clavier correspondant est `CMD-d`. Pour plus d'informations, rendez-vous dans la prochaine section !).



Les fenêtres sont superposées. Le Workspace est actif.

1-6 - Les raccourcis-clavier

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

🔔 Réévaluez l'expression dans le Workspace en utilisant cette fois-ci le raccourci-clavier : `CMD-d`.

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

🔔 Entrez l'expression « `3 + 4` » dans le Workspace. Maintenant, évaluez-la en faisant un `do it` avec le raccourci-clavier.

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

🔔 Sélectionnez « `3 + 4` » et faites `print it (CMD-p)`.

Cette fois, nous pouvons lire le résultat que nous attendions (voir la figure 1.7).

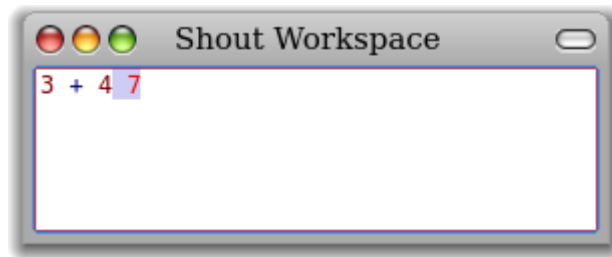


FIGURE 1.7 – Afficher le résultat sous forme de chaîne de caractères avec `print it` plutôt que de simplement évaluer avec `do it`.

3 + 4 → 7

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

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

Vous devriez maintenant voir une nouvelle fenêtre appelée inspecteur avec pour titre **SmallInteger: 7** (voir la figure 1.8). L'inspecteur ou (sous son nom de classe) `Inspector` est un outil extrêmement utile : il vous permet de naviguer et d'interagir avec n'importe quel objet du système. Le titre nous dit que 7 est une instance de la classe **SmallInteger** (classe des entiers sur 31 bits). Le panneau de gauche nous offre une vue des variables d'instance de l'objet en cours d'inspection. Nous pouvons naviguer entre ces variables, et le panneau de droite nous affiche leur valeur. Le panneau inférieur peut être utilisé pour écrire des expressions envoyant des messages à l'objet.

- Saisissez **self squared** dans le panneau inférieur de l'inspecteur que vous aviez ouvert sur l'entier 7 et faites un `print it`. Le message **squared** (carré) va élever le nombre 7 lui-même (*self*).
- Fermez l'inspecteur. Saisissez dans un Workspace le mot-expression `Object` et explorez-le grâce à `explore it` (CMD-I, i majuscule).

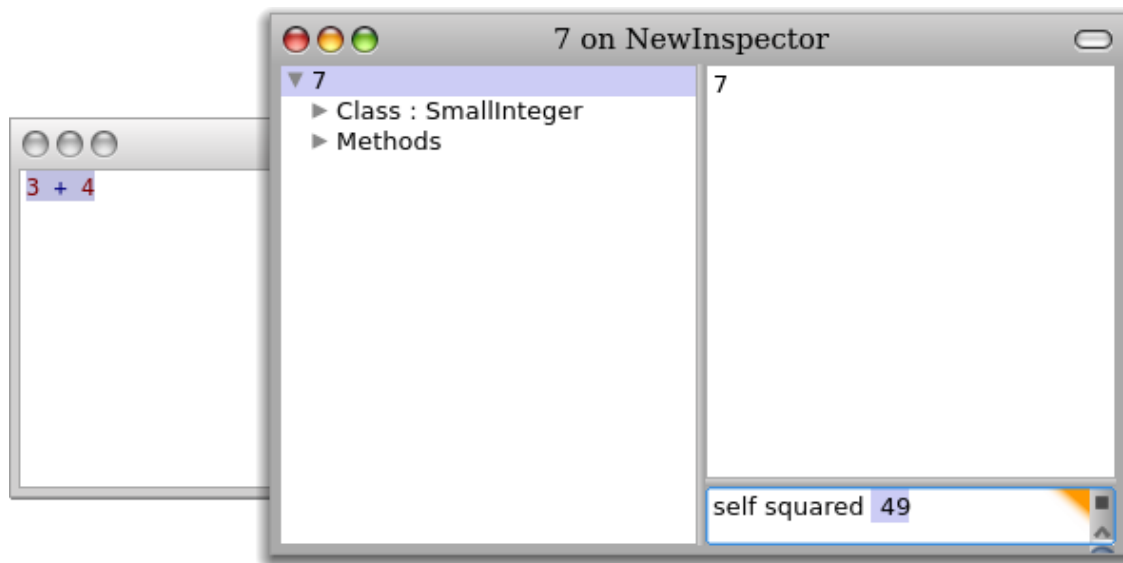


FIGURE 1.8 – Inspecter un objet.

Vous devriez voir maintenant une fenêtre intitulée **Object** contenant le texte « `> root: Object` ». Cliquez sur le triangle pour l'ouvrir (voir la figure 1.9).

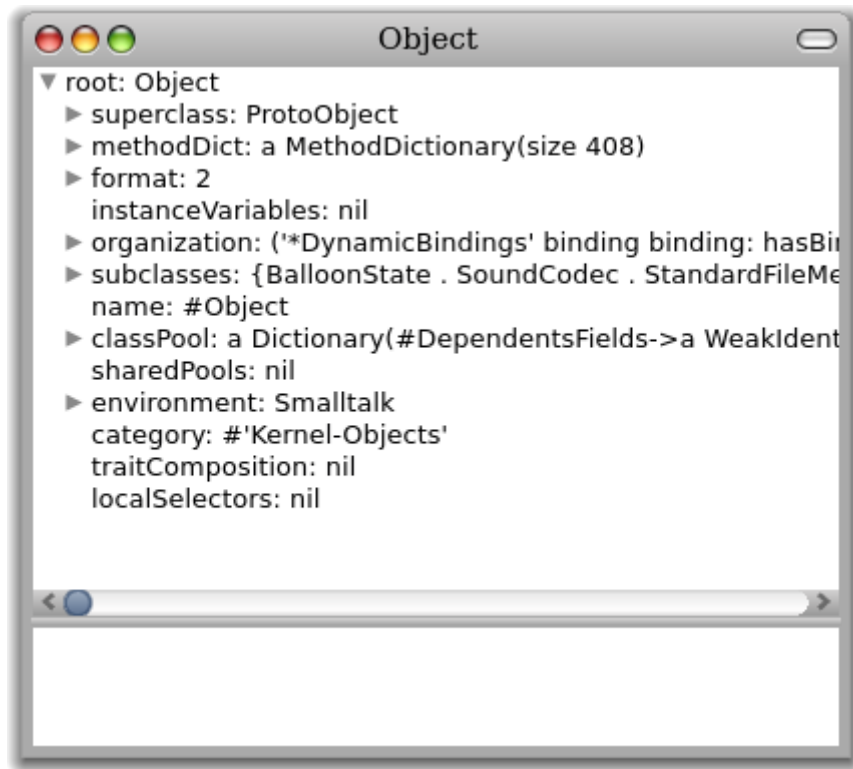


FIGURE 1.9 – Explorer Object.

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

1-7 - Le navigateur de classes Class Browser

Le navigateur de classes nommé Class Browser (13) est l'un des outils-clefs pour programmer. Comme nous le verrons bientôt, il y a plusieurs navigateurs ou *browsers* intéressants disponibles pour Pharo, mais c'est le plus simple que vous pourrez trouver dans n'importe quelle image, que nous allons utiliser ici.



Ouvrez un navigateur de classes en sélectionnant World ▸ Class Browser (14) .

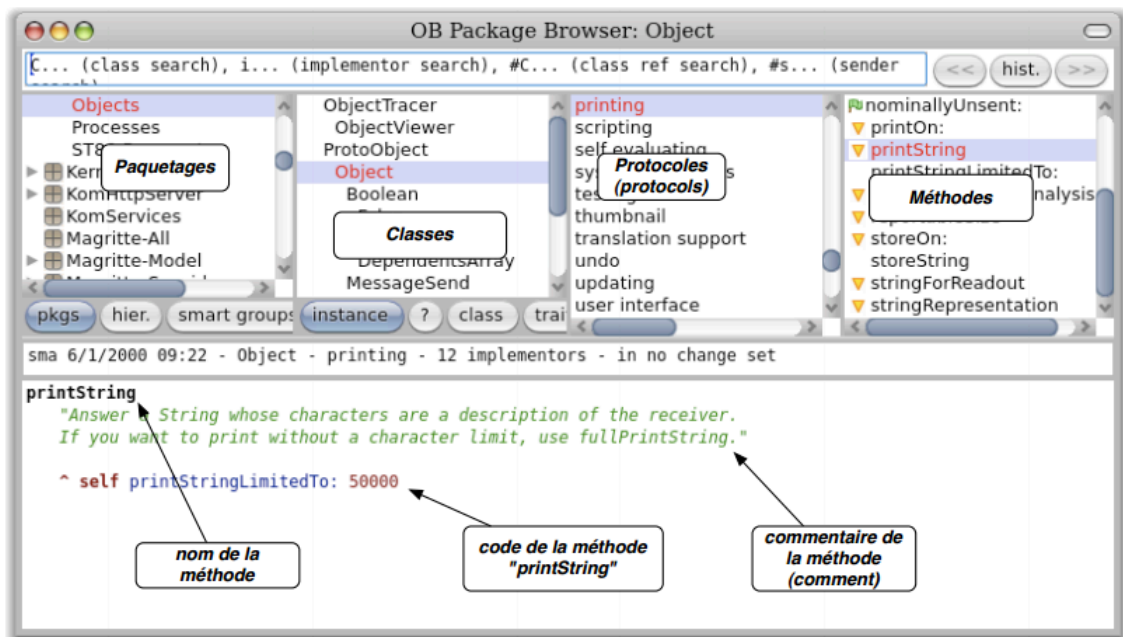


FIGURE 1.10 – Le navigateur de classes (ou Browser) affichant la méthode `printString` de la classe `Object`.

Nous pouvons voir un navigateur de classes sur la figure 1.10. La barre de titre indique que nous sommes en train de parcourir la classe **Object**.

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

1. cliquez sur le paquetage Kernel.

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

2. Sélectionnez la classe `Object`.

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

3. Sélectionnez le protocole `printing` , protocole de l'affichage.

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

4. Sélectionnez la méthode `printString`.

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

1-8 - Trouver des classes

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

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

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

Un navigateur s'ouvrira sur la classe **Boolean** (voir la figure 1.11). Il existe aussi un raccourci-clavier **CMD -b (browse)** que vous pouvez utiliser dans n'importe quel outil où vous trouvez un nom de classe ; sélectionnez le nom de la classe (par ex., **Boolean**) puis tapez **CMD -b**.

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

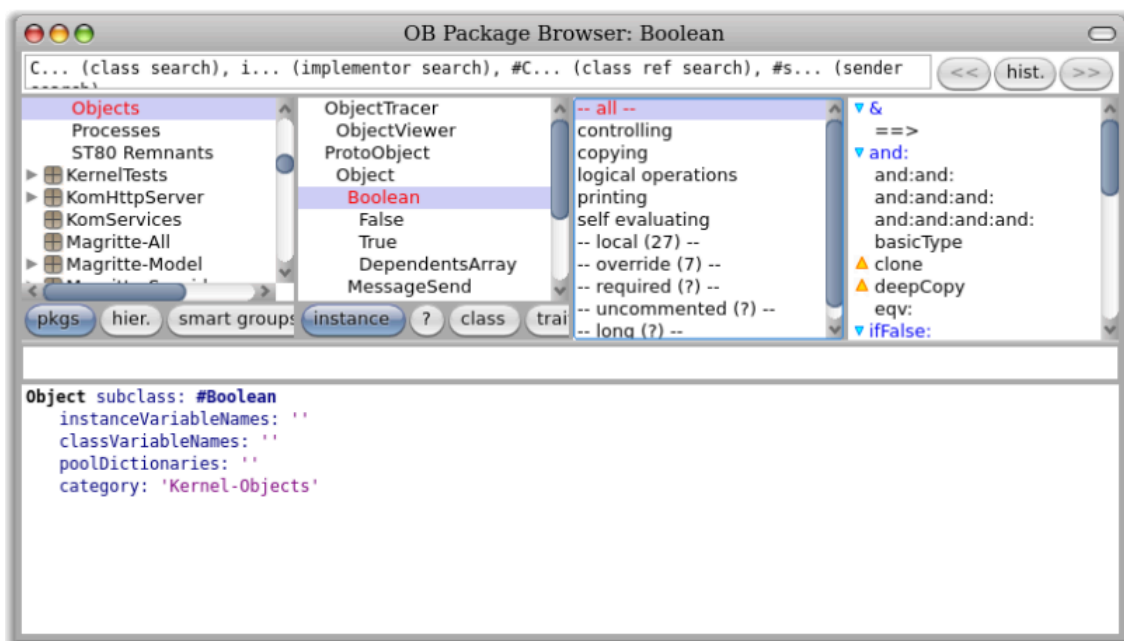


FIGURE 1.11 – Le navigateur de classes affichant la définition de la classe Boolean.

Remarquez que nous voyons une définition de classe quand la classe **Boolean** est sélectionnée, mais sans qu'aucun protocole ni aucune méthode ne le soit (voir la figure 1.11). Ce n'est rien de plus qu'un message Smalltalk ordinaire qui est envoyé à la classe parente lui réclamant de créer une sous-classe. Ici, nous voyons qu'il est demandé à la classe **Object** de créer une sous-classe nommée **Boolean** sans aucune variable d'instance, ni variable de classe ou « pool dictionaries » et de mettre la classe **Boolean** dans la catégorie *Kernel-Objects*. Si vous cliquez sur le bouton **?** en bas du panneau de classes, vous verrez le commentaire de classe dans un panneau dédié comme le montre la figure 1.12.

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

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

Une liste de classes dont le nom contient « **time** » vous sera présentée (voir la figure 1.13). Choisissez-en une, disons, **Time** ; un navigateur l'affichera avec un commentaire de classe suggérant d'autres classes pouvant être utiles. Si vous voulez naviguer dans l'une des autres classes, sélectionnez son nom (dans n'importe quelle zone de texte) et tapez **CMD-b**.

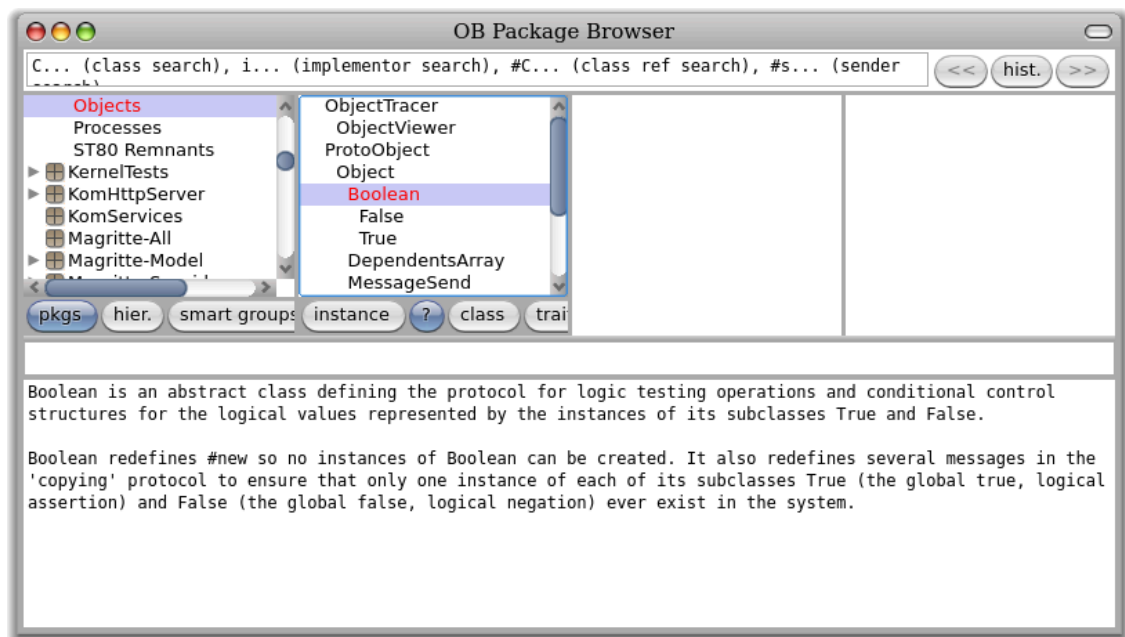
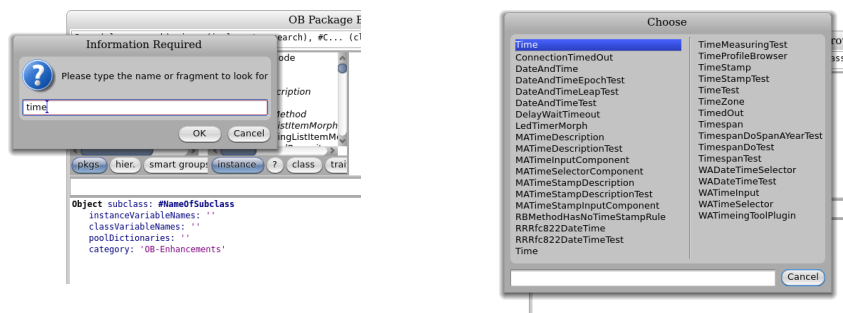


FIGURE 1.12 – Le commentaire de classe de Boolean.



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

1-9 - Trouver des méthodes

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



Sélectionnez World ► Tools ► Method Finder.

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

Pour défiler jusqu'à **now** lui-même, tapez « n » ; cette astuce fonctionne sur toutes les zones à défilement de n'importe quelle fenêtre. En sélectionnant « now », le panneau de droite vous présentera les classes qui définissent une méthode avec ce nom, comme le montre la figure 1.14. Sélectionner une de ces classes vous ouvrira un navigateur sur celle-ci.

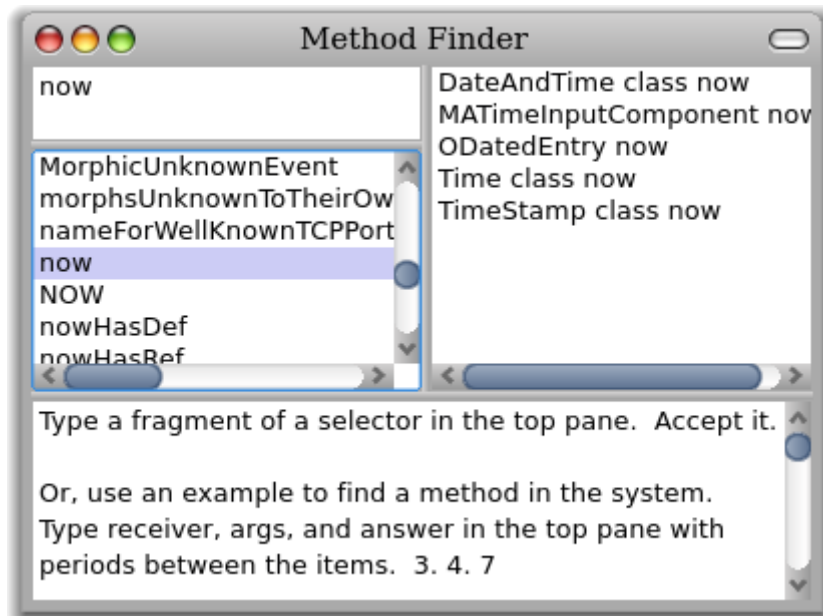


FIGURE 1.14 – Le Method Finder affichant toutes les classes qui définissent une méthode appelée now.

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



Saisissez '**eureka**' . '**EUREKA**' dans le Method Finder, comme le montre la figure 1.15.

Le Method Finder vous suggère une méthode qui fait ce que vous voulez (15) .

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

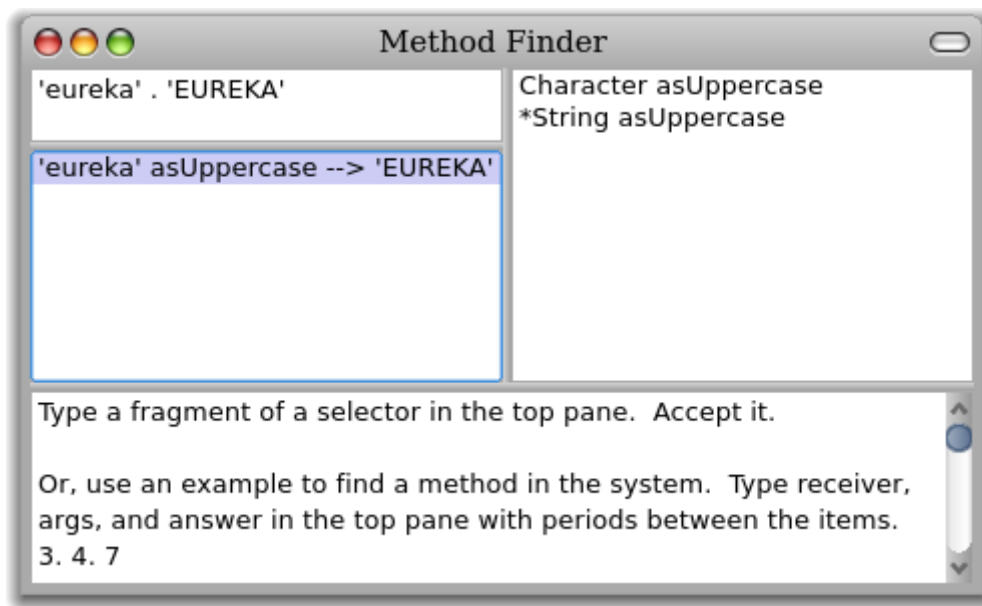


FIGURE 1.15 – Trouver une méthode par l'exemple.

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

1-10 - Définir une nouvelle méthode

L'avènement de la méthodologie de développement orientée tests ou Test Driven Development (16) a changé la façon d'écrire du code. L'idée derrière cette technique aussi appelée TDD se résume par l'écriture du test qui définit le comportement désiré de notre code avant celle du code proprement dit. À partir de là seulement, nous écrivons le code qui satisfait au test.

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

quand nous envoyons le message **shout** (qui veut dire « crier » en anglais) à la chaîne de caractères « Pas de panique », le résultat devrait être « PAS DE PANIQUE ! ».

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

Méthode 1.1 – Un test pour la méthode shout

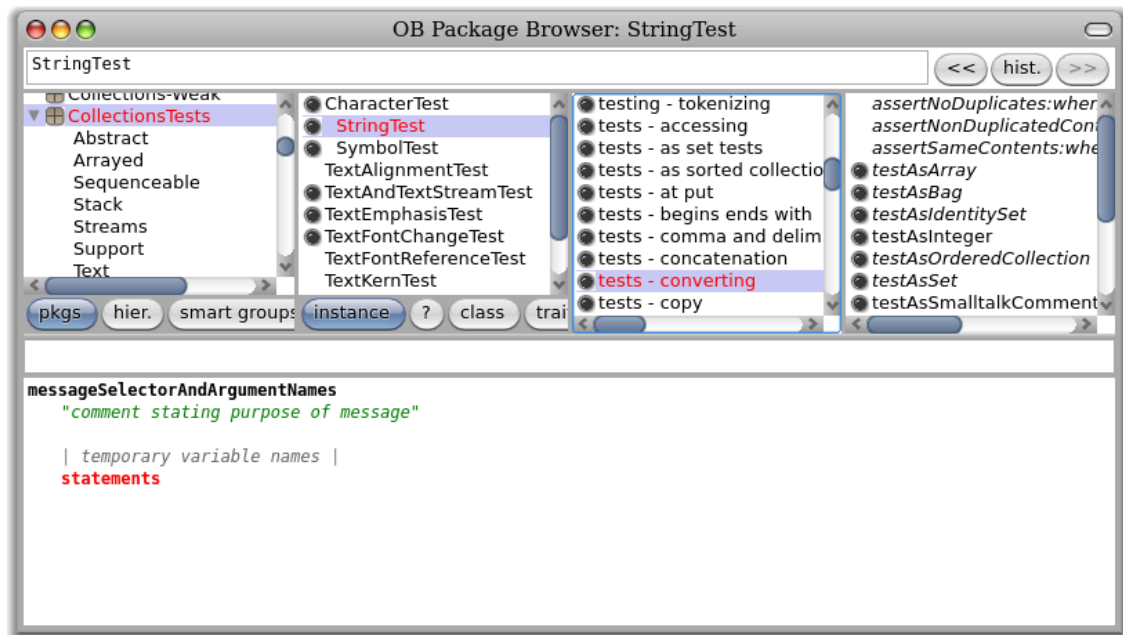
```
testShout
self assert: ('Pas de panique' shout = 'PAS DE PANIQUE!')
```

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

Ouvrez un navigateur de classes sur la classe **StringTest**. Sélectionnez un protocole approprié pour notre méthode ; dans notre cas, **tests - converting** (signifiant tests de conversion, puisque notre méthode modifiera le texte en retour), comme nous pouvons le voir sur la figure 1.16. Le texte surligné dans le panneau inférieur est un patron de méthode qui




vous rappelle ce à quoi ressemble une méthode. Effacez-le et saisissez le code de la méthode 1.1.




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

Si c'est la première fois que vous acceptez du code dans votre image, vous serez invité à saisir votre nom dans une fenêtre spécifique. Beaucoup de personnes ont contribué au code de l'image ; c'est important de garder une trace de tous ceux qui créent ou modifient les méthodes. Entrez simplement votre prénom suivi de votre nom sans espaces ni point de séparation.

Puisqu'il n'y a pas encore de méthode nommée shout, le Browser vous demandera confirmation que c'est bien le nom que vous désirez — il vous suggérera d'ailleurs d'autres noms de méthodes existantes dans le système (voir la figure 1.18). Ce comportement du navigateur est utile si vous aviez effectivement fait une erreur de frappe. Mais ici, nous voulons vraiment écrire shout puisque c'est la méthode que nous voulons créer. Dès lors, nous n'avons qu'à confirmer cela en sélectionnant la première option parmi celles du menu, comme vous le voyez sur la figure 1.18.

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

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

 Sélectionnez **CollectionsTests-Text** et le panneau juste à droite vous affichera alors toutes les classes de test de cette catégorie dont la classe **StringTest**.

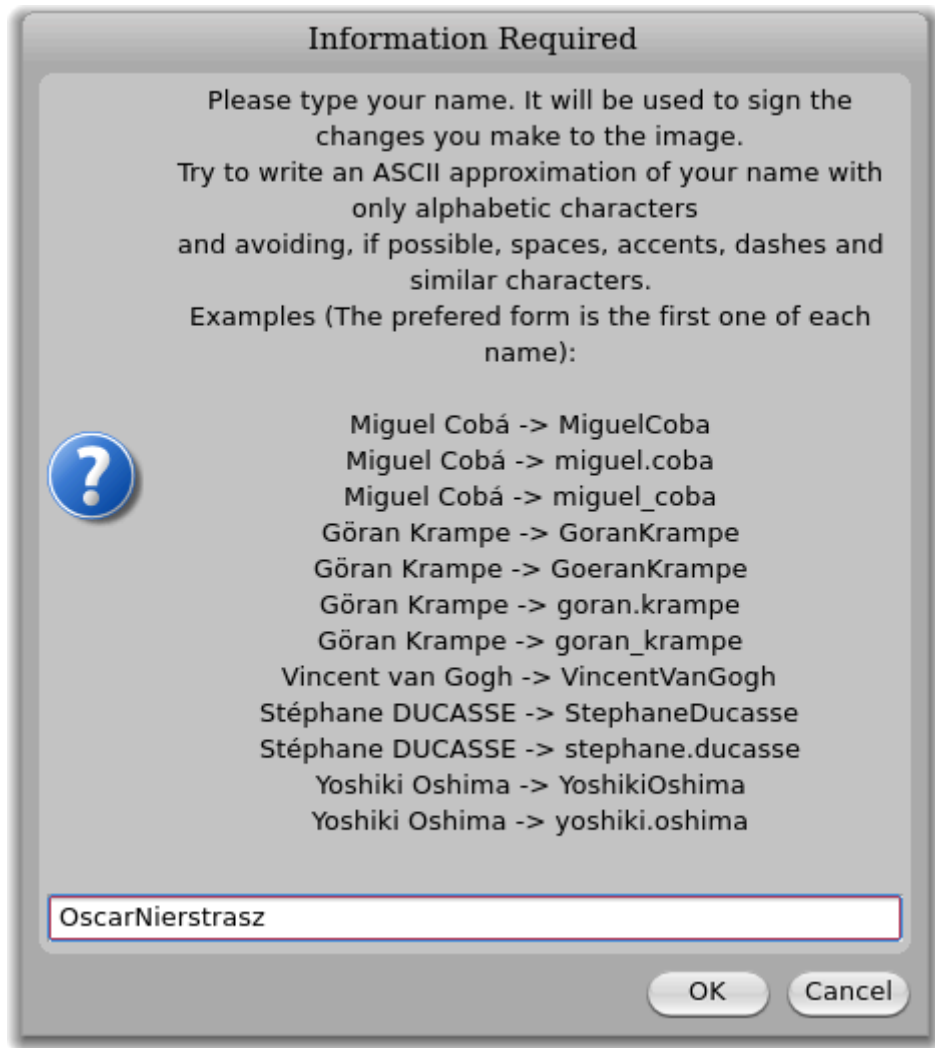


FIGURE 1.17 – Saisir son nom.

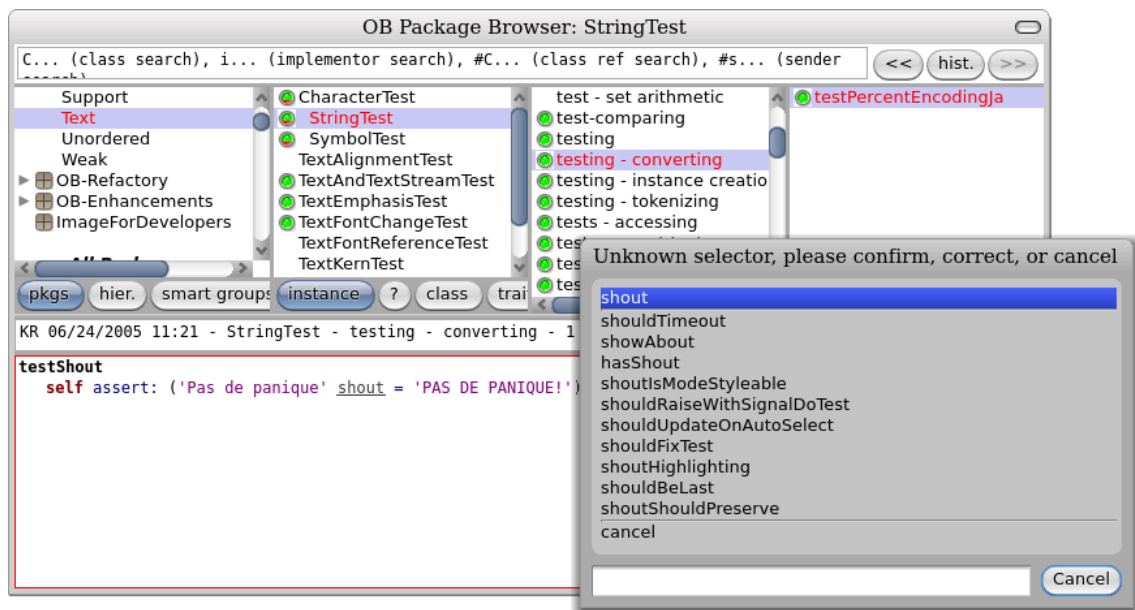


FIGURE 1.18 – Accepter la méthode testShout dans la classe StringTest.

Les classes sont déjà sélectionnées dans cette catégorie ; cliquez alors sur Run Selected pour lancer tous ces tests.

Vous devriez voir un message comme celui de la figure 1.19, vous indiquant qu'il y a eu une erreur lors de l'exécution des tests. La liste des tests qui donne naissance à une erreur est affichée dans le panneau inférieur de droite ; comme vous pouvez le voir, c'est bien **StringTest>>#testShout** le coupable (remarquez que la notation **StringTest>>#testShout** est la convention Smalltalk pour identifier la méthode de la classe **StringTest**). Si vous cliquez sur cette ligne de texte, le test erroné sera lancé à nouveau, mais cette fois-ci, de telle façon que vous voyez l'erreur surgir : « **MessageNotUnderstood: ByteString>>shout** ».

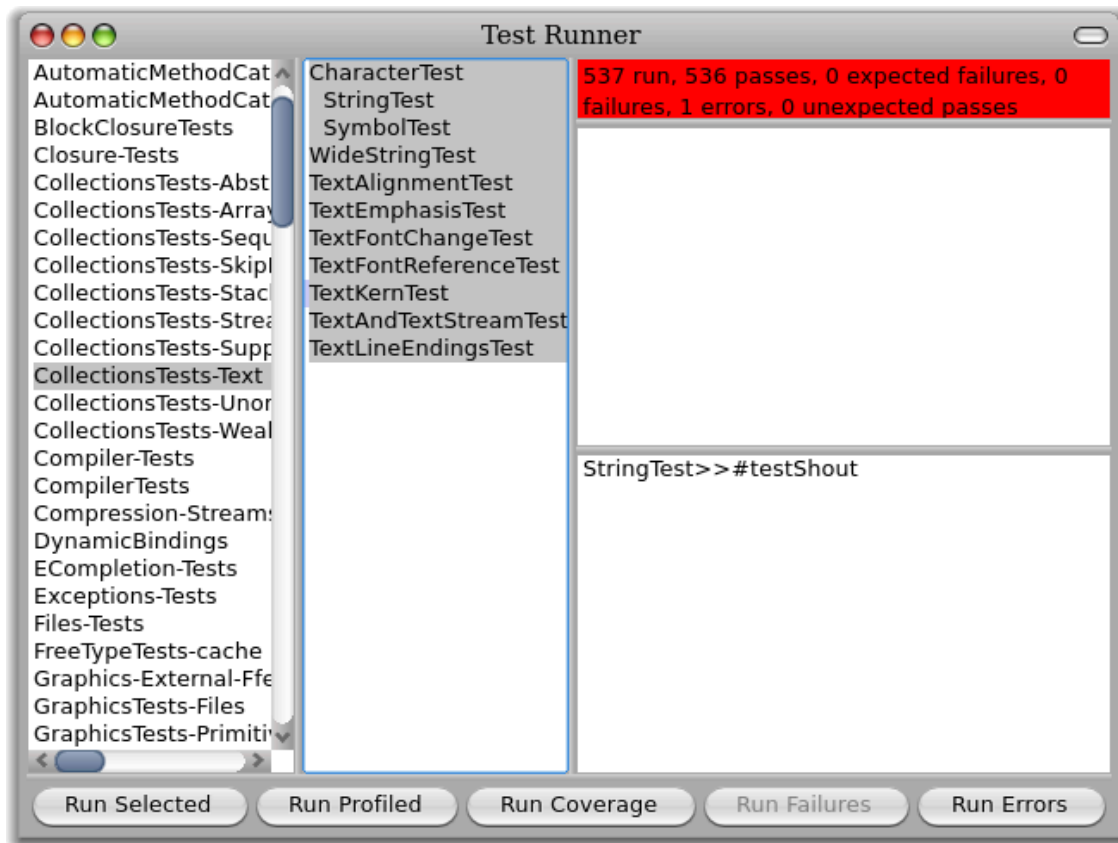


FIGURE 1.19 – Lancer les tests de String.

La fenêtre qui s'ouvre avec le message d'erreur est le débogueur Smalltalk (voir la figure 1.20). Nous verrons le débogueur nommé Debugger et ses fonctionnalités dans le chapitre 6.

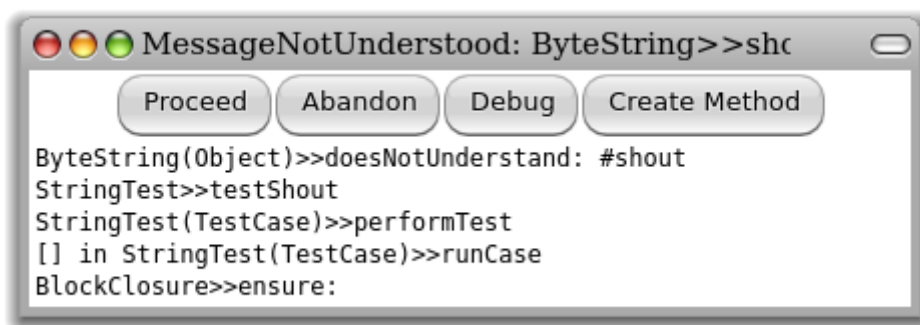


FIGURE 1.20 – La fenêtre de démarrage du débogueur.

L'erreur était bien sûr attendue ; lancer le test génère une erreur parce que nous n'avons pas encore écrit la méthode qui dit aux chaînes de caractères comment hurler c.-à-d. comment répondre au message shout. De toute façon, c'est une bonne pratique de s'assurer que le test échoue ; cela confirme que nous avons correctement configuré notre machine à tests et que le nouveau test est en cours d'exécution. Une fois que vous avez vu l'erreur, vous pouvez

cliquer sur le bouton **Abandon** pour abandonner le test en cours, ce qui fermera la fenêtre du débogueur. Sachez qu'en Smalltalk vous pouvez souvent définir la méthode manquante directement depuis le débogueur en utilisant le bouton **Create**, en y éditant la méthode nouvellement créée puis, in fine, en appuyant sur le bouton **Proceed** pour poursuivre le test. Définissons maintenant la méthode qui fera du test un succès !

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

Méthode 1.2 – La méthode shout

```
shout
↑ self asUppercase, '!'
```

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

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

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

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

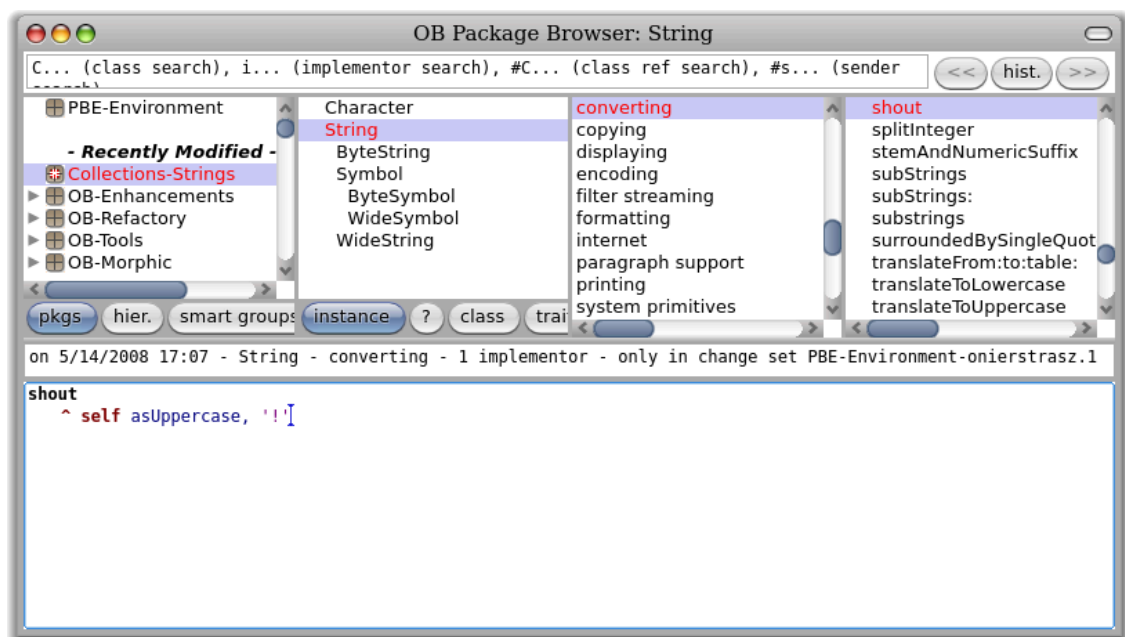


FIGURE 1.21 – La méthode shout dans la classe String.

1-11 - Résumé du chapitre

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

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

2 - Chapitre 2 - Une première application

Dans ce chapitre, nous allons développer un jeu simple de réflexion, le jeu Lights Out (17). En cours de route, nous allons présenter la plupart des outils que les développeurs Pharo utilisent pour construire et déboguer leurs programmes et voir comment les programmes sont échangés entre les développeurs. Nous verrons notamment le navigateur de classes, l'inspecteur d'objet, le débogueur et le navigateur de paquetages Monticello. Le développement avec Smalltalk est efficace : vous découvrirez que vous passerez beaucoup plus de temps à écrire du code et beaucoup moins à gérer le processus de développement. Ceci est en partie dû au fait que Smalltalk est un langage très simple et d'autre part que les outils qui forment l'environnement de programmation sont très intégrés avec le langage.

2-1 - Le jeu Lights Out

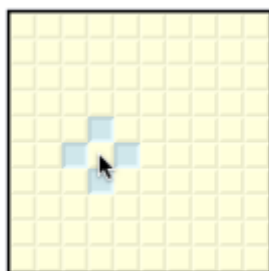


FIGURE 2.1 – Le plateau de jeu Lights Out. L'utilisateur vient de cliquer sur une case avec la souris comme le montre le curseur.

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

Le jeu Lights Out montré dans la figure 2.1 est fait de deux types d'objets : le plateau de jeu lui-même et une centaine de cellules-objets individuelles. Le code Pharo pour réaliser ce jeu va contenir deux classes : une pour le jeu et

une autre pour les cellules. Nous allons voir maintenant comment définir ces deux classes en utilisant les outils de programmation de Pharo.

2-2 - Créer un nouveau paquetage

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



Ouvrez un Browser et cliquez avec le bouton d'action sur le panneau des paquetages. Sélectionnez **create package** (18) .

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

2-3 - Définir la classe LOCell

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

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

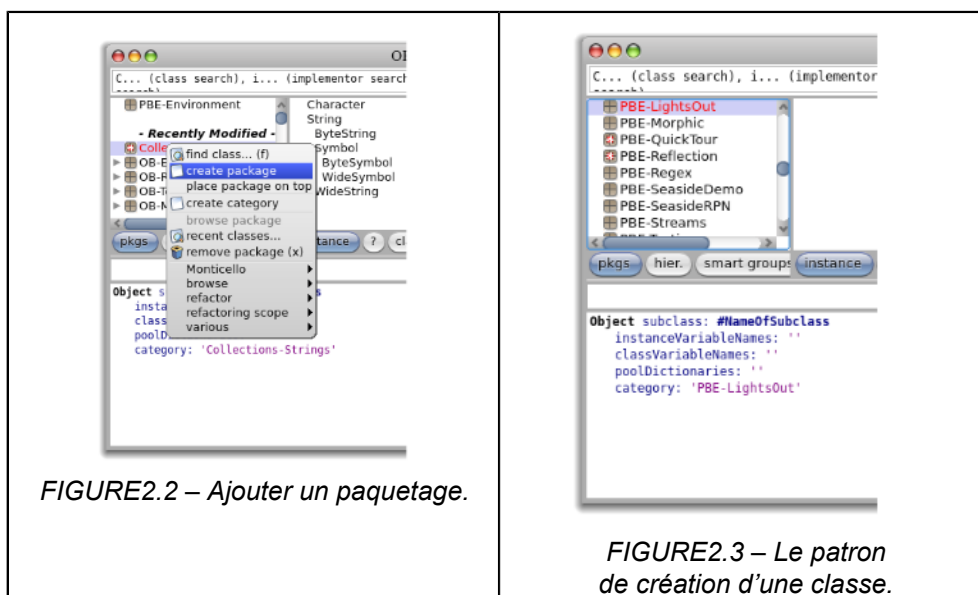


FIGURE2.2 – Ajouter un paquetage.

FIGURE2.3 – Le patron de création d'une classe.

2-3-1 - À propos des catégories et des paquetages

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

2-3-2 - Créer une nouvelle classe

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



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

- remplacez Object par SimpleSwitchMorph ;
- remplacez NameOfSubClass par LOCell ;
- ajoutez mouseAction dans la liste de variables d'instances.

Le résultat doit ressembler à la classe 2.1.

Classe 2.1 – Définition de la classe LOCell

```
1. SimpleSwitchMorph subclass: #LOCell
2.   instanceVariableNames: 'mouseAction'
3.   classVariableNames: ''
4.   poolDictionaries: ''
5.   category: 'PBE-LightsOut'
```

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

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

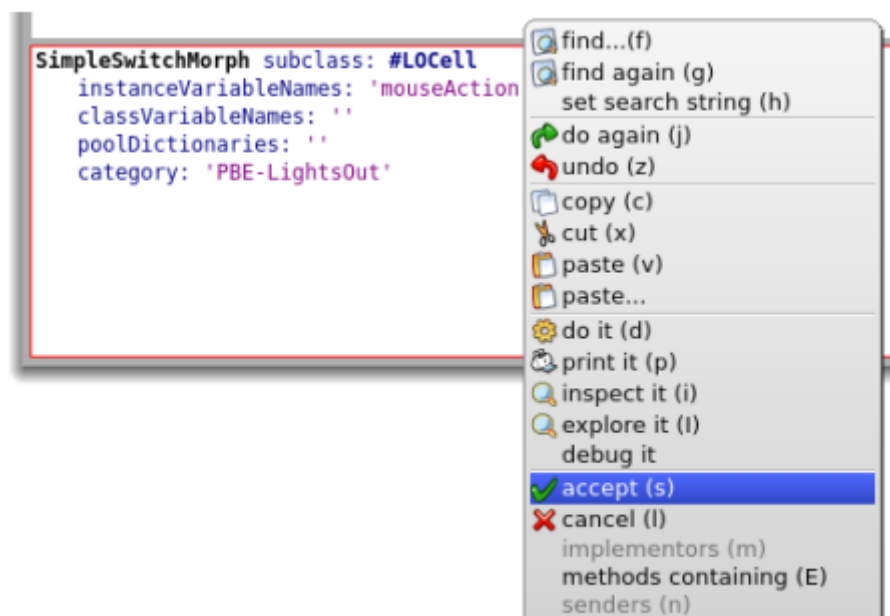


FIGURE 2.4 – Le modèle de création d'une classe.



Acceptez la nouvelle définition de classe.

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

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



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



Tapez un commentaire de classe pour LOCell et acceptez-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 protocoles.

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

Méthode 2.2 – Initialiser les instances de LOCell

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

Notez que les caractères `"` de la ligne 3 sont deux apostrophes (19) sans espace entre elles et non un guillemet (") ! `"` représente la chaîne de caractères vide.

 *Faites un accept de cette définition de méthode.*

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

Notons que la méthode s'appelle `initialize`. Ce nom dit bien ce qu'il veut dire (20) ! Par convention, si une classe définit une méthode nommée `initialize`, cette méthode sera appelée dès que l'objet aura été créé. Ainsi, dès que nous évaluons `LOCell new`, le message `initialize` sera envoyé automatiquement à cet objet nouvellement créé. Les méthodes d'initialisation sont utilisées pour définir l'état des objets, généralement pour donner une valeur à leurs variables d'instances ; c'est exactement ce que nous faisons ici.

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

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


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

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

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

2-5 - Inspecter un objet

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

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

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

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

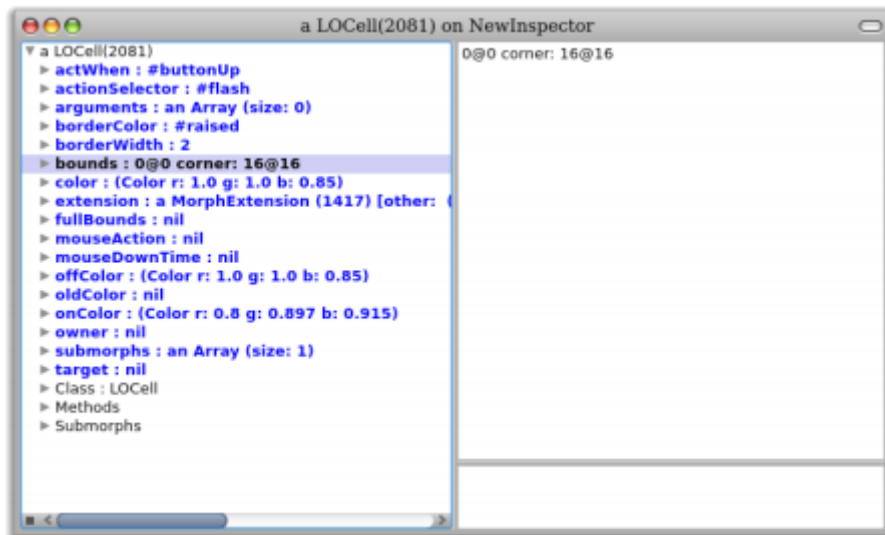



FIGURE 2.6 – L'inspecteur utilisé pour examiner l'objet LOCell.

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

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

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

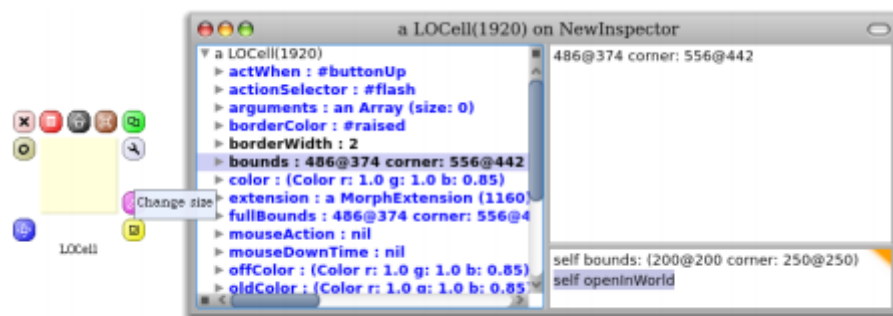




FIGURE 2.7 – Redimensionner la cellule.

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

2-6 - Définir la classe LOGame

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

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


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

Classe 2.3 – Définition de la classe LOGame

```
1. BorderedMorph subclass: #LOGame
2.   instanceVariableNames: ''
3.   classVariableNames: ''
4.   poolDictionaries: ''
5.   category: 'PBE-LightsOut'
```

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

Définissons maintenant une méthode `initialize` pour `LOGame`.


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

Méthode 2.4 – Initialisation du jeu

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

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

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

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

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

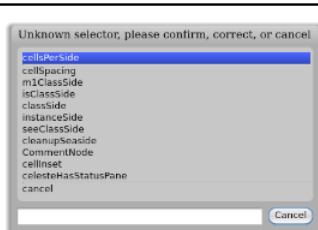


FIGURE 2.8 – Pharo détecte un sélecteur inconnu.

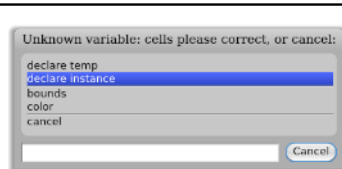



FIGURE 2.9 – Déclaration d'une nouvelle variable d'instance.

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

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

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

Examinons plus précisément cette méthode `initialize`. La ligne `| sampleCell width height n |` déclare quatre 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 quatre « variables » sont ici des constantes. Les lignes 4 à 7 définissent ces constantes.

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

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

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

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

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

`Matrix new: n tabulate: [:i:j] | self newCellAt: i at: j]` crée une nouvelle matrice de taille $n \times n$ et initialise ses éléments. La valeur initiale de chaque élément dépend de ses coordonnées. L'élément (i,j) sera initialisé avec le résultat de l'évaluation de `self newCellAt: i at: j`.

2-7 - Organiser les méthodes en protocoles

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

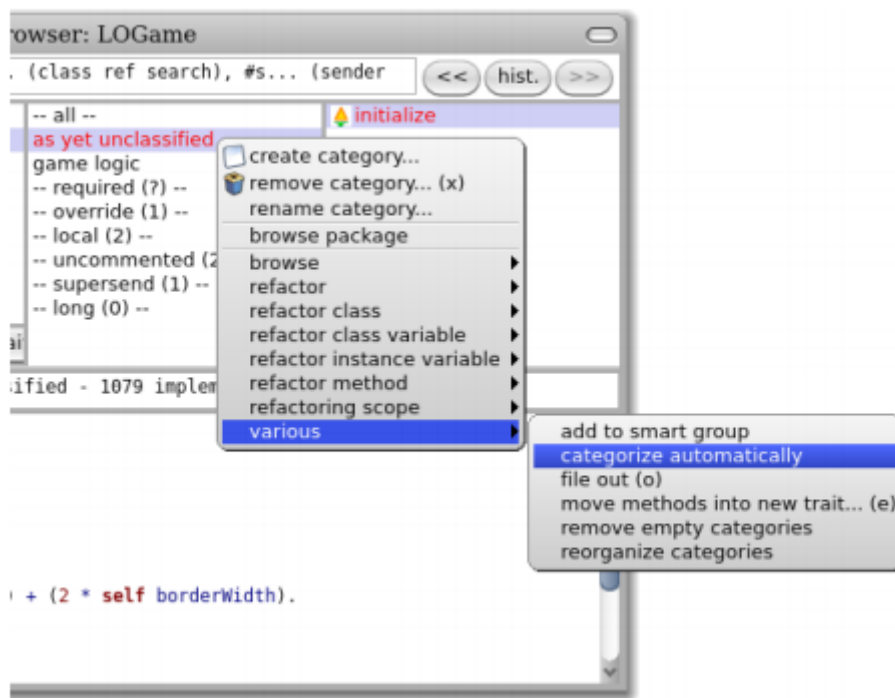



FIGURE 2.10 – Catégoriser de façon automatique toutes les méthodes non catégorisées.

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

Si vous avez suivi l'exemple jusqu'à présent, le troisième panneau doit contenir le protocole `as yet unclassified` (21).

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

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

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

À partir de maintenant, lorsque nous voudrions montrer une méthode dans ce livre, nous écrirons le nom de cette méthode sous cette forme. Bien sûr, lorsque vous tapez le code dans un navigateur, vous n'avez pas à taper le nom de la classe ou le » ; vous devrez juste vous assurer que la classe appropriée est sélectionnée dans le panneau des classes.

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

Méthode 2.5 – Une méthode constante

```
1. LOGame»cellsPerSide
2. "Le nombre de cellules le long de chaque côté du jeu"
3. ↑ 10
```

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

Méthode 2.6 – Une méthode auxiliaire pour l'initialisation

```
1. LOGame>newCellAt: i at: j
2. "Crée une cellule à la position (i,j) et l'ajoute dans ma représentation graphique à la
   position correcte. Retourne une nouvelle cellule"
3. | c origin |
4. c := LOCell new.
5. origin := self innerBounds origin.
6. self addMorph: c.
7. c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
8. c mouseAction: [self toggleNeighboursOfCellAt: i at: j]
```

Ajoutez les méthodes `LOGame>cellsPerSide` et `LOGame>newCellAt:at:`.



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

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

Méthode 2.7 – La méthode callback

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

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



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

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

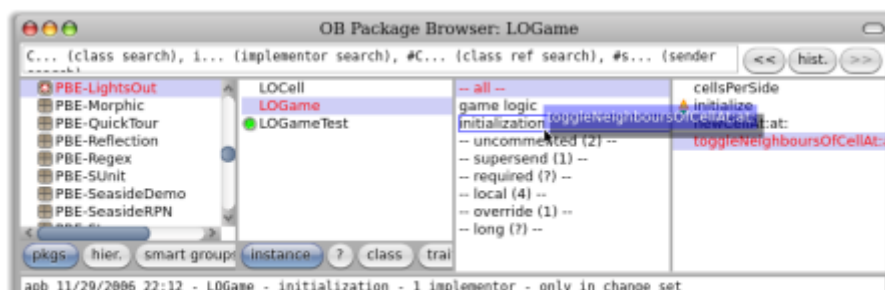


FIGURE 2.11 – Faire un glisser-déposer de la méthode dans un protocole.

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

Méthode 2.8 – Un mutateur typique

```
1. LOCell>>mouseAction: aBlock
2.   ↑ mouseAction := aBlock
```

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

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

Une méthode d'accès (en lecture ou en écriture) est appelée en anglais *accessor* et par convention, elle doit être placée dans le protocole *accessing*. En Smalltalk, *toutes* les variables d'instances sont privées à l'objet qui les possède, ainsi, la seule façon pour un autre objet de lire ou de modifier ces variables en Smalltalk se fait au travers de ces méthodes d'accès comme ici (22) .



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

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

Méthode 2.9 – Un gestionnaire d'événements

```
1. LOCell>>mouseUp: anEvent
2.   mouseAction value
```



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

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

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

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

2-8 - Essayons notre code

Voilà, le jeu *Lights Out* est complet !

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



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

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

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

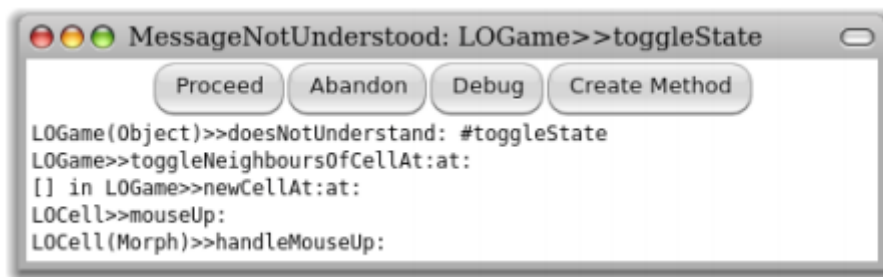




FIGURE 2.12 – Il y a une erreur dans notre jeu lorsqu'une cellule est sélectionnée !

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

 Cliquez sur le bouton debug de la fenêtre de notification.

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

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

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

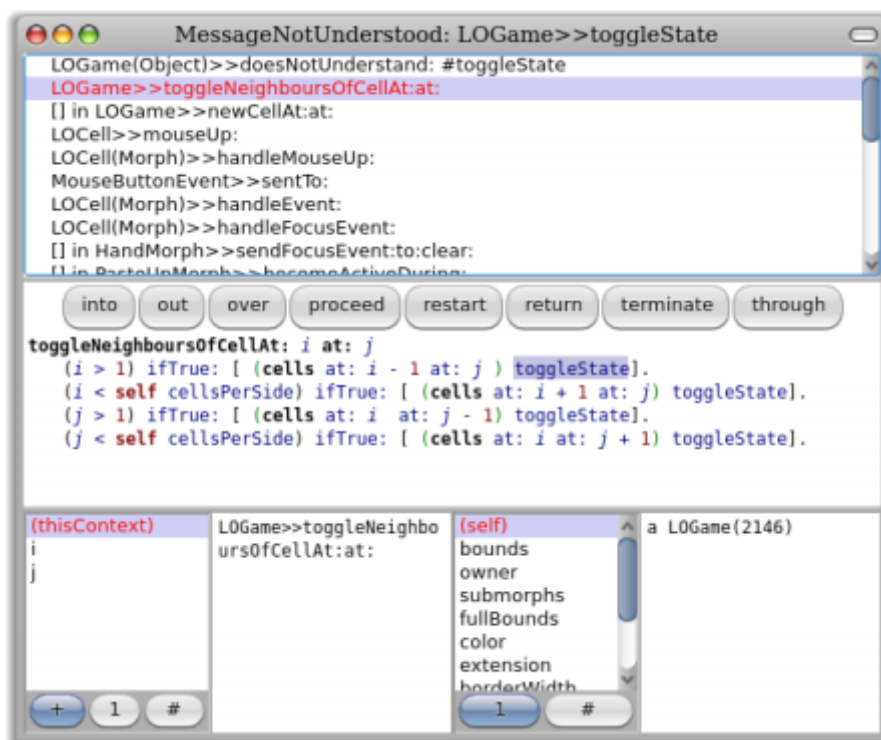


FIGURE 2.13 – Le débogueur avec la méthode toggleNeighboursOfCell:at: sélectionnée.

Dans la partie inférieure du débogueur, il y a deux petites fenêtres d'inspection. Sur la gauche, vous pouvez inspecter l'objet receveur du message qui cause l'exécution de la méthode sélectionnée. Vous pouvez voir ici les valeurs des variables d'instance. Sur la droite, vous pouvez inspecter l'objet qui représente la méthode en cours d'exécution. Il est possible d'examiner ici les valeurs des paramètres et les variables temporaires.

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

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



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

Méthode 2.10 – Corriger l'erreur

```
1. LOGame»newCellAt: i at: j
2. "Crée une cellule à la position (i,j) et l'ajoute dans ma représentation graphique à la
   position correcte. Retourne une nouvelle cellule"
3. | c origin |
4. c := LOCell new.
5. origin := self innerBounds origin.
6. self addMorph: c.
7. c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
8. c mouseAction: [self toggleNeighboursOfCellAt: i at: j].
9. ↑ c
```

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

Il est souvent possible de corriger le code directement dans la fenêtre du débogueur et de poursuivre l'application en cliquant sur `Proceed`. Dans notre cas, la chose la plus simple à faire est de fermer la fenêtre du débogueur, détruire l'instance en cours d'exécution (avec le halo `Morphic`) et d'en créer une nouvelle, parce que le bug ne se situe pas dans une méthode erronée, mais dans l'initialisation de l'objet.



Exécutez `LOGame new openInWorld` de nouveau.

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

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


```
LOGame»mouseMove: anEvent
```

Et voilà !


2-9 - Sauvegarder et partager le code Smalltalk

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

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

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

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

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

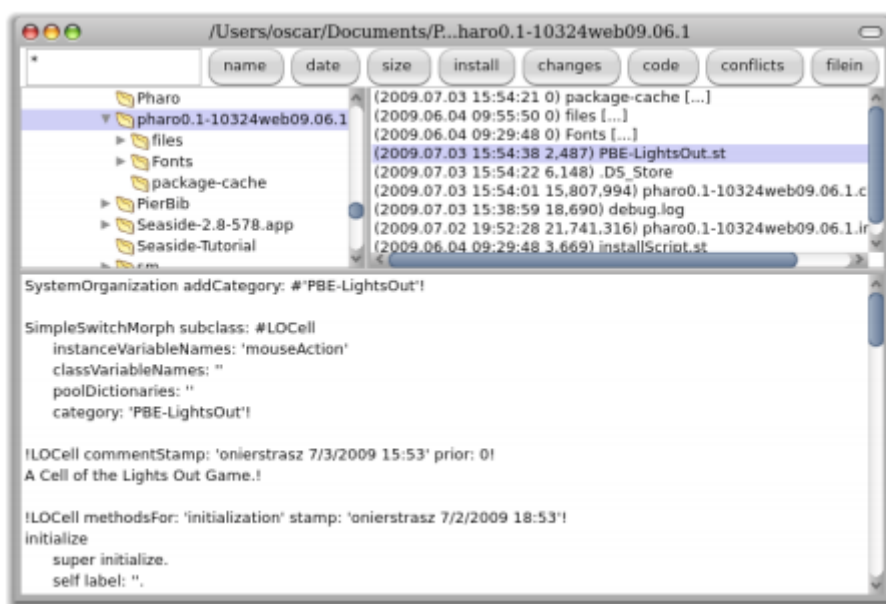


FIGURE 2.14 – Charger le code source dans Pharo.

2-9-1 - Les paquetages Monticello

Bien que les exportations de fichiers soient une façon convenable de faire des sauvegardes du code que vous avez écrit, elles font maintenant partie du passé. Tout comme la plupart des développeurs de projets libres *OpenSource* qui trouvent plus utile de maintenir leur code dans des dépôts en utilisant CVS (23) ou Subversion (24), les programmeurs sur Pharo gèrent maintenant leur code au moyen de paquetages Monticello (dit, en anglais, *packages*) : ces paquetages sont représentés comme des fichiers dont le nom se termine en *.mcz* ; ce sont en fait des fichiers compressés en *zip* qui contiennent le code complet de votre paquetage.

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

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



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

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

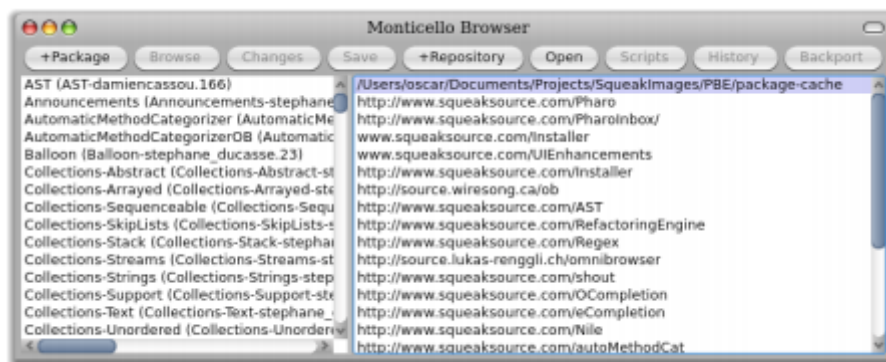


FIGURE 2.15 – Le navigateur Monticello.

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

2-9-2 - Sauvegarder et charger du code avec Monticello

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



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

2-9-3 - SqueakSource : un SourceForge pour Pharo

Nous pensons que la meilleure façon de sauvegarder votre code et de le partager est de créer un compte sur un serveur SqueakSource. SqueakSource est similaire à **SourceForge** (25) : il s'agit d'un *portail web* sur un serveur Monticello HTTP qui vous permet de gérer vos projets. Il y a un serveur public SqueakSource à l'adresse <http://www.squeaksource.com> et une copie du code concernant ce livre est enregistrée sur <http://www.squeaksource.com/PharoByExample.html>.

Vous pouvez consulter ce projet à l'aide d'un navigateur Internet, mais il est beaucoup plus productif de le faire depuis Pharo en utilisant l'outil *ad hoc*, le navigateur Monticello, qui vous permet de gérer vos paquetages.



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

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

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



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

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

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

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

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

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



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

Pour charger un paquetage dans votre image, vous devez d'abord sélectionner une version particulière. Vous pouvez faire cela dans le navigateur de dépôts *Repository Browser*, que vous pouvez ouvrir avec le bouton `Open` ou en cliquant avec le bouton d'action pour choisir `open repository` dans le menu contextuel. Une fois que vous avez sélectionné une version, vous pouvez la charger dans votre image.



Ouvrez le dépôt `PBE-LightsOut` que vous venez de sauvegarder.



FIGURE 2.16 – Parcourir un dépôt Monticello.

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

2-10 - Résumé du chapitre

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

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

3 - Chapitre 3 - Un résumé de la syntaxe

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

```
(Smalltalk hasClassName: 'Class') ifTrue: [ 'classe' ] ifFalse: [ 'pas classe' ]
```

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

3-1 - Les éléments syntaxiques

Les expressions sont composées des blocs constructeurs suivants :

- six mots-clefs réservés ou *pseudovariables* : `self`, `super`, `nil`, `true`, `false` et `thisContext` ;
- des expressions constantes pour des *objets littéraux* comprenant les nombres, les caractères, les chaînes de caractères, les symboles et les tableaux ;
- des déclarations de variables ;
- des affectations ;
- des blocs ou fermetures lexicales – *block closures* en anglais – et ;
- des messages.

expression	ce qu'elle représente
<code>startPoint</code>	un nom de variable
<code>Transcript</code>	un nom de variable globale
<code>self</code>	une pseudovariable
<code>1</code>	un entier décimal
<code>2r101</code>	un entier binaire
<code>1.5</code>	un nombre flottant
<code>2.4e7</code>	une notation exponentielle
<code>\$a</code>	le caractère 'a'
<code>'Bonjour'</code>	la chaîne « Bonjour »
<code>#Bonjour</code>	le symbole #Bonjour
<code>#{1 2 3}</code>	un tableau de littéraux
<code>{1. 2. 1+2}</code>	un tableau dynamique
<code>"c'est mon commentaire"</code>	un commentaire
<code> x y </code>	une déclaration de 2 variables x et y
<code>x := 1</code>	l'affectation de la valeur 1 à la variable x
<code>[x + y]</code>	un bloc qui évalue x+y
<code><primitive: 1></code>	une primitive de la VM (26) ou annotation
<code>3 factorial</code>	un message unaire
<code>3 + 4</code>	un message binaire
<code>2 raisedTo: 6 modulo: 10</code>	un message à mots-clefs
<code>↑ true</code>	Le renvoi de la valeur true pour vrai
<code>Transcript show: 'bonjour'. Transcript cr</code>	un séparateur d'expression (.)
<code>Transcript show: 'bonjour'; cr</code>	un message en cascade (;)

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

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

Les variables partagées. Les identifiants qui débutent par une lettre majuscule sont des variables globales, des variables de classes, des dictionnaires de pool ou des noms de classes. `Transcript` est une variable globale, une instance de la classe `TranscriptStream`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3-2 - Les pseudovariables

Dans Smalltalk, il y a 6 mots-clefs réservés ou *pseudovariables* : nil, true, false, self, super et thisContext. Ils sont appelés pseudovariables car ils sont prédéfinis et ne peuvent pas être l'objet d'une affectation. true, false et nil sont des constantes tandis que les valeurs de self, super et de thisContext varient de façon dynamique lorsque le code est exécuté.

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

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

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

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

3-3 - Les envois de messages

Il y a trois types de messages dans Pharo.

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

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

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

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

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

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

```
2 raisedTo: 1 + 3 factorial -> 128
```

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

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

```
1 + 2 * 3 -> 9
```

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

```
1 + (2 * 3) -> 7
```

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

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

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

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

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

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

3-4 - Syntaxe relative aux méthodes

Bien que les expressions puissent être évaluées n'importe où dans Pharo (par exemple, dans un espace de travail [Workspace], dans un débogueur [Debugger] ou dans un navigateur de classes), les méthodes sont en principe définies dans une fenêtre du Browser ou du débogueur ; les méthodes peuvent aussi être rentrées depuis une source externe, mais ce n'est pas une façon habituelle de programmer en Pharo).

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

Voilà la méthode `lineCount` (pour compter le nombre de lignes) dans la classe `String`. La convention habituelle consiste à se référer aux méthodes comme suit : `ClassName»methodName` ; ainsi nous nommerons cette méthode `String»lineCount` (28) .

Méthode 3.1 – Compteur de lignes

```
1. String»lineCount
2. "Answer the number of lines represented by the receiver, where ever y cr adds one line."
3. | cr count |
```

Méthode 3.1 – Compteur de lignes

```
4. cr := Character cr.
5. count := 1 min: self size.
6. self do:
7.   [:c | c == cr ifTrue: [count := count + 1]].
8. ↑ count
```

Sur le plan de la syntaxe, une méthode comporte :

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

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

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

3-5 - La syntaxe des blocs

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

```
[ 1 + 2 ] value -> 3
```

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

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

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

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

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

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

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

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

3-6 - Conditions et itérations

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

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

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

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

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

`whileFalse:` inverse la condition de sortie.

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

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

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

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

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

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

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

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

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


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

`select:` et `reject:` construisent des collections nouvelles, contenant un sous-ensemble d'éléments satisfaisant (ou non) la condition du bloc booléen. `detect:` retourne le premier élément satisfaisant la condition. Ne perdez pas de vue que les chaînes sont aussi des collections, ainsi, vous pouvez itérer aussi sur tous les caractères. La méthode `isVowel` renvoie `true` (c.-à-d. vrai) lorsque le receveur-caractère est une voyelle (29).

```
'Bonjour Pharo' select: [ :char | char isVowel ] -> 'ouao'
'Bonjour Pharo' reject: [ :char | char isVowel ] -> 'Bnjr Phr'
'Bonjour Pharo' detect: [ :char | char isVowel ] -> $o
```

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

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

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

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

3-7 - Primitives et Pragmas

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

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

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

Examinons le code pour `SmallInteger>+`. Si la primitive échoue, l'expression `super + aNumber` sera évaluée et renvoyée (30).

Méthode 3.2 – Une méthode primitive

```
1. + aNumber
2. "Primitive. Add the receiver to the argument and answer with the result
3. if it is a SmallInteger. Fail if the argument or the result is not a
4. SmallInteger Essential No Lookup. See Object documentation whatIsAPr imitive."
5.
6. <primitive: 1>
7. ↑ super + aNumber
```

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

3-8 - Résumé du chapitre

- Pharo a (seulement) six mots réservés aussi appelés pseudovariables : `true`, `false`, `nil`, `self`, `super` et `thisContext`.
- Il y a cinq types d'objets littéraux : les nombres (5, 2.5, 1.9e15, 2r111), les caractères (\$a), les chaînes ('bonjour'), les symboles (#bonjour) et les tableaux (#('bonjour' #bonjour)).

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

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

4 - Chapitre 4 - Comprendre la syntaxe des messages

Bien que la syntaxe des messages Smalltalk soit extrêmement simple, elle n'est pas habituelle pour un développeur qui viendrait du monde C/Java. Un certain temps d'adaptation peut être nécessaire. L'objectif de ce chapitre est de donner quelques conseils pour vous aider à mieux appréhender la syntaxe particulière des envois de messages. Si vous vous sentez suffisamment 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

Avec Smalltalk, exception faite des éléments syntaxiques rencontrés dans le chapitre 3 (`:=`, `↑`, `.`, `#`, `()`, `{}`, `:`, `|`), tout se passe par envoi de messages. Comme en C++, vous pouvez définir vos opérateurs comme `+` ou 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 il n'est pas possible d'avoir une forme unaire avec une surcharge différente.

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

Comme la plupart des calculs en Smalltalk sont effectués par des envois de messages, identifier correctement les messages est crucial. La 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 appelée un *envoi de message* comme il est montré dans la figure 4.1.

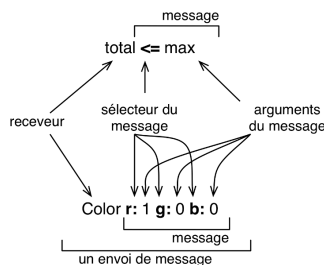


FIGURE 4.1 – Deux messages composés d'un receveur, d'un sélecteur de méthode et d'un ensemble d'arguments.



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

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

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

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

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

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

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

4-2 - Trois sortes de messages

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

- *Les messages unaires* sont des messages qui sont envoyés à un objet sans autre information. Par exemple dans `3 factorial`, `factorial` (pour factorielle) est un message unaire.
- *Les messages binaires* sont des messages formés avec des opérateurs (souvent arithmétiques). Ils sont binaires car ils ne concernent que deux objets : le receveur et l'objet argument. Par exemple, dans `10 + 20`, `+` est un message binaire qui est envoyé au receveur `10` avec l'argument `20`.
- *Les messages à mots-clefs* sont des messages formés avec plusieurs mots-clefs, chacun d'entre eux se finissant par deux points (`:`) et prenant un paramètre. Par exemple, dans `anArray at: 1 put: 10`, le mot-clef `at:` prend un argument `1` et le mot-clef `put:` prend l'argument `10`.

4-2-1 - Messages unaires

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

```
89 sin      -> 0.860069405812453
3 sqrt      -> 1.732050807568877
Float pi    -> 3.141592653589793
'blop' size -> 4
true not    -> false
Object class -> Object class "La classe de Object est Object class (!)"
```

Les messages unaires sont des messages qui ne nécessitent pas d'argument. Ils suivent le moule syntaxique : `receveur sélecteur`.

4-2-2 - Messages binaires

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

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

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

4-2-3 - Messages à mots-clefs

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

Chaque mot-clef utilise un argument. Ainsi `rgb:` 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 `rgb:` comme dans `Color r: 1 g: 0 b: 0` créant ainsi la couleur rouge. Notez que les deux points ne font pas partie du sélecteur.

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

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

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

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

4-3 - Composition de messages

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

- Les messages unaires sont envoyés en premier, puis les messages binaires et enfin les messages à mots-clefs.
- Les messages entre parenthèses sont envoyés avant tout autre type de messages.
- Les messages de même type sont envoyés de gauche à droite.

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

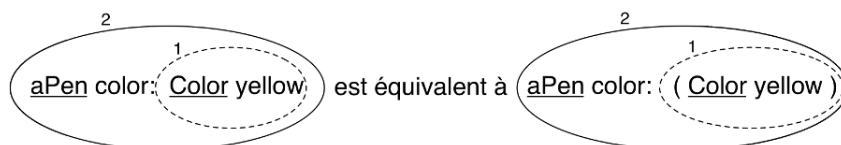


FIGURE 4.3 – Les messages unaires sont envoyés en premier ; donc ici le premier message est `Color yellow`. Il renvoie un objet de couleur jaune qui est passé comme argument du message `aPen color:`

4-3-1 - Unaire > Binaire > Mots-clefs

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

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

Unaire > Binaire > Mots-clefs

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

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

Malheureusement, les règles sont un peu trop simplistes pour les expressions arithmétiques. Dès lors, des parenthèses doivent être introduites chaque fois que l'on veut imposer un ordre de priorité entre deux opérateurs binaires :

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

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

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

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

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

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

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

Exemple. Dans le message `aPen go: 100 + 20`, il y a le message *binaire* `+ 20` et un message à *mots-clefs* `go:`. Les messages binaires sont d'abord envoyés avant les messages à mots-clefs, ainsi `100 + 20` est envoyé en premier (1) : le message `+ 20` est envoyé à l'objet `100` et renvoie le nombre `120`. Ensuite le message `aPen go: 120` est envoyé avec comme argument `120(2)`. L'exemple 4.2 nous montre comment l'expression est évaluée.

Exemple 4.2 – Décomposition de `aPen go: 100 + 20`

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

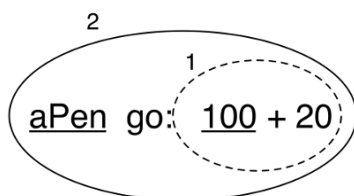


FIGURE 4.4 – Les messages unaires sont envoyés en premier, ainsi `Color yellow` est d'abord envoyé. Il retourne un objet de couleur jaune qui est passé en argument du message `aPen color:`.

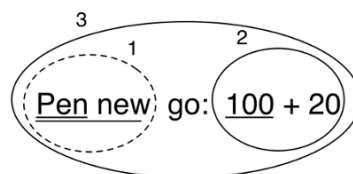


FIGURE 4.5 – Décomposition de `Pen new go: 100 + 20`.

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

4-3-2 - Les parenthèses en premier

Règle deux. Les messages encadrés de parenthèses sont envoyés avant tout autre message.

(Msg) > Unaire > Binaire > Mots-clefs

```
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-clefs lowMajorScaleOn:.
(3) joue le son résultant."
```

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

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

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

4-3-3 - De gauche à droite

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

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

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

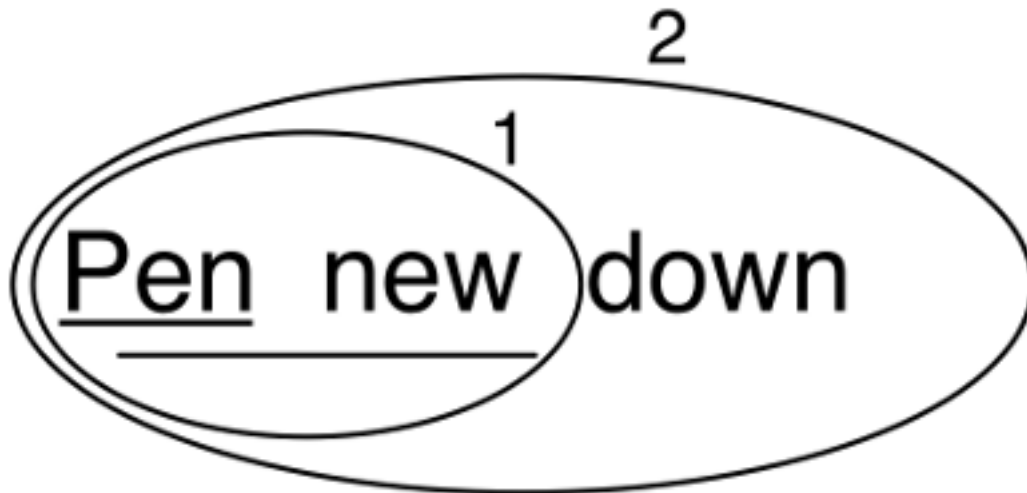
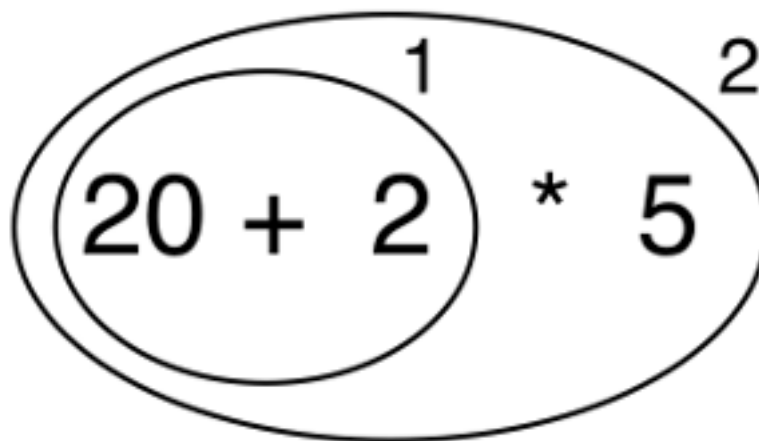


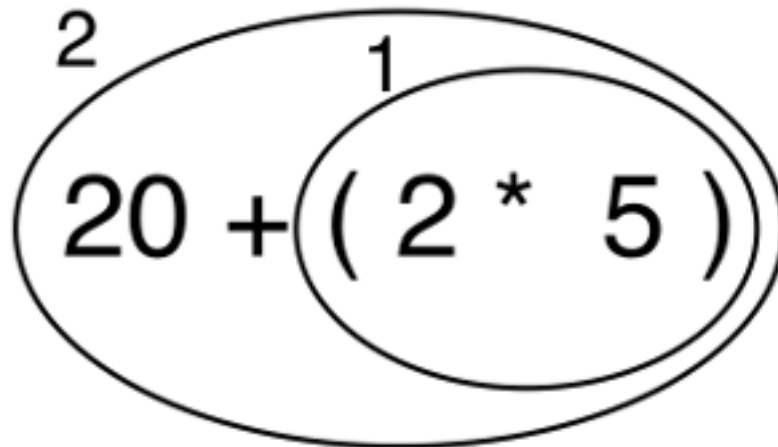
FIGURE 4.6 – Décomposition de Pen new down

4-3-4 - Incohérences arithmétiques

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

```
3 + 4 * 5    -> 35 "(pas 23) les messages binaires sont envoyés de gauche à droite"
3 + (4 * 5)  -> 23
1 + 1/3      -> (2/3) "et pas 4/3"
1 + (1/3)    -> (4/3)
1/3 + 2/3    -> (7/9) "et pas 1"
(1/3) + (2/3) -> 1
```





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

Exemple 4.4 – Décomposer $20 + 2 * 5$

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

```

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

```

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

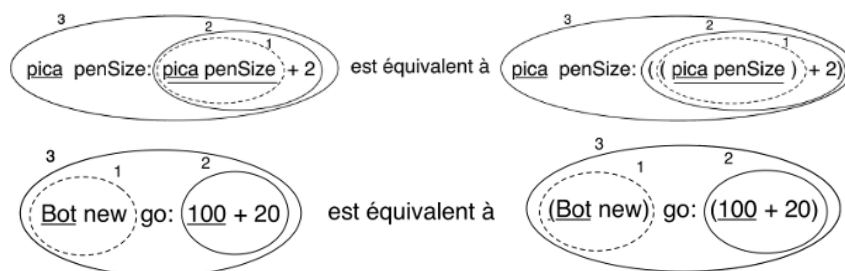
Exemple 4.5 – Décomposition de $20 + (2 * 5)$

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

```

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

```



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

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

Priorité implicite	Équivalent explicite avec parenthèses
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

4-4 - Quelques astuces pour identifier les messages à mots-clefs

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

4-4-1 - Des parenthèses ou pas ?

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

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

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

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



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

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

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

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

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

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

4-4-2 - Quand utiliser les [] ou les () ?

Vous pouvez avoir des difficultés à comprendre quand utiliser des crochets plutôt que des parenthèses. Le principe de base est que vous devez utiliser des [] lorsque vous ne savez pas combien de fois une expression peut être

évaluée (peut-être même jamais). [*expression*] va créer une fermeture lexicale ou bloc (c.-à-d.. un objet) à partir de *expression*, qui peut être évaluée autant de fois qu'il le faut (voire jamais) en fonction du contexte.

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

Les parenthèses quant à elles n'affectent que l'ordre d'envoi des messages. Aucun objet n'est créé, ainsi dans (*expression*), *expression* sera *toujours* évalué exactement une fois (en supposant que le code englobant l'expression soit évalué une fois).

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

4-5 - Séquences d'expression

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

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

4-6 - Cascades de messages

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

Expression `Msg1 ; Msg2`

```
Transcript show: 'Pharo est équivalent à :
est '
Transcript show: 'extra
'
Transcript cr.
```

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

Notez que l'objet qui reçoit la cascade de messages peut également être le résultat d'un envoi de message. En fait, le receveur de la cascade est le receveur du premier message de la cascade. Dans l'exemple qui suit, le premier message en cascade est `setX:setY` puisqu'il est suivi du point-virgule. Le receveur du message cascadié `setX:setY:` est le nouveau point résultant de l'évaluation de `Point new` et *non pas* `Point`. Le message qui suit `isZero` (pour tester s'il s'agit de zéro) est envoyé au même receveur.

```
Point new setX: 25 setY: 35; isZero -> false
```

4-7 - Résumé du chapitre

- Un message est toujours envoyé à un objet nommé le receveur qui peut être le résultat d'autres envois de messages.
- Les messages unaires sont des messages qui ne nécessitent pas d'arguments. Ils sont de la forme **receveur sélecteur**.

- Les messages binaires sont des messages qui concernent deux objets, le receveur et un autre objet *et* dont le sélecteur est composé d'un ou deux caractères de la liste suivante : `+`, `-`, `*`, `/`, `|`, `&`, `=`, `>`, `<`, `~` et `@`. Ils sont de la forme : `receveur sélecteur argument`.
- Les messages à mots-clefs sont des messages qui concernent plus d'un objet et qui contiennent au moins un caractère deux-points (:). Ils sont de la forme : `receveur motUnDuSelecteur: argumentUn motDeuxDuSelecteur: argumentDeux`.
- **Règle un.** Les messages unaires sont d'abord envoyés, puis les messages binaires et finalement les messages à mots-clefs.
- **Règle deux.** Les messages entre parenthèses sont envoyés avant tous les autres.
- **Règle trois.** Lorsque les messages sont de même nature, l'ordre d'évaluation est de gauche à droite.
- En Smalltalk, les opérateurs arithmétiques traditionnels comme `+` ou `*` ont la même priorité. `+` et `*` ne sont que des messages binaires ; donc `*` n'a aucune priorité sur `+`. Vous devez utiliser les parenthèses pour obtenir un résultat différent.

- 1 : Un fork, ou embranchement, est un nouveau logiciel créé à partir du code source d'un logiciel existant. Cela suppose que les droits accordés par les auteurs le permettent : ils doivent autoriser l'utilisation, la modification et la redistribution du code source. C'est pour cette raison que les forks se produisent facilement dans le domaine des logiciels libres. (Extrait de Wikipedia)
- 2 : Dan Ingalls et al., Back to the Future : The Story of Squeak, a Practical Smalltalk Written in Itself dans Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97). ACM Press, novembre 1997. URL : <http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html>.
- 3 : Un framework est un ensemble de bibliothèques, d'outils et de conventions permettant le développement d'applications.
- 4 : <http://SqueakByExample.org/fr> ; traduction française de Squeak By Example (<http://SqueakByExample.org>).
- 5 : Alec Sharp, Smalltalk by Example. McGraw-Hill, 1997 hURL: <http://stephane.ducasse.free.fr/FreeBooks/ByExample/i>.
- 6 : <http://www.surfscranton.com/architecture/KnightsPrinciples.htm>
- 7 : Basé sur Squeak3.9, Pharo utilise par défaut une machine virtuelle similaire.
- 8 : Les couleurs de boutons sont rouge, jaune et bleu. Les auteurs de ce cours n'ont jamais pu se souvenir à quelle couleur se réfère chaque bouton.
- 9 : En anglais, le terme utilisé est « to actclick ».
- 10 : Notez que les icônes Morphic sont inactives par défaut dans Pharo, mais vous pouvez les activer avec le Preferences Browser que nous verrons plus loin.
- 11 : Souvenez-vous que vous pourriez avoir besoin d'activer l'option **halosEnabled** dans le *Preference Browser*.
- 12 : N.d.T. C'est une tradition de la programmation : tout premier programme dans un nouveau langage de programmation consiste à afficher la phrase en anglais « hello world » signifiant « bonjour le monde ».
- 13 : Ce navigateur est confusément référé sous les noms « System Browser » ou « Code Browser ». Pharo utilise l'implémentation OmniBrowser du navigateur connue aussi comme « OB » ou « Package Browser ». Dans ce cours, nous utiliserons simplement le terme de Browser ou, s'il y a ambiguïté, nous parlerons de navigateur de classes.
- 14 : Si votre Browser ne ressemble pas à celui visible sur la figure 1.10, vous pourriez avoir besoin de changer le navigateur par défaut. Voyez la FAQ 7, p. 349
- 15 : Si une fenêtre s'ouvre soudain avec un message d'alerte à propos d'une méthode obsolète — le terme anglais est deprecated method — ne paniquez pas : le Method Finder est simplement en train d'essayer de chercher parmi tous les candidats incluant ainsi les méthodes obsolètes. Cliquez alors sur le bouton **Proceed**.
- 16 : Kent Beck, Test Driven Development : By Example. Addison-Wesley, 2003, ISBN 0-321-14653-0.
- 17 : En anglais, [http://en.wikipedia.org/wiki/Lights_Out_\(game\)](http://en.wikipedia.org/wiki/Lights_Out_(game)).
- 18 : Nous supposons que le Browser est installé en tant que navigateur de classes par défaut. Si le Browser ne ressemble pas à celui de la figure 2.2, vous aurez besoin de changer le navigateur par défaut. Voyez la FAQ 7, p. 349.
- 19 : Nous utilisons le terme « quote » en anglais.
- 20 : En anglais, puisque c'est la langue conventionnelle en Smalltalk.
- 21 : N.D.T. : pas encore classées.
- 22 : En fait, les variables d'instances peuvent être accédées également dans les sous-classes.
- 23 : <http://www.nongnu.org/cvs>

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

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

26 : VM est l'abrégié de « *Virtual Machine* » c.-à-d. « Machine Virtuelle ».

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

28 : Le commentaire de la méthode dit : « Retourne le nombre de lignes représentées par le receveur, dans lequel chaque `cr` ajoute une ligne »

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

30 : Le commentaire de la méthode dit : « Ajoute le receveur à l'argument et renvoie le résultat en réponse s'il s'agit d'un entier de classe `SmallInteger`. Échoue si l'argument ou le résultat n'est pas un `SmallInteger`. Voir la documentation de la classe `Object` : *whatIsPrimitive* (qu'est-ce qu'une primitive). »