MANUELS INFORMATIQUES MASSON

Gilles Clavel • Nathalie Lopez
Luc Veillon

Programmer objets avec Smalltalk

applications OS/2 Windows et AIX avec IBM Smalltalk

2e édition actualisée et complétée

Programmer objets avec Smalltalk

Applications OS/2, Windows et AIX, avec IBM Smalltalk

CHEZ LE MÊME ÉDITEUR

Des mêmes auteurs

COMPRENDRE ET UTILISER C++ POUR PROGRAMMER OBJET, par G. CLAVEL, I. TRILLAUD, L. VEILLON. *Collection MIM-Algorithmique, programmation.* 1994, 248 pages.

INTRODUCTION À LA PROGRAMMATION, par G. CLAVEL et J. BIONDI. *Collection MIM-Algorithmique, Programmation.*
Tome 2. — Structures des données. 1994, 3e tirage, 272 pages. Également publié en espagnol (1985) et en italien (1985).

Dans la collection MIM

LE DÉVELOPPEMENT DE LOGICIEL EN C++, par R. WINDER. Traduit de la 2e édition anglaise par P.-Y. BONNETAIN. 1994, 568 pages.

CONCEPTION OBJET DES STRUCTURES DE DONNÉES. Réalisation en langage C, par B. QUÉMENT. 1992, 248 pages.

LE LANGAGE C, norme ANSI, par B.W. KERNIGHAN et D.M. RITCHIE. Traduit de l'anglais par J.-F. GROFF et E. MOTTIER. 1994, 2e édition, 4e tirage, 296 pages.

LE LANGAGE C, solutions aux exercices de la 2e édition de l'ouvrage de B.W. KERNIGHAN et D.M. RITCHIE, par C.L. TONDO et S.E. GIMPEL. Traduit de l'anglais par D. BERTIER. 1992, 2e édition, 2e tirage, 168 pages.

LA BIBLIOTHÈQUE C STANDARD, par P.J. PLAUGER. Traduit par I. SAINT-SAËNS. Coédition avec Prentice Hall. 1995, 544 pages.

Autres ouvrages

INGÉNIERIE DES OBJETS. Approche classe-relation, application à C++, par Ph. DESFRAY. *Méthodes Informatiques et Pratique des Systèmes.* 1993, 2e tirage, 248 pages.

L'APPROCHE OBJET. Concepts et techniques, par R. MOREAU. *Collection Méthodes Informatiques et Pratique des Systèmes.* 1994, 312 pages.

OMT, par J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY et W. LORENSEN. Coédition Prentice Hall.
Tome 1. — MODÉLISATION ET CONCEPTION ORIENTÉES OBJET. Édition française revue et augmentée, compléments sur la 2e génération de la méthode. Traduit de l'anglais par A.-B. FONTAINE, G.-P. REICH et V. ZAÏM. 1995, 3e tirage, 536 pages.
Tome 2. — SOLUTION DES EXERCICES. Traduit de l'anglais par A.-B. FONTAINE et V. ZAÏM. 1996, 264 pages.

MANUELS INFORMATIQUES MASSON

Programmer objets avec Smalltalk

Applications OS/2, Windows et AIX, avec IBM Smalltalk

Gilles CLAVEL

Directeur consultant de la société IMA-informatique
Professeur à l'Institut national agronomique

Nathalie LOPEZ

Ingénieur de la société IMA-informatique
Responsable des formations méthodes objet

Luc VEILLON

Chargé de recherches à l'ORSTOM
Responsable informatique du laboratoire ERMES

2e édition actualisée et complétée

Paris Milan Barcelone

Ce logo a pour objet d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, tout particulièrement dans le domaine universitaire, le développement massif du «photocopillage».
Cette pratique qui s'est généralisée, notamment dans les établissements d'enseignement, provoque une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que la reproduction et la vente sans autorisation, ainsi que le recel, sont passibles de poursuites. Les demandes d'autorisation de photocopier doivent être adressées à l'éditeur ou au Centre français d'exploitation du droit de copie: 3, rue Hautefeuille, 75006 Paris. Tél.: 43 26 95 35.

Extraits : © Digitalk, Inc., 1991. Tous droits réservés.

Cet ouvrage est la 2e édition de *Découvrir la programmation orientée objets avec Smalltalk V* par G. Clavel et L. Veillon.

Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de l'éditeur, est illicite et constitue une contrefaçon. Seules sont autorisées, d'une part, les reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective, et d'autre part, les courtes citations justifiées par le caractère scientifique ou d'information de l'œuvre dans laquelle elles sont incorporées (art. L. 122-4. L. 122-5 et L. 335-2 du Code de la propriété intellectuelle).

© *Masson, Paris, 1991, 1996*

ISBN : 2-225-**85157**-3

ISSN : 0249-6992

MASSON S.A. 120, bd Saint-Germain, 75280 Paris Cedex 06

Table des matières

Avant-propos IX

1 - Objets, messages, classes et méthodes

1.1 - Des objets et des messages 1
1.2 - Evaluer une séquence d'expressions 2
1.3 - Premier contact avec l'interface-utilisateur de Smalltalk 5
1.4 - Classes et méthodes 7
1.5 - Classes et instanciation 10
1.6 - Hiérarchie des classes et héritage 12
1.7 - Héritage et polymorphisme 14
1.8 - Les outils de Smalltalk pour la manipulation des classes 17

2 - Le langage Smalltalk

2.1 - Les expressions 21
2.2 - Les constantes littérales 21
2.2.1 - *Les constantes numériques* 22
2.2.2 - *Les constantes-chaînes* 22
2.2.3 - *Les constantes-caractères* 22
2.2.4 - *Symboles et constantes-symboles* 22
2.2.5 - *Les tableaux* 23
2.3 - Les séquences d'expressions 24
2.4. - Les règles de priorité pour l'évaluation d'une expression 25
2.4.1 - *Messages unaires* 25
2.4.2 - *Messages binaires* 25
2.4.3 - *Messages à mot(s)-clé(s)* 25
2.4.4 - *Les priorités* 26
2.6 - Les variables d'instance 27
2.7 - Gestion dynamique de la mémoire disponible 30
2.8 - Les variables temporaires 33
2.8.1 *Les variables temporaires d'une séquence d'expressions* 33
2.8.2 *Les arguments d'un bloc* 33
2.8.3 *Les arguments d'une méthode* 35
2.9 - Les variables partagées 35
2.9.1 - *Les variables globales* 36
2.9.2 - *Les dictionnaires partagés* 37
2.9.3 - *Les variables de classe* 38

2.10 - La classe Context et l'évaluation d'un bloc 39
2.10.1 - *Un bloc est un objet* 40
2.10.2 - *Bloc récursif* 41
2.10.3 - *Blocs avec arguments* 41
2.11 - Les classes True et False et la réalisation d'un choix 42
2.11.1 - *Schémas conditionnels* 42
2.11.2 - *Opérateurs logiques* 43
2.12 - La classe Context et les itérations. 44

3 - Aller plus loin avec Smalltalk

3.1 - Faire des calculs 47
3.1.1 - *La représentation littérale des nombres* 48
3.1.2 - *Entiers, rationnels et réels approchés* 48
3.1.3 - *Inspection d'un objet* 50
3.1.4 - *La classe Float* 51
3.1.5 - *Opérations arithmétiques élémentaires* 53
3.1.6 - *Les intervalles* 55
3.2 - Les tableaux 56
3.2.1 - *Variables et tableaux* 56
3.2.2 - *Un exemple d'utilisation des tableaux : le calcul matriciel* 59
3.3 - Les chaînes de caractères 65
3.3.1 - *Quelques manipulations de chaînes de caractères* 66
3.3.2 - *Problèmes d'héritage liés à la classe String* 68
3.3.3 - *Les arguments d'une méthode : variables ou objets ?* 72
3.4 - Créer une classe, sauvegarder et récupérer ses méthodes 74
3.4.1 - *Quel type de classe créer ?* 74
3.4.2 - *Comment sauvegarder une classe et ses méthodes ?* 77
3.4.3 - *Restaurer une classe* 77
3.5 - Les dictionnaires de Smalltalk 78
3.5.1 - *La classe Association* 78
3.5.2 - *Un tableau pour enregistrer des associations* 79
3.5.3 - *Un dictionnaire pour enregistrer des associations* 81
3.5.4 - *Un exemple de dictionnaire analogique* 82
3.5.5 - *La variable super* 85
3.5.6 - *Enrichir le dictionnaire des analogies* 87
3.5.7 - *Le dictionnaire de Smalltalk* 90

4 - La classe Collection et sa descendance

4.1 - Collection et ses sous-classes 95
4.2 - La classe Bag : aperçu de quelques méthodes 97
4.2.1 - *Créer une instance de la classe Bag* 97
4.2.2 - *Remplir une instance de la classe Bag* 97
4.2.3 - *Explorer le contenu d'une instance de la classe Bag* 98
4.3 - Les classes Set et Dictionary 100
4.3.1 - *Des méthodes de conversion présentes dans Collection* 100
4.3.2 - *Une sous-classe importante de Set : Dictionary* 101
4.3.3 - *Les méthodes de Dictionary classées par fonctions* 101

4.3.4 - *Un exemple d'utilisation de dictionnaire : la paye du personnel* ... 103

4.4 - Classe IndexedCollection ... 105

4.4.1 - *Quelques méthodes pour manipuler les index* ... 106

4.4.2 - *Traiter la collection (méthodes définies dans IndexedCollection)* ... 106

4.4.3 - *Modifier des objets de la collection* ... 107

4.4.4 - *Copier la collection* ... 107

4.4.5 - *Effectuer des traitements sur tous les objets d'une collection* ... 107

4.5 - La classe FixedSizeCollection ... 108

4.6 - La classe OrderedCollection ... 113

4.6.1 - *Principales méthodes de OrderedCollection* ... 114

4.6.2 - *Un exemple d'utilisation de OrderedCollection : une salle d'attente.* ... 115

4.6.3 - *La classe SortedCollection* ... 117

4.7 - La classe Bag et ses méthodes ... 118

4.7.1 - *Redéfinition des méthodes inadéquates pour la classe Bag* ... 118

4.7.2 - *Initialisation de la classe Bag* ... 119

4.7.3 - *La variable d'instance elements* ... 120

4.8 - Méthodes directement définies dans la classe Collection ... 121

4.8.1 - *Ajouter un objet dans une collection* ... 121

4.8.2 - *Passer d'une sous-classe de Collection à une autre* ... 122

4.9 - Quelques méthodes de Set ... 123

4.9.1 - *Ajouter un objet dans une instance de Set* ... 123

4.9.2 - *Stockage des objets dans une instance de Set : notion de hachage* ... 124

4.9.3 - *Croissance d'une instance de Set* ... 128

4.10 - La classe Dictionary et ses sous-classes ... 130

4.10.1 - *Ajouter des éléments à un dictionnaire* ... 130

4.10.2 - *Limiter les redéfinitions de méthodes dans une sous-classe : l'exemple de select et reject* ... 131

4.10.3 - *La classe IdentityDictionary : un dictionnaire un peu particulier* ... 132

4.11 - La classe IndexedCollection ... 134

4.11.1 - *La classe IndexedCollection* ... 134

4.11.2 - *La classe FixedSizeCollection* ... 135

4.11.3 - *La classe OrderedCollection* ... 135

4.11.4 - *La classe SortedCollection* ... 136

5 - La classe Stream et sa descendance

5.1 - Premiers pas dans la classe Stream ... 137

5.1.1 - *Qu'est-ce qu'un flux ?* ... 138

5.1.2 - *Pourquoi une classe réservée aux flux ?* ... 138

5.1.3 - *Opérations de transferts* ... 139

5.1.4 - *Ouvrir un flux* ... 140

5.2 - Utiliser une instance de ReadStream ... 141

5.2.1 - *Comment créer une instance de ReadStream ?* ... 141

5.2.2 - *Parcourir une collection avec un flux* ... 142

5.2.3 - *Découper une collection en sous-collections avec un flux* ... 143

5.2.4 - *Tester la présence d'un objet sans déplacer la position courante* 144
5.2.5 - *Parcourir une collection de repère en repère* 145
5.2.6 - *Accès direct dans une collection* 146
5.3 - Les sous-classes WriteStream et ReadWriteStream 146
5.3.1 - *Différences entre les sous-classes de Stream* 146
5.3.2 - *Ouvrir un flux en écriture* 147
5.3.3 - *La variable d'instance collection et son emploi selon les sous-classes de Stream* 149
5.3.4 - *Les variables d'instance de position* 151
5.4 - Travailler sur un fichier avec FileStream 153
5.4.1 - *Comment accéder à un fichier sous Smalltalk ?* 153
5.4.2 - *Copier un fichier existant* 156
5.4.3 - *Manipuler des octets* 160

6 - Réaliser une application avec IBM Smalltalk

6.1 - Quelle application ? 167
6.1.1 - *Le cahier des charges* 167
6.1.2 - *La structure des fichiers-dictionnaires* 168
6.2 - Une classe, comme support de l'application 169
6.2.1 - *La nouvelle variable d'instance et l'héritage* 169
6.2.2 - *L'interface avec les fichiers* 171
6.2.3 - *L'environnement objets facilite la programmation incrémentale* 174
6.3 - L'interface graphique de l'application 177
6.3.1 - *Faire apparaître une fenêtre dédiée à l'application* 177
6.3.2 - *Les différentes classes de fenêtres* 179
6.3.3 - *Faire apparaître des informations dans la fenêtre* 180
6.3.4 - *Gérer un menu pour la fenêtre* 182
6.4 - Structurer la fenêtre de l'application 186
6.4.1 - *Une nouvelle classe pour l'application* 187
6.4.2 - *Le schéma Model / View / Controller* 188
6.4.3 - *Créer une instance de la classe ConsulteDico* 189
6.4.4 - *Partager la fenêtre en plusieurs panneaux* 190
6.4.5 - *Synchroniser les panneaux* 192
6.4.6 - *Améliorer la synchronisation* 193
6.5 - L'application finale 193
6.5.1 - *Les nouvelles variables d'instance de la classe ConsulteDico* 195
6.5.2 - *Des variables d'instance pour désigner les panneaux* 196
6.5.3 - *La version finale de la méthode ouvrirFenetre* 197
6.5.4 - *Effet d'une sélection dans le panneau droit* 198
6.5.5 - *Le menu des panneaux verticaux* 201

Annexe - utilisation d'un dictionnaire analogique : l'application 203

Bibliographie 217

Index 219

Avant-propos

Voici un ouvrage pratique, sur un thème d'actualité. Notre précédent livre de 1991 anticipait une évolution que les dernières années ont largement confirmée : l'objet est aujourd'hui au cœur de l'industrie du logiciel. Dans ce nouvel ouvrage nous expliquons les mécanismes de l'approche objet à travers Smalltalk, le langage qui en a le premier formalisé les concepts. Ce langage lui aussi fait partie de l'actualité : la concurrence entre ses différentes implémentations le montre bien.

Réaliser un ouvrage sur Smalltalk, en s'appuyant sur deux implémentations différentes, est sans doute un exercice délicat. Si nous avons choisi de le faire, c'est pour montrer que le langage est désormais suffisamment uniforme pour que sa présentation ne dépende pas d'une implémentation particulière. D'une version à l'autre, les principales différences concernent la programmation d'une interface graphique, étudiée ici au chapitre 6 avec IBM Smalltalk.

Par rapport à 1991, date de notre premier livre, la situation industrielle est bien différente. Smalltalk ne sert plus uniquement à illustrer les concepts objets. Il est désormais un outil reconnu pour le développement de logiciel. Le fait que le marché soit très disputé entre trois éditeurs (dont deux viennent de décider de se rapprocher) en est une preuve. La majeure partie des caractéristiques du langage est commune aux trois implémentations. Nous les présentons dans les cinq premiers chapitres avec des exemples en Smalltalk V et IBM Smalltalk. Pour l'étude de l'interface graphique, chaque éditeur a choisi une implémentation différente et nous avons été contraints de faire un choix, en retenant celle d'IBM Smalltalk.

Construit comme une découverte progressive, cet ouvrage introduit les concepts en s'appuyant à chaque fois sur des exemples pratiques. Notre expérience de formateurs nous a montré que cette approche était efficace. Le livre s'articule en trois parties :

- Les trois premiers chapitres sont consacrés à la découverte des concepts de base, présentés à travers Smalltalk, mais que l'on pourra retrouver dans la plupart des environnements objets.
- Les deux chapitres suivants approfondissent ces concepts, en expliquant les mécanismes internes mis en jeu et en s'appuyant sur l'architecture propre à Smalltalk.
- La dernière partie montre la facilité d'emploi et la grande simplicité d'un langage objets pour construire des applications avec une interface graphique.

Nous souhaitons que le lecteur prenne autant de plaisir que nous en avons trouvé à programmer objets avec Smalltalk. De tous les langages informatiques que nous connaissons, Smalltalk est sans doute le plus gratifiant, grâce à la communication qu'il installe entre le programmeur et son application. Ailleurs, des variables subissent des traitements ; ici des objets répondent ou refusent des messages : la mise au point d'un programme s'apparente plus à une conversation qu'à une laborieuse recherche d'erreurs. Un autre aspect de la programmation objets est le confort qu'elle offre pour programmer aussi bien de manière ascendante que descendante. Et là est sans doute une des raisons de son succès croissant : voilà enfin une approche qui permet d'obtenir des programmes capables d'évoluer. Car nous sommes convaincus, pour reprendre littéralement l'expression de Brooks, que « *programs are grown, not built* » .

Nous voulons ici remercier tous ceux qui nous ont aidés lors de la rédaction de cet ouvrage. La société IMA-informatique a fourni le support logistique pour la réalisation technique et matérielle du manuscrit. Son concours nous a été précieux et nous lui en sommes vivement reconnaissants. Nous exprimons aussi notre gratitude à la société IBM-France qui a soutenu efficacement notre projet. Nous remercions également Christian Burgei qui a développé en Smalltalk le logiciel qui a permis de réaliser l'index de ce livre et qui a patiemment validé le texte de cet index. Enfin, nous ne saurions oublier Frédérique Lauque qui avait assuré la composition du livre de 1991 : la qualité de son travail nous a grandement facilité la reprise de ceux des textes de 1991 qui étaient restés actuels.

Gilles Clavel, Nathalie Lopez et Luc Veillon
Paris - Orléans, juillet 1995

1 - Objets, messages, classes et méthodes

1.1 - Des objets et des messages

Pour celui qui l'aborde en ayant toujours pratiqué des langages classiques, Smalltalk est un peu surprenant. Les expressions que Smalltalk est amené à évaluer dans l'exécution d'une application ont la plupart du temps une forme inhabituelle. Considérons par exemple les expressions du langage Smalltalk qui sont représentées à la figure suivante.

```
5 + 3
'Alexandre' size
'abcde' reverse
3 between: 1 and: 5
'voici une petite phrase' subStrings size
```

Figure 1.1 : quelques expressions du langage

Si la première d'entre elles a, pour tous les programmeurs une allure familière, elle n'en est pas moins interprétée par Smalltalk d'une manière particulière. Pour Smalltalk en effet, cette expression fait intervenir deux *objets* (les entiers *5* et *3*) et un *message* (+). Les deux objets n'ont pas des rôles symétriques : l'un d'entre eux est *actif* : l'objet *5 reçoit* le message et y *répond.* Sa réponse sera le résultat de l'évaluation de l'expression. L'autre objet (*3*) a un rôle *passif* : il est le paramètre ou l'argument du message +.

« Que de complications pour une écriture qui, dans un langage comme C ou Pascal, s'évaluera 8 » sera tenté de dire le lecteur. Rassurons-le tout d'abord sur le résultat de l'évaluation par Smalltalk : il sera identique à celui obtenu avec un langage classique. Demandons-lui ensuite un peu de recul et de la patience : l'informatique est une discipline qui a vite fait d'enfermer l'utilisateur dans une habitude hors de laquelle celui-ci portera souvent un jugement a priori défavorable sur tout ce qui peut le déranger de cette habitude. Examinons les autres exemples de la figure 1.1, pour nous *habituer au dérangement.*

```
'Alexandre' size
```

L'objet qui reçoit le message figure toujours au début de l'expression : il s'agit de la chaîne de caractères *'Alexandre'* qui reçoit le message *size*. En réponse, l'objet communique sa taille, c'est-à-dire son nombre de caractères. Le résultat de l'évaluation de Smalltalk est donc ici *9*. Notons que le message *size* n'a pas d'argument : c'est un message *unaire*. Dans l'exemple précédent, le message + utilise un argument. C'est un message *binaire*.

```
'abcde' reverse
```

L'évaluation produira la chaîne *'edcba'*. En effet, le message unaire *reverse* demande à l'objet *'abcde'* de renvoyer l'objet qui est obtenu en énumérant les caractères dans l'ordre inverse.

```
3 between: 1 and: 5
```

Ici, l'objet *3* reçoit le message à mot-clés *between:and:* qui est un message à deux arguments. Dans un tel message, chaque mot-clé se termine obligatoirement par le signe : et doit être suivi par un argument. Ce message *demande* à l'objet *3* de *dire* s'il se classe entre *1* et *5*. L'évaluation par Smalltalk donne la réponse de l'objet *3* : *true*. Cette dernière valeur est l'objet du langage Smalltalk qui représente la valeur logique *vrai*.

Notons, à propos de tous les exemples précédents, que la réponse d'un objet à un message est toujours un autre objet : *8*, *9*, *'edcba'* et *true* sont eux-mêmes des objets qui, à leur tour, peuvent recevoir des messages, ce qu'illustre le dernier exemple de la figure 1.1 :

```
'voici une petite phrase' subStrings size
```

Cette expression comprend un objet (la chaîne *'voici une petite phrase'*) et deux messages unaires (*subStrings* et *size*). Pour suivre l'évaluation qui sera faite par Smalltalk, il est nécessaire de savoir que, dans une succession de deux messages unaires, le premier est traité avant le second et que, si o_1 est un objet et m_1 et m_2 deux messages unaires, l'expression o_1 m_1 m_2 s'évaluera de la manière suivante :

- l'objet o_1 reçoit le message m_1 et renvoie en réponse un objet o_2 ;
- le nouvel objet o_2 reçoit le message m_2 et renvoie en réponse un objet o_3.

Dans notre exemple, la chaîne *'voici une petite phrase'* reçoit le message *subStrings* et répond en renvoyant un tableau dont chaque élément est un mot de la chaîne. Ce tableau reçoit ensuite le message *size* et répond en indiquant sa dimension : il renvoie donc l'objet *4*.

1.2 - Evaluer une séquence d'expressions

Examinons la suite d'expressions du langage Smalltalk présentée à la figure 1.2. Pour en faciliter la lecture, nous l'avons écrite sur quatre lignes. Les trois dernières sont des expressions que Smalltalk devra évaluer. Conformément aux règles d'écriture du langage, chaque expression qui n'est pas la dernière est suivie d'un point qui joue le rôle de séparateur. La première ligne de la séquence n'est

pas une expression à évaluer : elle représente une déclaration de deux variables *chaine* et *chaineInverse*. Ces deux variables sont des variables temporaires : elles seront créées dans l'environnement de Smalltalk au début de l'évaluation de la séquence et elles disparaîtront de cet environnement à la fin de cette évaluation.

Avant d'étudier le mécanisme de l'évaluation de la séquence de la figure 1.2, précisons quel est le contexte de l'utilisateur : comment pourra-t-on demander à Smalltalk l'évaluation de cette séquence ?

On utilise généralement Smalltalk par l'intermédiaire d'un poste de travail clavier / écran disposant d'une souris. A l'écran, dans l'environnement de travail de Smalltalk, on peut faire apparaître une ou plusieurs fenêtres et, parmi celles-ci, il y en a toujours au moins une qui est une fenêtre d'édition de texte : on peut aisément y enregistrer la saisie de la séquence de la figure 1.2 puis, à l'aide de la souris, on peut sélectionner le texte complet de cette séquence. Le texte sélectionné apparaît alors à l'écran en inverse vidéo. Ensuite on active le menu de la fenêtre dans lequel on choisira l'option *Execute*. En réponse, Smalltalk effectuera l'évaluation du texte sélectionné. Pour préciser la manière d'effectuer ces manipulations, des indications supplémentaires seront données, avec des exemples, au paragraphe suivant (1.3).

```
|chaine  chaineInverse|
chaine  :=  'abcde'.
chaineInverse :=  chaine reverse.
Transcript  nextPutAll: chaineInverse
```

Figure 1.2 : une première séquence d'expressions

Revenons à l'évaluation de la séquence de la figure 1.2. La première expression évaluée :

```
chaine := 'abcde'
```

affecte à la variable *chaine* l'objet *'abcde'*. Plutôt que d'évoquer le mécanisme d'affectation classique des langages usuels, on peut considérer que l'opération revient à donner le nom *chaine* à l'objet *'abcde'* ce qui nous permet, une variable étant elle-même un objet, de parler désormais de l'objet *chaine*, dont la valeur est *'abcde'*[1]. Ici, l'objet *chaine* est une variable temporaire : par convention, toute variable déclarée entre deux barres verticales (| et |) et dont le nom commence par une lettre minuscule est une variable temporaire, qui disparaîtra de l'environnement de Smalltalk, dès que la séquence dans laquelle elle a été déclarée aura été évaluée. On peut également déclarer des variables globales en leur choisissant un nom dont le premier caractère est une lettre majuscule. Une variable globale fera partie de l'environnement permanent de Smalltalk (et sera sauvegardée, avec cet environnement, à chaque fin d'exécution de Smalltalk). Une variable globale est donc permanente : elle ne disparaît que si l'utilisateur

1 En fait, Smalltalk va réaliser de manière interne l'affectation en faisant pointer l'objet sur la valeur affectée (cf chapitre 2, paragraphe 2.7).

prend la décision de la supprimer de l'environnement Smalltalk (cf. chapitre 2, paragraphe 2.9).

Toujours à propos des variables, remarquons que, lorsque le processeur Smalltalk évalue, dans le début de la séquence, les deux lignes :

```
|chaine  chaineInverse|
chaine := 'abcde'.
```

les opérations effectuées sont les suivantes :

- à l'issue du traitement de la première ligne, le processeur Smalltalk a créé les deux variables *chaine* et *chaineInverse* ; elles désignent toutes deux l'objet *nil*, qui sert à représenter toute valeur indéterminée.
- Après l'évaluation de la deuxième ligne, *chaine* représente l'objet *'abcde'* tandis que *chaineInverse* désigne toujours la valeur *nil*.

L'évaluation de la troisième ligne de la figure 1.2 va d'abord envoyer, à l'objet *chaine*, le message *reverse* (cf. 1.1). L'objet *chaine* va renvoyer en retour la chaîne *'edcba'*, qui va devenir la nouvelle valeur de la variable *chaineInverse*.

La dernière expression évaluée envoie le message *nextPutAll: chaineInverse* à l'objet *Transcript*. Le nom de cet objet commence par une majuscule : il s'agit donc d'un objet global, toujours présent dans l'environnement de Smalltalk. *Transcript* est l'éditeur de texte qui gère la fenêtre intitulée *System Transcript*. Cette fenêtre apparaît en permanence sur l'écran de l'environnement de travail de Smalltalk. La plupart des messages envoyés à l'utilisateur seront affichés dans cette fenêtre. L'utilisateur peut lui-même se servir de *Transcript* pour enregistrer et modifier du texte dans la fenêtre. Quand un objet-éditeur reçoit le message *nextPutAll:* avec une chaîne comme argument (*chaineInverse* sur la figure 1.2), la réponse de cet éditeur consiste à afficher la chaîne à l'écran à partir de la position du curseur. L'effet de la dernière évaluation de la séquence de la figure 1.2 est donc d'afficher, dans la fenêtre *System Transcript*, la chaîne *'edcba'*.

Pour en terminer avec l'étude de la séquence de la figure 1.2, signalons que l'écriture, en Smalltalk, est relativement libre et que les expressions peuvent se succéder sur une même ligne, à la seule condition d'être séparées les unes des autres par des points. On aurait donc pu présenter toute la séquence sur une seule ligne (les lignes traitées par un objet-éditeur ne sont pas limitées en longueur). Enfin, pour obtenir le résultat décrit, il n'était pas nécessaire d'utiliser les variables *chaine* et *chaineInverse* et l'on aurait pu tout aussi bien demander l'évaluation de :

```
Transcript nextPutAll: 'abcde' reverse
```

ou encore, bien entendu :

```
Transcript nextPutAll: 'edcba'
```

Avant d'aller plus loin, précisons, à propos de l'affectation, la distinction qu'il faut faire entre les termes de *variable* et d'*objet*. Nous reviendrons au chapitre 2 sur le mécanisme interne de l'opération d'affectation mais, dès à présent, indiquons que, quand nous « affectons » l'objet *o* à la variable *v*, avec l'évaluation de :

```
v := o
```

nous prenons ensuite la liberté de parler, par abus de langage, de l'*objet v* pour désigner la valeur de *v*, alors qu'il conviendrait plutôt de dire *l'objet désigné par la*

variable v. Cet abus de langage est bien compréhensible dans la mesure où, si on envoie après l'affectation le message *m* à la variable *v*, c'est l'objet *o* (valeur de *v*) qui répondra à ce message.

Pour terminer, remarquons enfin que, dans cet ouvrage, nous utiliserons systématiquement le symbole := pour noter l'affectation, conformément à la notation utilisée en IBM Smalltalk et Smalltalk V (Smalltalk 80 utilise le symbole ←).

1.3 - Premier contact avec l'interface-utilisateur de Smalltalk

Précisons maintenant quel est l'interface-utilisateur de Smalltalk. Quand on travaille avec Smalltalk, on peut faire apparaître une ou plusieurs fenêtres à l'écran et, à chaque instant, l'une d'entre elles est la fenêtre active. La figure 1.3 présente un tel écran sur lequel apparaissent deux fenêtres.

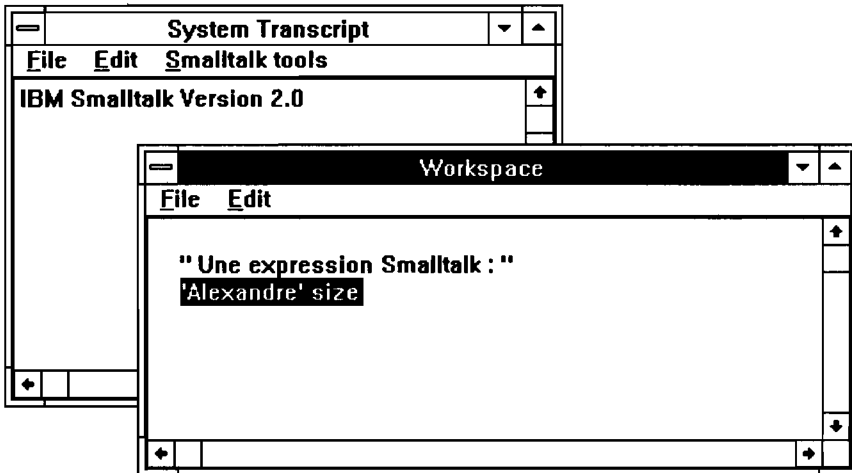
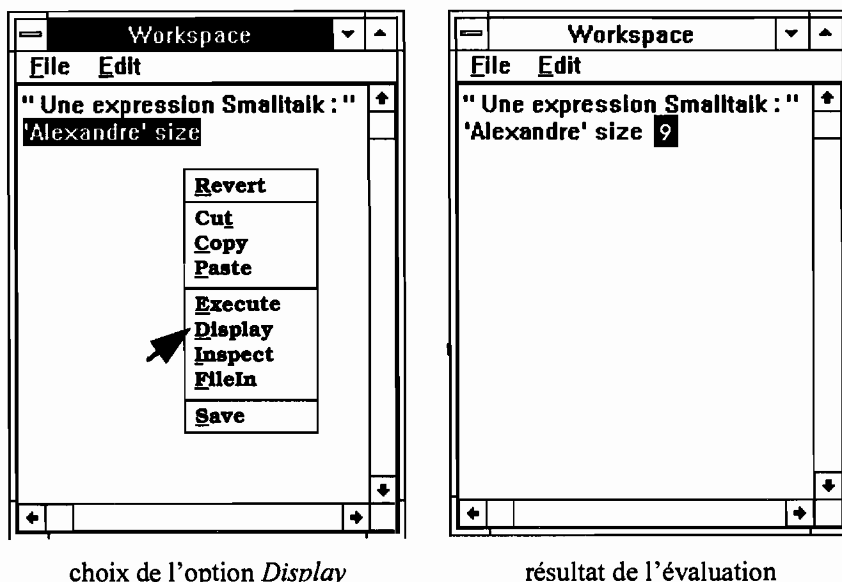


Figure 1.3 : deux fenêtres à l'écran de l'environnement Smalltalk

Sur cet écran, la fenêtre *System Transcript* apparaît en arrière plan et sa barre de titre est en caractères noirs sur fond blanc : ce n'est pas la fenêtre active. En revanche, la fenêtre *Workspace* est au premier plan et sa barre de titre est affichée en inversion vidéo : c'est la fenêtre active. Dans cette fenêtre, le texte *'Alexandre size'* : est sélectionné : il apparaît en inversion vidéo. Avec cette sélection, si l'utilisateur tape un caractère dans la fenêtre, ce caractère fera disparaître la sélection et viendra se placer à l'endroit qui était occupé par le premier caractère de cette sélection. Un des premiers usages possibles de la sélection est donc l'effacement du texte sélectionné[2].

2 On peut aussi effacer la sélection sans la remplacer par un caractère, en appuyant sur la touche d'effacement, immédiatement après avoir fait la sélection. Nous reviendrons au chapitre 3 sur les manipulations élèmentaires de l'interface graphique et en particulier sur les mécanismes de copie et suppression liés à la sélection.

Etudions maintenant l'utilisation de la sélection d'une expression pour déclencher une évaluation. Si nous revenons à la figure 1.3, où le texte *'Alexandre size'* est sélectionné dans la fenêtre *Workspace* et si nous cliquons, dans cette fenêtre, avec le bouton *droit* de la souris, nous faisons apparaître le menu général de l'édition de texte qui est présenté sur la fenêtre de gauche de la figure 1.4.



choix de l'option *Display* résultat de l'évaluation

Figure 1.4 : évaluation d'une expression avec l'option *Display*

Cette figure ne représente pas un écran de Smalltalk, mais deux états successifs de la fenêtre *Workspace*. La partie gauche de la figure donne la représentation de cette fenêtre après l'apparition du menu. Deux des options de ce menu permettent l'évaluation :

Display le choix de cette option déclenchera l'évaluation de la séquence sélectionnée et demandera à Smalltalk d'afficher, dans la fenêtre, immédiatement après la sélection, la dernière valeur renvoyée par l'évaluation.

Execute le choix de cette option déclenchera l'évaluation de la séquence sélectionnée sans affichage de la dernière valeur renvoyée.

Examinons la figure 1.4. Dans la fenêtre de gauche, la position du curseur sélectionne l'option *Display*, qui apparaît en inversion vidéo. Si l'on clique alors avec le bouton gauche de la souris, on déclenche l'évaluation et le résultat est représenté dans la partie droite de la figure. La conséquence visible de l'évaluation est double :

- l'expression qui était sélectionnée ne l'est plus : elle apparaît désormais en caractères noirs sur fond blanc ;
- la valeur résultant de l'évaluation est l'entier *9* (longueur de la chaîne *'Alexandre'*) ; elle apparaît sous la forme d'une nouvelle sélection qui suit immédiatement la dernière position de la sélection antérieure.

Pour terminer, revenons sur l'option *Execute*. Comme l'option *Display*, elle déclenche l'évaluation mais elle ne provoque pas l'affichage de la dernière valeur renvoyée par cette évaluation. On l'utilisera donc chaque fois que l'évaluation est faite pour obtenir un résultat qui n'est pas représenté par la dernière valeur renvoyée. En effet, de nombreux messages provoquent, de la part de l'objet qui les reçoit, une réaction qui ne peut être réduite au renvoi d'une valeur.

Par exemple, le message *nextPutAll: uneChaîne* reçu par l'objet *Transcript* entraînera l'apparition de *uneChaîne*, sur la fenêtre *System Transcript*, à l'emplacement du curseur.

On notera aussi qu'en réponse au message *nextPutAll:*, l'objet récepteur renvoie également sa propre valeur. Si donc on saisit, dans la fenêtre *Workspace*, l'expression :

```
Transcript nextPutAll: 'Bonjour'
```

et si, après avoir sélectionné cette expression, on en déclenche l'évaluation avec *Display*, on constatera deux modifications à l'écran :

- le mot *Bonjour* apparaîtra sur *System Transcript*, représentant ainsi la réaction de l'objet *Transcript* au message reçu,
- la locution *an EtTranscript* apparaîtra sur *Fenêtre de travail*, à la suite de l'expression évaluée. Cette locution désigne la catégorie des objets tels que *Transcript* (la *classe* de ces objets, cf. paragraphe suivant). Smalltalk indiquera ainsi, conformément au mode d'évaluation de l'option *Display*, que l'objet renvoyé en fin d'évaluation est un éditeur de texte.

En revanche, si on déclenche la même évaluation avec *Execute*, on obtiendra seulement l'apparition de *Bonjour* sur *System Transcript*.

Le mode d'évaluation *Execute* est donc à choisir quand le but de l'évaluation n'est pas l'examen de la dernière valeur renvoyée. A l'inverse, si l'on sélectionne une expression dont l'évaluation n'a pas d'autre effet que le retour de la valeur renvoyée par le récepteur, l'évaluation devra être faite avec *Display* pour faire apparaître le résultat. Ainsi, une évaluation avec *Execute* de *5+8* n'aura aucun effet visible à l'écran.

1.4 - Classes et méthodes

Nous avons vu que l'évaluation de *'abcde' reverse* renvoyait la chaîne *'edcba'*. Si nous essayons maintenant d'évaluer :

```
1234 reverse
```

nous obtenons, à l'écran, l'apparition d'une fenêtre de diagnostic qui indique la détection d'une erreur. Le libellé de l'erreur :

```
SmallInteger does not understand reverse
```

nous indique que l'objet *1234* n'a pas pu traiter le message *reverse*. Si donc un objet *x* « comprend » un message *m*, il n'en est pas forcément de même pour un autre objet *y*.

En fait, les objets sont répartis en *classes*, une classe rassemblant des objets ayant les mêmes propriétés. Dans Smalltalk, chaque classe définit les *messages* qui sont susceptibles d'être envoyés aux objets qui lui appartiennent : la définition d'un message décrit la *méthode* à appliquer (i.e. les expressions à évaluer) pour traiter ce message.

Ainsi, la classe *String*, à laquelle appartient l'objet *'abcde'*, « possède[3] » la méthode correspondant au *sélecteur* (c'est ainsi qu'on désigne le mot-clé identifiant le message) *reverse*. En revanche, la classe *SmallInteger*, à laquelle appartient l'objet *1234* ne contient aucune méthode dont le sélecteur est *reverse*. Nous en arrivons à l'un des aspects fondamentaux de la programmation orientée objets. A la différence de la programmation classique, où les données sont décrites à côté des traitements, la définition d'une classe d'objets comprend à la fois la description des données qui composent chaque objet de la classe et les traitements susceptibles d'être appliqués à ces objets (i.e. les méthodes). L'utilisateur d'un objet n'a pas besoin de connaître les détails de la réalisation des méthodes applicables à cet objet : il lui suffit d'en connaître les modalités de fonctionnement.

L'environnement de la programmation orientée objets permet à l'utilisateur de compléter, chaque fois qu'il le désire, la description d'une classe en modifiant ou en ajoutant des méthodes. Définissons par exemple, pour la classe *SmallInteger* d'IBM Smalltalk, qui comprend les entiers relatifs codés sur 32 bits, la méthode correspondant au sélecteur *reverse*. Le code correspondant à cette nouvelle méthode est présenté à la figure 1.5.

Examinons les opérations effectuées par l'exécution de cette méthode. La première ligne énonce le sélecteur de la méthode (message unaire *reverse*). Les deux lignes suivantes représentent un commentaire qui décrit le traitement effectué par la méthode. En Smalltalk, un commentaire peut s'écrire librement, sur une ou plusieurs lignes, encadré par des guillemets *"* et *"*.

Après le commentaire, apparaît la déclaration des variables temporaires qui seront utilisées par la méthode :

valeurAbsolue	prendra au départ la valeur absolue de l'objet récepteur ;
signe	indiquera son signe (+1 pour *plus*, - 1 pour *moins*) ;
inversePositif	contiendra la valeur inversée de la valeur absolue ;
chiffre	désignera chaque chiffre traité.

L'expression *(self between: -9 and: 9) ifTrue: [^ self]* représente une opération conditionnelle. Nous reviendrons en 2.11 sur l'évaluation d'une telle expression. Etudions-la brièvement. Le mot *self* est un mot réservé de Smalltalk qui sert, dans la description d'une méthode, à désigner l'objet récepteur du message correspondant à la méthode décrite (ici, la méthode *reverse*). Ainsi, l'évaluation de *self between: -9 and: 9* amène Smalltalk à envoyer le message à mots-clés *between: -9 and: 9* à *self*, c'est-à-dire à l'objet qui a reçu le message *reverse*. Si cet objet répond *true*, l'expression encadrée par des crochets, immédiatement après *ifTrue:*, est évaluée : c'est l'expression *^ self*. Le symbole ^ indique à Smalltalk

3 En fait, comme nous le verrons plus loin (cf chapitre 5), ce n'est pas la classe String qui définit la méthode reverse, mais la classe IndexedCollection, dont String est une descendante. Ainsi, String hérite de la méthode reverse.

qu'il doit renvoyer, comme réponse de la méthode, l'expression qui suit ce symbole. Une telle opération termine toujours l'exécution d'une méthode. Si donc l'objet récepteur est supérieur ou égal à -9 et inférieur ou égal à 9, la méthode se termine en envoyant cet objet comme réponse. Par exemple, l'évaluation de *8 reverse* renverra l'objet *8*.

```
reverse
 "renvoie l'entier obtenu en énumérant les chiffres du récepteur de la droite vers la gauche :
  par exemple 123 reverse renvoie 321"
  |inversePositif valeurAbsolue signe chiffre|
  "si le récepteur n'a qu'un chiffre, on renvoie ce chiffre"
  (self between: -9  and: 9)  ifTrue: [^ self].
  "ici, on est sûr que le récepteur a au moins deux chiffres"
  valeurAbsolue := self abs. "abs renvoie la valeur absolue"
  signe := self // valeurAbsolue. "// renvoie le quotient entier"
  inversePositif := 0.
  [valeurAbsolue > 0] whileTrue:  "tant qu'il reste des chiffres à traiter"
     [chiffre := valeurAbsolue \\ 10. "on divise par 10 pour prendre
     le chiffre de droite (\\ renvoie  le reste de la division entière)"
     inversePositif := inversePositif * 10 + chiffre.
     "on introduit ce chiffre dans inversePositif en décalant vers la gauche"
     valeurAbsolue :=  valeurAbsolue // 10
     "on divise par 10 afin de passer au chiffre suivant"
     ].
  "fin de l'itération, il reste à renvoyer l'entier inverse avec un signe correct"
  ^ signe * inversePositif
```

Figure 1.5 : méthode *reverse* pour la classe *SmallInteger*

Si l'objet receveur n'est pas dans l'intervalle [-9, +9], l'expression ^*self* n'est pas évaluée et l'exécution de la méthode continue en affectant des valeurs initiales aux variables *valeurAbsolue*, *signe* et *inversePositif*. Par exemple, si l'expression évaluée est *-357 reverse* , l'objet-récepteur (*self*) est ici *-357* et l'on affecte :

- l'objet *357* à *valeurAbsolue* (*self abs* renvoie la valeur absolue de *self*),
- l'objet *-1* à *signe* (signe *moins*),
- une valeur initiale nulle à *inversePositif*.

Ensuite, on aborde une itération dont la forme est :

[*prédicat*] `whileTrue:` [*expressions à évaluer tant que le prédicat est vrai*]

La forme générale des structures itératives sera étudiée au chapitre 2. Pour l'instant, le lecteur constatera aisément, en lisant le texte de la figure 1.5, que les évaluations répétées dans l'itération composent un nouvel entier dans la variable *inversePositif*. Cet entier est obtenu en prélevant les chiffres *par la droite* dans la valeur absolue de l'objet récepteur. Chaque chiffre ainsi obtenu est *inséré* avec un

décalage à gauche dans *inversePositif*. A la sortie de l'itération, on renvoie, en réponse, la valeur obtenue en lui attribuant le même signe que l'objet récepteur : ^ *signe * inversePositif*.

Si nous ajoutons, à la classe *SmallInteger*[4], la méthode *reverse*, telle qu'elle est définie dans la figure 1.5, nous pouvons alors envoyer à l'objet *1234* le message *reverse* et nous obtiendrons, en réponse, l'objet *4321*. Mais si nous essayons ensuite d'évaluer :

```
1231231234  reverse
```

nous constatons un échec : une fenêtre de diagnostic apparaît et nous annonce que le message *reverse* n'a pas été « compris ». Le contenu de la fenêtre donne l'indication supplémentaire suivante :

```
LargeInteger does not understand reverse
```

Ce diagnostic signifie que le message *reverse* a été reçu par un objet de la classe *LargeInteger*, pour laquelle il est inconnu. En effet, nous avons défini ce message pour les objets de la classe *SmallInteger* alors que l'entier *1231231234* est de la classe *LargeInteger*. Un objet ne peut, en principe, accepter que les messages définis pour sa classe.

1.5 - Classes et instanciation

Revenons sur les notions d'objet, de classe et de méthode. L'un des objectifs de la programmation orientée objets est d'encapsuler, dans un objet, les données et les méthodes qui sont propres à cet objet : c'est l'objet lui-même qui est responsable du traitement des messages qui lui seront adressés. La notion de classe peut être rapprochée de celle de type : une classe définira les propriétés de tous les objets qui lui seront associés. Chaque objet est, en quelque sorte, une matérialisation de sa classe. Un objet peut exister de manière permanente, comme par exemple l'objet *true* de la classe *True* (cf. 2.11), mais, le plus souvent, il est créé par *instanciation*. Pour cela, un message est envoyé à la classe.

En effet, dans Smalltalk, tout est objet et une classe est elle-même un objet qui peut recevoir des messages. Parmi les messages que l'on peut adresser à une classe donnée, il y en a souvent plusieurs qui servent à créer un nouvel objet de cette classe. Lorsqu'une classe *C* reçoit un tel message, elle répond donc en général en renvoyant une nouvelle *instance* de la classe, c'est-à-dire un nouvel objet, qui est ajouté à l'environnement de Smalltalk. La figure 1.6 présente quelques exemples d'instanciations.

Dans cette figure, les objets créés sont affectés aux variables temporaires *c1*, *c2*, *t*, *e*, *d* et *sac*. On notera que, dans chacun des messages d'instanciation, l'objet récepteur est toujours la classe dont on veut créer une instance. La réponse de la classe au message est l'objet créé, qui est alors affecté à la variable temporaire correspondante. Les deux premiers messages sont adressés à la classe *String* et

4 Nous verrons, en 1.8, comment modifier, ajouter et supprimer des méthodes pour une classe donnée

créent respectivement une chaîne vide (message *new*) et une chaîne de cinq caractères, de valeur indéterminée (message à mot-clé *new:5*).

Avec les deux exemples suivants, on constate que la même méthode *new:* est acceptée par les classes *Array* et *Set* pour créer un tableau et un ensemble.

L'évaluation suivante instancie un objet *date*. La classe *Date* définit les propriétés de ces objets. Elle répond au message *today* en renvoyant une instance qui représente la date d'aujourd'hui.

Enfin, l'évaluation de la dernière expression crée, avec la méthode *with:with:with:*, un objet de la classe *Bag*. Un tel objet (cf. chapitre 4) est un conteneur, dans lequel on peut ranger, sans ordre, des objets quelconques et qui peut, à la différence d'un ensemble, contenir plusieurs fois le même objet. Ici, l'objet créé *sac* contient trois objets : *'Durand'* et *'Lille'*, de la classe *String* et *1945* de la classe *SmallInteger*.

```
|c1 c2 t e d sac|
c1 := String new.   "c1 est une chaîne vide"
c2 := String new: 5.
    "c2 est une chaîne de cinq caractères, de valeur indéterminée"
t := Array new: 10.
    "t est un tableau de 10 éléments ; chaque élément a la valeur nil (UndefinedObject)"
e := Set new: 40.
    "e est un ensemble pouvant accueillir jusqu'à 40 éléments ; actuellement e est vide"
d := Date today.   "d est la date d'aujourd'hui"
sac := Bag with: 'Durand' with: 1945 with: 'Lille'.
    "sac est un objet contenant trois valeurs"
```

Figure 1.6 : exemples d'instanciations

Une classe dispose donc en général d'une ou de plusieurs *méthodes de classe* qui représentent des messages qui peuvent être adressés à la classe (et non pas aux objets de cette classe). Ces messages peuvent avoir pour effet de créer un objet, nouvelle instance de cette classe.

Les objets de la classe peuvent, à leur tour, recevoir des messages qui correspondront à des *méthodes d'instance*. La méthode d'instance *m*, définie dans la classe *C*, décrit donc le traitement que doit effectuer tout objet *o*, instance de *C*, en réponse à un message de sélecteur *m*.

En principe, un message correspondant à une méthode d'instance *m*, définie pour la classe *C*, ne peut être accepté par un objet d'une autre classe *C'*, pour laquelle *m* n'a pas été définie. C'est ce que nous avons constaté avec le message *reverse* défini pour la classe *String* et que nous avons redéfini pour la classe *SmallInteger*. Cependant, pour éviter de définir plusieurs fois la même méthode, comme nous pourrions être tentés de le faire, avec la méthode *reverse*, pour les

classes *SmallInteger* et *LargeInteger*, il existe un mécanisme fondamental pour la programmation orientée objets : *l'héritage*.

Avant d'étudier l'héritage, au paragraphe suivant, illustrons les notions qui viennent d'être exposées, en examinant les exemples de la figure 1.7. Le lecteur comprendra aisément l'effet de l'évaluation de la séquence sachant que :

- le message *at: indice put: valeur* correspond à la méthode d'instance *at:put:* de la classe *Array* et affecte *valeur* à l'élément de rang *indice* du tableau récepteur ;
- le message *printString* peut être envoyé à tout objet : il renvoie une représentation de cet objet sous forme d'une chaîne de caractères[5]. Ici, ce message est envoyé à l'objet *Array y* et la chaîne obtenue en réponse est fournie en argument d'un message *nextPutAll:* qui provoque l'apparition de cette chaîne sur la fenêtre *Transcript*.

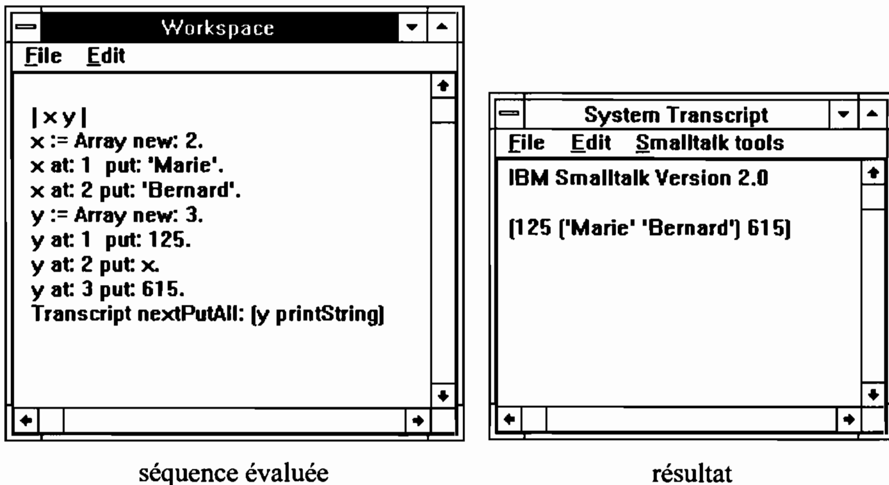


Figure 1.7 : méthodes de classe et d'instance (exemples de messages)

1.6 - Hiérarchie des classes et héritage

Mettons en évidence les mécanismes d'héritage, propres à Smalltalk, en généralisant la méthode *reverse* à tous les nombres entiers. Recopions, pour cela, cette méthode définie dans la classe *SmallInteger* et introduisons-la dans la classe *Integer*[6].

---

5 Selon les classes, cette représentation est plus ou moins détaillée. Par défaut, quand une classe CCC n'a pas prévu d'implémentation pour printString, Smalltalk utilise un message printString qui renvoie la chaîne 'a CCC' qui indique, avec l'article indéfini Anglais, que l'objet est une instance de CCC.

6 Nous verrons, en 1.8, comment modifier, ajouter et supprimer des méthodes pour une classe donnée.

On constate maintenant que :

- l'évaluation de *987654 reverse* donne *456789* ;
- l'évaluation de *-123456 reverse* donne *-654321* .

et, même si nous supprimons la méthode *reverse* de la classe *SmallInteger*, l'évaluation de *1234 reverse* donnera *4321*.

Nous avons utilisé le fait que la classe *Integer* est la classe-mère des deux classes *SmallInteger* et *LargeInteger*. La classe *SmallInteger* rassemble les entiers que l'on peut gérer directement avec les circuits de calcul de la machine. La classe *LargeInteger* rassemble les autres entiers dont la valeur absolue est plus grande et qui nécessitent des fonctions logicielles de calcul[7]. Ces deux *sous-classes* de la classe *Integer héritent* de l'ensemble des méthodes de leur classe-mère : en particulier, chaque objet de l'une de ces sous-classes reconnaîtra désormais la méthode *reverse*, dont il aura hérité de la classe *Integer*.

Plus généralement, une sous-classe *SC* d'une classe *C* est considérée comme une spécialisation de *C* qui peut comporter des propriétés supplémentaires par rapport à celles de *C*, ou encore des modifications de certaines propriétés de *C*. Mais, a priori, les objets de *SC* ont un comportement que l'on peut dériver de celui des objets de *C* : on dira qu'ils *héritent* des méthodes de *C* (méthodes de classes et méthodes d'instances). Si, toutefois, la spécialisation induite par *SC* nécessite la modification du comportement de *SC* ou d'un objet de *SC* pour la méthode *m*, on redéfinira, avec le même sélecteur, la méthode *m*, pour la classe *SC*, conformément au traitement nécessaire. Après cette redéfinition, chaque objet de *SC* reconnaîtra toutes les méthodes de la classe *C*, sauf la méthode *m* pour laquelle il utilisera la redéfinition qui lui est propre.

Chaque sous-classe d'une classe C peut elle même être la classe-mère d'une ou plusieurs sous-classes. L'ensemble des classes définies en Smalltalk forme un arbre dont la racine est la classe *Object*. Nous reviendrons sur la structure de cet arbre au paragraphe 1.7 et dans les chapitres suivants. Pour l'instant, intéressons-nous au sous-arbre dont l'origine est la classe *Magnitude* et qui est représenté à la figure 1.8.

La classe Magnitude est une classe *abstraite* qui rassemble des méthodes pour comparer, compter et mesurer des valeurs qui seront des objets de ses classes descendantes. Toutes ces classes descendantes hériteront des méthodes de la classe *Magnitude*.

Parmi ces classes descendantes, on trouve :

- la classe *Character*, qui rassemble les caractères du code utilisé,
- la classe *Date*, dont chaque instance représente une date du calendrier,
- les classes *Float*, *Fraction*, *Integer* et *LargeInteger* dont les instances représentent les valeurs numériques que l'on peut utiliser pour des calculs,
- la classe *Time*, dont chaque instance représente une heure de la journée.

---

7 Ces fonctions sont fournies et gérées par Smalltalk, qui permet ainsi de travailler avec des entiers dont la représentation nécessite plus de 32 bits.

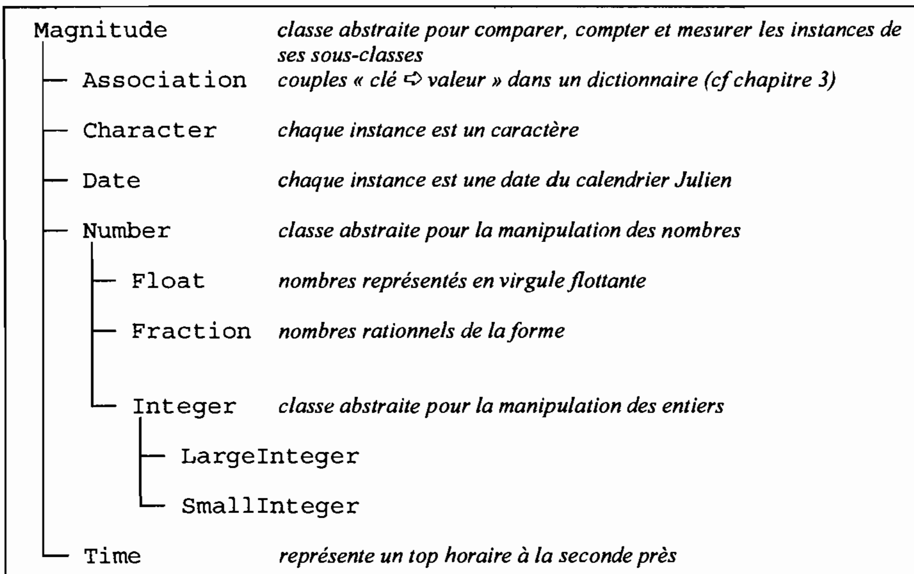


Figure 1.8 : La classe *Magnitude* et sa descendance

En écrivant précédemment que la classe *Magnitude* est une classe abstraite, nous indiquons que cette classe n'a en principe aucune instance : seules celles de ses sous-classes qui ne sont pas des classes abstraites pourront être instanciées. Une classe abstraite a pour rôle de « mettre en facteur » les traitements génériques qui sont les mêmes pour toutes ses sous-classes. Elle n'est pas instanciable parce qu'elle n'a pas suffisamment de propriétés pour que l'on puisse caractériser avec suffisamment de précision les objets qui relèveraient de cette classe. Ainsi, la classe *Number* est abstraite, parce qu'on ne pourrait décider de la représentation d'un objet *Number* (virgule fixe, virgule flottante, couple *numérateur et dénominateur* ? En revanche, elle définit, pour toutes ses sous-classes, la méthode *squared* :

```
squared
    "Answer the receiver, a type of Number, multiplied by itself."
    ^ self * self
```

Nous allons maintenant, dans le paragraphe suivant, étudier certains aspects du mécanisme de l'héritage en nous appuyant sur des exemples de la classe *Magnitude* et de ses sous-classes.

1.7 - Héritage et polymorphisme

Examinons la méthode *max:* de la classe *Magnitude*, telle qu'elle est présentée à la figure 1.9. Pour des objets sur lesquels une relation d'ordre est définie, cette méthode permet d'obtenir la plus grande valeur des deux objets x et y, à partir de l'évaluation de l'expression *x max: y* (l'objet *x* reçoit le message *max: y*).

```
max: aMagnitude
    "Answer the receiver if it is greater than aMagnitude, else answer aMagnitude."
    self > aMagnitude
        ifTrue:  [^self]
        ifFalse: [^aMagnitude]
```

Figure 1.9 : la méthode *max:* de la classe *Magnitude*

Cette méthode est représentée par un sélecteur à mot-clé et, à ce titre, elle correspond à un message qui sera envoyé avec un argument, cet argument apparaissant immédiatement après le mot-clé. Par exemple, l'expression *3.5 max: 5* représente le message *max:* qui est envoyé avec l'argument *5* à l'objet *3.5*. Dans la définition de la méthode donnée à la figure 1.9, l'argument du message est représenté par *aMagnitude*.

L'exécution de la méthode compare d'abord l'objet-récepteur à l'argument, en évaluant le prédicat : *self > aMagnitude*. Cette comparaison renvoie l'objet *true* ou l'objet *false*. Si le prédicat est vrai, le bloc[8] [^ *self*] est exécuté et la méthode renvoie l'objet récepteur qui est effectivement supérieur à l'argument du message. Sinon, la méthode renvoie l'argument *aMagnitude*. Le mécanisme de choix représenté par l'envoi du message à mots-clés *ifTrue:ifFalse:* à un objet de la classe *True* ou de la classe *False* sera étudié au chapitre 2.

La méthode *max:* est héritée par toutes les sous-classes de la classe *Magnitude*, permettant ainsi à des objets aussi différents l'un de l'autre qu'une date et une fraction de réagir de manière comparable à la même forme de message. Nous mettons ici en évidence le *polymorphisme* de la méthode *max:*. Le polymorphisme d'une méthode est la caractéristique qu'a cette méthode de provoquer des réactions différentes en fonction de la classe des objets auxquels elle s'applique. En 1.3, le polymorphisme de *reverse* est obtenu par la définition, pour chacune des deux classes *String* et *Integer* qui accepteront le message, d'un *protocole*[9] différent. Il est important de noter qu'avec la programmation orientée objets, le comportement d'un objet *O* est de la « responsabilité » de *O* : c'est dans la classe de *O* que l'on définira le comportement des objets tels que *O*, alors qu'en programmation classique, ce sont les procédures et les fonctions qui spécifient le traitement.

Cette encapsulation du traitement à l'intérieur de l'objet est une garantie pour le programmeur : tout objet recevant un message, pour lequel il ne dispose pas de méthode de traitement, réagira immédiatement avec une indication d'erreur. En revanche, dans un langage comme C, l'exécution d'une fonction, à laquelle on a fourni un paramètre incorrect, laissera le programme se dérouler jusqu'à ce que les conséquences de l'erreur initiale entraînent une autre anomalie qui, elle, sera bloquante mais n'aura souvent que très peu de rapports avec le paramètre incorrect et sera donc difficile à expliquer.

Si nous revenons à la méthode *max:*, le polymorphisme est très large puisqu'il s'applique aux dix sous-classes de *Magnitude* et, à la différence de *reverse*, il est

[8] Un bloc est une expression ou une suite d'expressions encadrée par des crochets (cf 2.10).

[9] On appelle protocole la description d'une méthode, exprimée dans le langage Smalltalk.

obtenu par l'effet de l'héritage, mettant ainsi en évidence un des intérêts de ce concept. En Pascal, pour obtenir le même résultat, il aurait fallu définir et décrire autant de fonctions `max(x,y:T)`, qu'il y a de types `T` susceptibles de permettre le classement de deux valeurs du type.

Remarquons également que le polymorphisme correspond bien à une caractéristique du raisonnement humain qui est souvent conduit à procéder par analogie pour appliquer un même concept à des situations différentes. Ainsi dirons-nous par exemple que le *10 mars 1989* est entre le *2 mars 1989* et le *30 juin 1990*, que la lettre *d* est entre les lettres *a* et *f*. De la même manière, la méthode *between:and:*, de la classe *Magnitude* (cf figure 1.10), pourra être utilisée avec la classe *Date* ou la classe *Character*, sous-classes de *Magnitude*.

```
between: min and: max
    "Answer true if the receiver is greater than or equal to min and less than or equal to max,
    else answer false."
    ^(min <= self) and: [self <= max]
```

Figure 1.10 : la méthode *between:and:* de la classe *Magnitude*

Notons enfin que le polymorphisme ne s'obtient pas avec un coup de baguette magique. Pour la méthode *max:* de la figure 1.9, il repose sur la réalisation correcte, pour chaque sous-classe de *Magnitude*, de la méthode correspondant au sélecteur > qui, elle aussi, témoignera de son polymorphisme mais en nous contraignant à la redéfinir pour bon nombre de ces sous-classes. Tout héritage de la méthode >, à partir de la classe *Magnitude* n'aurait en effet pas grande signification. Et Smalltalk le prévoit d'ailleurs puisqu'il décrit la méthode > de la classe *Magnitude* de la manière suivante :

```
> aMagnitude
    "Answer true if the receiver is greater than aMagnitude, else answer false."
    ^self implementedBySubclass
```

Si une sous-classe de *Magnitude* ne redéfinit pas cette méthode, l'envoi à un objet de cette sous-classe, d'un message correspondant au sélecteur > provoquera l'exécution de la méthode héritée, c'est-à-dire l'envoi, à ce même objet (*self*), du message *implementedBySubclass*. Ce message correspond à une méthode de la classe *Object* (donc héritée par tous les objets). Il provoque l'apparition d'une fenêtre de diagnostic indiquant que l'objet-récepteur aurait dû redéfinir la méthode.

L'exemple que nous venons de traiter illustre deux caractéristiques fondamentales du polymorphisme :

- le polymorphisme est un mécanisme *naturel* de la programmation orientée objets : chaque fois que pour toutes les classes descendantes d'une classe C, une méthode *m* peut être décrite par le même énoncé, le fait de définir *m* dans la classe C assure le polymorphisme. Ainsi, quelle que soit la sous-classe C' de *Magnitude* à laquelle appartiennent

les objets *x* et *y* le principe de l'évaluation de *x max:y* peut être exprimé par l'énoncé :

le plus grand de deux objets x et y est x, si x est supérieur à y, sinon c'est y.

- Ce mécanisme naturel doit être *construit.* Si, pour plusieurs classes C_1, C_2, ..., C_n, une méthode *m* s'exprime par le même énoncé, dans lequel on fait appel à une méthode *m'* et si *m'* ne peut être exprimée par un énoncé unique pour toutes les classes C_1, C_2, ..., C_n, il faut construire le polymorphisme de *m'* pour assurer celui de *m.* C'est-à-dire qu'il faut rédiger, pour *m'*, autant de protocoles différents qu'il y a d'énoncés distincts pour exprimer cette méthode.

1.8 - Les outils de Smalltalk pour la manipulation des classes

Après cette première approche de la programmation orientée objets, le lecteur qui désire se familiariser avec l'utilisation de Smalltalk peut, s'il dispose d'un poste de travail sur lequel le logiciel correspondant est disponible, effectuer ses premières manipulations. Pour lui faciliter le travail, nous indiquons, dans ce paragraphe, comment il est possible d'explorer et de manipuler l'arborescence des classes.

Smalltalk fournit, pour cela, une interface permettant de modifier une méthode d'une classe donnée, d'ajouter une méthode à une classe donnée, de créer une nouvelle classe en la définissant comme sous-classe d'une classe donnée. Pour ce faire, il est nécessaire d'ouvrir à l'écran une *fenêtre d'édition* de la hiérarchie des classes (*class hierarchy browser*). Pour faire apparaître une telle fenêtre, on sélectionnera, dans le menu général de Smalltalk, l'option *browse classes*[10].

Selon les implémentations, la présentation d'une fenêtre d'édition de la hiérarchie des classes prendra diverses formes mais on retrouvera toujours les mêmes fonctionnalités pour la consultation et la mise à jour de l'ensemble des classes. Nous présentons ici, aux figures 1.11 et 1.12, deux états successifs d'une telle fenêtre avec IBM Smalltalk.

La fenêtre de la figure 1.11 comprend, sous sa barre de titre, plusieurs panneaux. Le panneau supérieur gauche est le *panneau des classes.* Il donne une vue partielle de l'arbre des classes. En cliquant, avec le bouton gauche, sur le nom d'une classe, on sélectionne cette classe, faisant ainsi apparaître, dans les autres panneaux, certaines caractéristiques de cette classe. La figure 1.11 montre l'état de la fenêtre d'édition des classes quand on a fait défiler l'arbre jusqu'à la classe *Magnitude*, puis sélectionné cette classe.

Quand une classe vient d'être sélectionnée, le panneau inférieur de la fenêtre donne la définition de la classe, en énumérant notamment, s'il y a lieu, ses variables d'instances (cf. 2.6). Dans l'exemple de la figure 1.11, la définition de la classe *Magnitude* indique que cette classe est une classe-fille de la classe Object et qu'elle n'a pas de variables d'instances. Le panneau inférieur de la fenêtre est le

10 Les moyens de faire apparaître ce menu général dépendent des implémentations de Smalltalk.

panneau d'édition. Ce panneau est géré par un objet *éditeur de texte* qui permettra à l'utilisateur de modifier les descriptions de classes ou les méthodes qui y seront affichées.

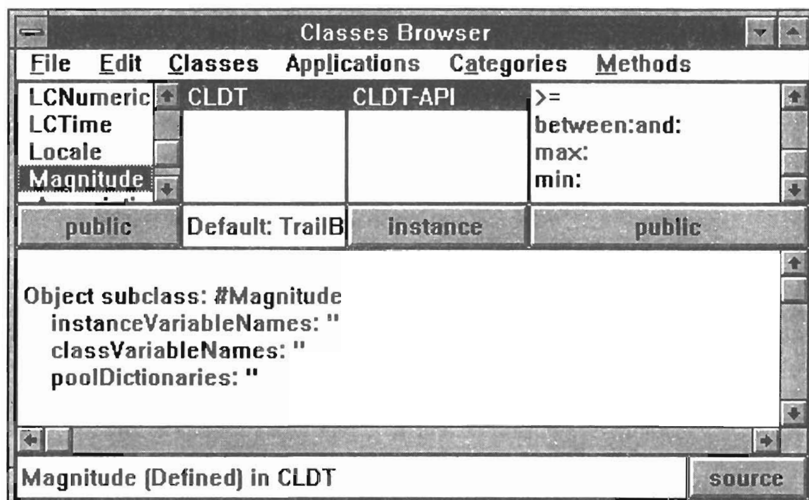


Figure 1.11 : la fenêtre d'édition des classes avec sélection de la classe *Magnitude*

Dans le panneau supérieur droit de la fenêtre, est affichée la liste des sélecteurs des méthodes d'instance de la classe Magnitude. On peut faire défiler cette liste et sélectionner l'une de ces méthodes, en cliquant sur son sélecteur. On peut aussi faire apparaître la liste de méthodes de classes en cliquant sur la barre de libellé *instance* qui sera alors remplacé par le libellé *class*.

Quand un sélecteur est sélectionné dans le panneau des sélecteurs, le protocole de la méthode correspondante apparaît dans le panneau d'édition. La figure 1.12 montre l'état de la fenêtre d'édition des classes ouverte sur la descendance de *Magnitude*, dans laquelle on a sélectionné la classe *Number* et, pour cette classe, la méthode *even*. Quand une méthode est affichée dans le panneau d'édition, l'utilisateur peut alors examiner le texte de la méthode et, éventuellement le modifier. Pour valider les modifications, l'utilisateur fera apparaître le menu du panneau d'édition, en cliquant brièvement sur le bouton droit de la souris, le curseur-souris étant placé dans le panneau d'édition. Il sélectionnera ensuite l'option *save* de ce menu.

D'une manière générale, cliquer brièvement avec le bouton droit de la souris, fait apparaître, dans le panneau actif[11], le menu de ce panneau (s'il en existe un). Le panneau des classes dispose lui aussi d'un menu que l'on peut faire apparaître de la même manière. Une option du menu du panneau des classes, l'option *add subclass*, permet d'ajouter une sous-classe à la classe sélectionnée. L'activation de

11 Le panneau actif est celui dans lequel est placé le curseur de la souris.

cette option déclenche un dialogue dans lequel Smalltalk demande à l'utilisateur d'indiquer les caractéristiques de la nouvelle classe (cf. chapitre 3).

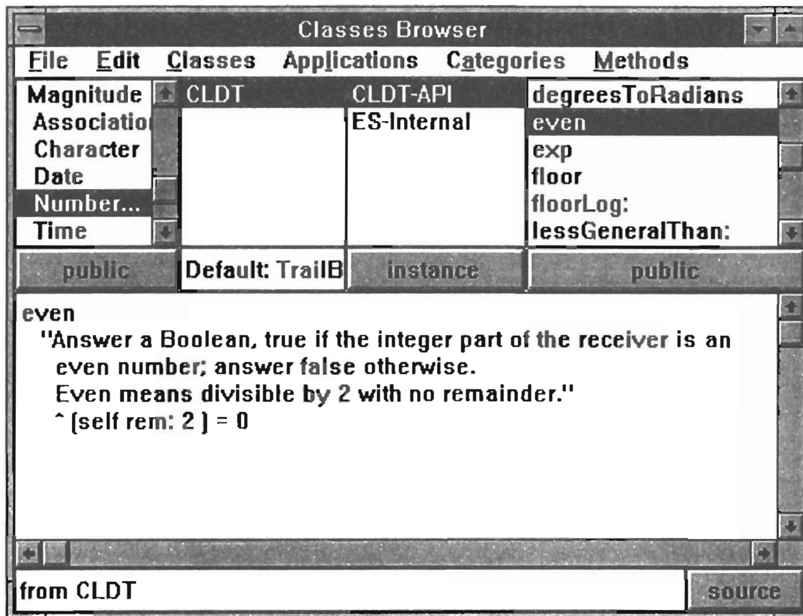


Figure 1.12 : une vue partielle sur la descendance de la classe *Magnitude*

Pour terminer, décrivons brièvement les options du menu du panneau des sélecteurs qui comprend quatre options : *delete methods*, *new method template*, *browse implementors* et *browse senders*.

delete methods

cette option permet de supprimer la méthode sélectionnée (pour la classe considérée, cette méthode disparaît de l'environnement de Smalltalk).

new method template

quand on active cette option, le panneau d'édition devient actif pour la saisie d'une nouvelle méthode (que l'on ajoutera à l'environnement de Smalltalk en utilisant l'option *save* du menu du panneau d'édition).

browse implementors

quand un sélecteur est sélectionné dans le panneau des sélecteurs, l'activation de cette option ouvre une nouvelle fenêtre qui donnera la liste des classes qui ont défini une méthode correspondant à ce même sélecteur. Cette option permet donc d'avoir une vision globale du polymorphisme d'une méthode.

browse senders

quand un sélecteur est sélectionné dans le panneau des sélecteurs, l'activation de cette option ouvre une nouvelle fenêtre qui indique les

méthodes qui font appel à la méthode indiquée par le sélecteur. Cette fenêtre indique les classes concernées et, pour chaque classe, les sélecteurs des méthodes qui font appel à la méthode sélectionnée. Cette option est donc utile, chaque fois que l'on souhaite modifier une méthode. En l'utilisant, on obtient la liste des autres méthodes (et des classes correspondantes) qui peuvent être touchées par la modification.

Avis au lecteur

Dans la suite de cet ouvrage, nous allons étudier en détail les caractéristiques de la programmation objets avec Smalltalk. Pour les chapitres 2 à 5, nous présenterons des exemples réalisés avec l'implémentation de Digitalk, Smalltalk V. Pour tous les thèmes abordés par ces chapitres, les trois versions les plus courantes de Smalltalk sont très proches et le lecteur n'aura aucune difficulté pour passer de Smalltalk V à Smalltalk 80 ou IBM Smalltalk. Dans le chapitre 6, nous étudions la programmation d'une application sous interface graphique, pour laquelle les différences entre les trois implémentations sont importantes. Pour ce dernier chapitre, nous avons choisi l'interface graphique MS-Windows et l'implémentation IBM Smalltalk.

Les quelques exemples que nous venons de présenter dans ce chapitre ont été composés avec la version 2 de IBM Smalltalk et, pour la site du livre, nous signalons dans le tableau suivant les quelques différences avec Smalltalk V, pour les notations utilisées dans le chapitre et reprises dans les chapitres suivants.

IBM Smalltalk	Smalltalk V
méthode *reverse*	méthode *reversed*
méthode *subStrings*	méthode *asArrayOfSubStrings*
option *Execute*	option *do it*
option *Display*	option *show it*

2 - Le langage Smalltalk

Nous allons maintenant préciser sur quel formalisme s'appuient les exemples que nous avons donnés dans le premier chapitre de cet ouvrage.

2.1 - Les expressions

Jusqu'à présent, nous avons manipulé des expressions sans indiquer quelles étaient les règles qui permettaient de former ces expressions. En Smalltalk, une expression peut-être :

- une constante littérale comme *123* ou *'Durand'*,
- un nom de variable comme *x* ou *chaine*,
- un message comme *chaine reversed* ou *x between: 5 and: y*,
- un bloc rassemblant, entre crochets, des déclarations facultatives de variables de blocs suivies d'une séquence d'expressions séparées les unes des autres par des points :

```
[:lettre|
 lettre asUpperCase printOn: Transcript.
 Transcript cr]
```

Il importe de noter qu'en Smalltalk, tout est objet : ainsi, une expression représente toujours un objet qui est le résultat de l'évaluation de cette même expression. Décrivons maintenant de manière précise les formes d'expressions que nous venons de citer.

2.2 - Les constantes littérales

Ce sont des nombres (instances d'une sous-classe de *Number*), des chaînes (instance de la classe *String*), des caractères (instances de la classe *Character*), des symboles (instances de la classe *Symbol*) ou des tableaux (instances de la classe *Array*) représentés directement par leur valeur.

2.2.1 - Les constantes numériques

Une constante numérique peut être un objet de la classe *Integer*, *Float* ou *Fraction* :

```
32  -120   1.414  15e-5  3/4
```

La classe *Fraction* (cf 2.6) représente les nombres rationnels non entiers et, à chaque instance de cette classe, est associé un couple de deux valeurs entières : numérateur et dénominateur. Une instance de cette classe est créée par l'envoi, à un entier *e1*, du message */ e2* (à condition que l'entier *e2* ne soit pas un diviseur de *e1*). Une constante de la classe *Fraction* est donc représentée par deux constantes entières, séparées par le sélecteur /.

2.2.2 - Les constantes-chaînes

Une constante-chaîne est une suite de caractères encadrée par deux apostrophes. Si l'un des caractères est une apostrophe, celle-ci doit être redoublée pour ne pas être confondue avec l'apostrophe de fin de chaîne. Une constante-chaîne est une instance de la classe *String*.

2.2.3 - Les constantes-caractères

Une constante-caractère est une instance de la classe *Character* que l'on représente par le caractère *$* suivi par le caractère :

$a représente le a minuscule
$! représente un point d'exclamation
$$ représente le caractère $
*$* représente l'espace ($ suivi d'un blanc).

On notera que l'utilisation du symbole *$* sert à bien distinguer une instance de la classe *Character* d'une instance de la classe *String* ne comportant qu'un seul caractère. Ainsi, *$a* et *'a'* sont deux objets différents.

2.2.4 - Symboles et constantes-symboles

Un symbole est un objet de la classe *Symbol*, elle-même sous-classe de la classe *String*. La principale différence entre un symbole et une chaîne, instance de la classe *String*, est que le symbole est invariable alors que les caractères d'une chaîne peuvent être modifiés. Ainsi, la séquence

```
|chaine|  chaine := 'abcde'.
          chaine at: 2 put: $Z.
```

modifie la première valeur de l'objet *chaine* et se termine avec, pour cet objet, la valeur *'aZcde'*. A l'inverse, un symbole est une chaîne invariable de caractères qui représentera la plupart du temps un identificateur (nom de classe, sélecteur de méthode, etc...). Il s'écrira sans être encadré par des apostrophes (ce qui permet de distinguer un symbole d'une instance de la classe *String*).

Quand un symbole est présent dans une expression, Smalltalk considère qu'il représente l'objet qu'il désigne. Par exemple, l'expression *String new* est formée de deux symboles :

String qui est le nom de la classe String,
new qui est le sélecteur de la méthode *new*.

L'évaluation de cette expression amènera Smalltalk à envoyer à la classe *String* le message *new* (et donc à créer une nouvelle instance de cette classe).

Si l'on souhaite que Smalltalk traite un symbole comme une instance de la classe *Symbol* (et non comme l'objet que ce symbole représente), il faut exprimer ce symbole sous la forme d'une *constante-symbole* qui s'écrira avec le caractère # précédant immédiatement le premier caractère du symbole. Pour illustrer la différence de traitement, considérons la séquence d'expressions suivante :

```
|s1 s2 chaine|
s1 := String.
s2 := #String.
chaine := s1 new
```

Après l'évaluation de la deuxième expression les variables *s1* et *s2* représentent deux objets distincts : *s1* désigne la classe *String* et *s2* désigne le symbole *String*. En fin d'évaluation la variable *chaine* représente une chaîne vide (résultat de l'envoi du message *new* à la classe *String*).

2.2.5 - Les tableaux

Un tableau est un objet de la classe *Array* (cf chapitre 3). Une *constante-tableau* s'écrit en énumérant les éléments du tableau et en encadrant l'énumération par un couple de parenthèses, la parenthèse gauche étant précédée par le caractère #. Voici quelques exemples de constantes-tableau :

#(10 -3 28) tableau de trois éléments dont les valeurs sont *10*, *-3* et *28*.

#('Irène' 'Durand' 1945 'Paris')
tableau de quatre éléments dont les trois valeurs de rang 1, 2 et 4 sont des chaînes et la valeur de rang 3 est l'entier 1945.

#(22 ($A $B $C) 3 57)
tableau de quatre éléments, dont le second est lui-même un tableau de trois caractères.

A propos du dernier exemple, on notera que la syntaxe de Smalltalk impose, lorsqu'un élément d'une constante-tableau est lui-même une constante-tableau, d'omettre le caractère #. En effet, la syntaxe d'une constante-tableau est définie par les règles de la page suivante[1].

1 cf [Dig 86] pour les conventions de la présentation syntaxique (Smalltalk syntax, page 454).

```
<constanteTableau> ::= "#" <tableau>
<tableau> ::= "(" {<constante numérique>
                   | <constante-chaîne>
                   | <symbole>
                   | <tableau>
                   | <constante-caractère>
                  } ")"
```

2.3 - Les séquences d'expressions

Une séquence d'expressions (cf 1.2) est une suite d'expressions séparées les unes des autres par des points. La dernière expression de la séquence n'est donc pas suivie d'un point. Quand une séquence d'expressions représente la description d'une méthode, il est fréquent que la dernière expression soit précédée par le signe ^ qui représente la fin d'exécution de la méthode avec le renvoi de la valeur de l'expression qui suit le ^.

Plus généralement, le signe de renvoi ^ peut apparaître devant n'importe quelle expression d'une séquence. Si cette expression est évaluée, sa valeur est renvoyée en réponse et l'évaluation de la séquence est terminée. Examinons, par exemple, la figure 2.1, qui donne une réalisation possible de la méthode *quo:* de la classe *Integer*[2]. Si *entier* est un objet de cette classe et si *nombre* est une valeur numérique, l'évaluation de l'expression

```
entier quo: nombre
```

renvoie la partie entière du quotient de *entier* par *nombre*.

```
quo: aNumber
  "Answer the integer quotient of the receiver divided by aNumber
   with truncation toward zero."
  aNumber = 0 ifTrue: [^self zeroDivisor].
  ^(self / aNumber) truncated
```

Figure 2.1 : une réalisation de la méthode *quo:* pour la classe *Integer*

La séquence de la figure est une série de deux expressions. La première expression est une évaluation conditionnelle : si l'argument *aNumber* de la méthode est nul, le bloc qui suit le sélecteur *ifTrue:* est exécuté. Dans ce cas, le message *zeroDivisor* est envoyé à l'objet-récepteur et la réponse obtenue constituera la réponse de la méthode : l'exécution est alors terminée avant l'évaluation de la dernière expression. Le traitement du message *zeroDivisor* provoquera l'apparition d'une fenêtre de diagnostic qui indiquera l'erreur rencontrée.

Si *aNumber* n'est pas nul, le bloc qui suit *ifTrue:* n'est pas exécuté et l'évaluation de la dernière expression renvoie le résultat attendu.

---

2 La représentation donnée à la figure ne correspond pas exactement à la réalisation en Smalltalk V qui utilise, pour cette méthode, une primitive en langage-machine.

2.4. - Les règles de priorité pour l'évaluation d'une expression

Nous avons vu, au paragraphe 2.1 qu'une expression pouvait être un message. Nous avons, en 1.1, distingué plusieurs catégories de messages. Rappelons-les brièvement.

2.4.1 - Messages unaires

Un message unaire est un message sans argument, qui se représente par le sélecteur de la méthode correspondante :

```
'Alain' size  "la chaîne Alain reçoit le message unaire size"
x reversed    "l'objet représenté par la variable x reçoit le message unaire reversed"
```

2.4.2 - Messages binaires

Un message binaire comprend un et un seul argument qui suit immédiatement le sélecteur :

```
5 + 3         "l'objet 5 reçoit le message binaire + 3 (la réponse est l'objet 8)"

18 @ 24       "l'objet 18 reçoit le message binaire @ 24 et renvoie le point de
               coordonnées 18 et 24"

#(2 3) , #('a' 'b')
              "l'objet #(2 3) reçoit le message , #('a' 'b') (sélecteur « virgule ») et renvoie
               le tableau #(2 3 'a' 'b')"
```

On notera que le dernier message binaire des exemples précédents correspond au sélecteur , (virgule)[3] qui représente la concaténation de la valeur du récepteur avec celle de l'argument.

2.4.3 - Messages à mot(s)-clé(s)

Un message à mot(s)-clé(s) est un message qui comprend un ou plusieurs arguments. Chaque argument est précédé d'un mot-clé qui se termine obligatoirement par le signe : (deux points). Le sélecteur du message est la concaténation des mots-clés :

```
5 max: 10               "sélecteur max:"
x between: y and: z     "sélecteur between:and:"
```

---

3 Ce sélecteur correspond à une méthode d'instance définie pour les classes IndexedCollection et SortedCollection (cf chapitre 4) et pour toutes les sous-classes de ces classes. L'évaluation de l'expression a,b où a et b sont des objets de la classe C renvoie un nouvel objet de C, formé de la valeur de a, suivie de celle de b.

2.4.4 - Les priorités

Quand une expression comprend plusieurs messages différents, Smalltalk applique les règles de priorité suivantes :

- Les messages unaires ont priorité sur les messages binaires, qui ont eux-mêmes priorité sur les messages à mots-clés.
- Pour un même niveau de priorité, l'évaluation se fait de gauche à droite.
- L'utilisation de parenthèses permet de modifier les priorités : les messages entre parenthèses sont évalués d'abord.

La figure 2.2 présente quelques exemples qui mettent en relief ces règles.

```
1 + 2 * 3                "renvoie 9"
1 + (2 * 3)              "renvoie 7"
3 + 'abcdef' size        "évaluation de 'abcdef' size qui renvoie 6,
                          puis de 3 + 6 qui renvoie 9"
```

Figure 2.2 : évaluation et priorités

A propos des opérations arithmétiques, on notera, comme le montre le premier des trois exemples précédents, que les règles de priorité de Smalltalk ne sont pas celles des langages de programmation classiques pour les mêmes opérations. Il appartiendra donc au programmeur de veiller, en Smalltalk, à l'utilisation de parenthèses pour faire exécuter, en priorité, un calcul figurant, dans une expression, à droite d'un autre calcul moins prioritaire.

Egalement à titre d'exemple, notons que, dans l'expression finale, présentée à la figure 2.1 :

```
^(self / aNumber) truncated
```

l'utilisation des parenthèses est obligatoire pour déclencher l'évaluation du message binaire / avant celle du message unaire *truncated.*

Quand un même objet doit recevoir successivement plusieurs messages, on peut écrire une « *cascade* » de messages dans laquelle chaque message est séparé du suivant par un point-virgule :

```
Transcript nextPutAll: 'Alain';
           cr;
           nextPutAll: 'Durand';
           positionAtBeginning
```

Dans l'exemple précédent, l'objet *Transcript* reçoit successivement quatre messages. Le premier affiche le prénom *'Alain'*, le second envoie le curseur en début de ligne suivante, le troisième affiche le nom *'Durand'* et le dernier place le curseur au début du texte contenu dans *Transcript.*

2.5. - Les variables

Une variable Smalltalk représente un objet. De manière interne, elle contient un pointeur sur cet objet, sauf si l'objet peut être représenté par un octet ou un mot-machine : dans ce cas, l'objet est stocké directement dans la variable. Quand une variable est définie, mais qu'elle n'a pas encore reçu de valeur, elle a la valeur *nil*. La valeur *nil* est elle-même un objet[4] qui est la seule instance de la classe *UndefinedObject*. On peut vérifier cette affirmation en évaluant, avec *show it*[5], l'expression

```
|x| x
```

qui rend la valeur *nil*. Deux méthodes de la classe *Object* permettent de savoir si une variable a déjà reçu une valeur. Ce sont les méthodes *isNil* et *notNil*.

Il existe, en Smalltalk, trois catégories de variables : les variables d'instance, les variables temporaires et les variables partagées. La syntaxe impose d'attribuer aux variables d'instance et aux variables temporaires des noms commençant par une lettre minuscule et aux variables partagées des noms commençant par une majuscule. Nous allons, dans les paragraphes suivants, étudier les trois catégories de variables.

2.6 - Les variables d'instance

Les variables d'instance sont des variables associées à un objet et leur durée de vie est celle de cet objet. Elles représentent la connaissance statique privée de cet objet (les méthodes d'instance représentant, elles, la connaissance dynamique). Examinons par exemple la figure 2.3, qui reproduit la définition de la classe *Fraction*.

```
Number subclass: #Fraction
  instanceVariableNames:
    'numerator denominator '
  classVariableNames: ''
  poolDictionaries: ''
```

Figure 2.3 : La classe *Fraction*

Cette définition indique que la classe dont le nom est le symbole *#Fraction* est une sous-classe de la classe *Number* et que chaque objet de cette classe possède

4 Ce qui permet d'affirmer la généralité de la première phrase du paragraphe 2.5. Même quand elle n'a pas encore reçu de valeur, une variable désigne un objet.

5 Nous avons vu au chapitre 1 (cf 1.3) que l'on pouvait, dans une fenêtre d'édition de texte, sélectionner une séquence d'expressions puis déclencher l'évaluation de la séquence sélectionnée avec l'option do it ou l'option show it. Ici, pour l'évaluation de |x| x, il faut utiliser show it. Une évaluation avec do it aurait bien rendu la valeur nil mais ne l'aurait pas affichée dans la fenêtre.

deux variables d'instance respectivement appelées *numerator* et *denominator*. Si un objet de cette classe représente le rationnel *n/d*, la variable d'instance *numerator* aura la valeur *n* et la variable d'instance *denominator* aura la valeur *d*. On peut créer un objet de classe *Fraction* soit avec la méthode de classe *new* héritée de la classe-mère, soit avec la méthode de classe *numerator:denominator:* qui donne une valeur déterminée aux variables d'instance, comme le montre la séquence de la figure 2.4.

```
|f1 f2|
f1 := Fraction new.
   "les variables d'instance numerator et denominator de la fraction f1 ont la valeur nil"

f2 := Fraction numerator: 3 denominator: 5.
   "les variables d'instance de f2 ont les valeurs 3 et 5"

f2 := f2 * (2/7)
   "f2 prend la valeur 6/35"
```

Figure 2.4 : quelques instanciations de la classe *Fraction*

Il est important de noter que chaque objet de la classe *Fraction* possède ses propres variables d'instances qui ne sont accessibles que par les messages qui lui sont adressés. Les variables d'instance d'un objet *x* ne peuvent être modifiées qu'avec l'accord de ce même objet : seule une méthode propre à *x* peut changer la valeur d'une variable d'instance de *x*.

Pour illustrer cette caractéristique de la programmation orientée « objets », étudions l'évaluation de la dernière expression de la figure 2.4 :

```
f2 := f2 * (2/7)
```

Au moment de l'évaluation de cette expression, Smalltalk manipule deux variables d'instance *numerator* (celle de *f2* et celle de *2/7*) et deux variables d'instance *denominator*. Pour bien comprendre comment sont traitées ces variables, examinons les méthodes de la figure 2.5. Trois méthodes d'instance de la classe *Fraction* y sont décrites. Les deux premières permettent d'accéder aux valeurs des variables d'instance d'un objet de cette classe. En effet, une des caractéristiques de la programmation orientée « objets » est de donner aux objets la responsabilité de la gestion des informations (données et méthodes) dont ils sont propriétaires et ceci dans le but d'une plus grande sécurité en programmation. Ainsi, la plupart des opérations incorrectes sont immédiatement signalées et diagnostiquées par les objets qui possèdent les informations mises en danger par ces opérations incorrectes[6]. Dans cet esprit, pour accéder à la connaissance privée d'une fraction (c'est-à-dire à son numérateur et à son dénominateur), il est nécessaire d'envoyer des messages à cette fraction. Pour la commodité de la compréhension, ces messages portent ici le même nom que les variables correspondantes.

---

6 A la différence de la programmation classique, avec laquelle une procédure ou une fonction peut, par erreur, endommager une donnée sans que le traitement soit interrompu.

```
numerator
    "Answer the numerator of the receiver."
    ^numerator

denominator
    "Answer the denominator of the receiver."
    ^denominator

* aNumber
    "Answer the result of multiplying the receiver by aNumber."
    ^(numerator * aNumber numerator)
      /
     (denominator * aNumber denominator)
```

Figure 2.5 : quelques méthodes d'instances de la classe *Fraction*

Dans la figure 2.5, la réalisation de la méthode *, qui reçoit l'argument *aNumber*, illustre ce que nous venons de dire. Si, dans l'expression $x*y$, x représente une fraction, lorsque l'expression est évaluée, la méthode * de la classe *Fraction* est exécutée. Alors, seule la variable *numerator* de l'objet-récepteur x est directement accessible. La valeur de la même variable d'instance de y devra être obtenue par le message *numerator* envoyé à y. Ainsi, la première partie du calcul effectué par la méthode *, pour multiplier les numérateurs de x et y, s'écrit-elle :

```
(numerator * aNumber numerator)
```

Remarquons enfin que cette manière de procéder assure le polymorphisme de la méthode * de la classe *Fraction.* En effet, la responsabilité de fournir son numérateur et son dénominateur incombant à chaque objet, si l'argument *aNumber* n'est pas une fraction mais un objet de la classe *Float* ou d'une sous-classe de la classe *Integer*, cet argument renverra les valeurs correctes grâce aux méthodes héritées de la classe-mère *Number*, que nous présentons à la figure 2.6.

```
numerator
  "Answer the numerator of the receiver.
   Default is the receiver which can be overridden by the subclasses."
  ^self

denominator
   "Answer the denominator of the receiver.
    Default is one which can be overridden by the subclasses."
   ^1
```

Figure 2.6 : les méthodes *numerator* et *denominator* de la classe *Number*

Dans les exemples que nous venons d'étudier, les variables d'instances *numerator* et *denominator* sont des variables d'instances *nommées*, c'est-à-dire désignées par leur nom (en anglais : *named instance variables*). Un objet peut

aussi avoir des variables d'instances *indexées* (*indexed instance variables*) qui sont repérées par un indice entier et ne sont accessibles qu'à travers un message adressé à l'objet.

Ainsi, un objet de la classe *Array* est un tableau dont les variables d'instances sont indexées et représentent les éléments du tableau. De même, un objet de la classe *String* possède autant de variables d'instance indexées que sa valeur comporte de caractères. Par exemple, l'évaluation de la séquence suivante :

```
|initiale|
initiale := 'Marie' at: 1
```

affectera, à la variable *initiale*, la valeur *$M*.

Pour une même classe, le nombre de variables d'instance nommées est le même pour tous les objets de la classe. En revanche, le nombre des variables d'instances peut varier d'un objet à l'autre. Ainsi, pour la classe *Array*, ce nombre est, pour chaque objet, égal au nombre d'éléments du tableau représenté.

Les méthodes *at:* et *at:put:*, qui permettent respectivement d'obtenir et de modifier la valeur d'une variable d'instance indexée sont définies dans la classe *Object* et sont donc héritées par toutes les classes. Dans ces deux méthodes, la valeur de l'argument correspondant au mot-clé *at:* représente le rang (supérieur ou égal à 1) de la variable d'instance que l'on souhaite examiner ou modifier. Si l'objet récepteur n'a pas de variables d'instances ou si le rang indiqué correspond à une variable inexistante, une fenêtre d'erreur apparaît. La séquence suivante illustre le fonctionnement de ces deux méthodes.

```
|x|
x := Array new: 5.
   "x est un tableau de cinq éléments qui ont tous la valeur nil"
x at: 3 put: 1990.
   "l'élément de rang 3 prend la valeur 1990"
x at:1 printOn: Transcript.
   "la valeur nil apparaît sur la fenêtre Transcript"
x at: 8 printOn: Transcript
   "une fenêtre de diagnostic apparaît car x n'a pas de variable d'instance de rang 8"
```

2.7 - Gestion dynamique de la mémoire disponible

Pour gérer les objets qu'il est amené à manipuler, le processeur Smalltalk va utiliser un espace-mémoire dans lequel il distinguera des domaines disponibles et des domaines occupés par la représentation des objets utilisés. Chaque nouvel objet créé restreint l'espace disponible au profit de l'espace occupé. Pour comprendre cette gestion de l'espace-mémoire, étudions l'évaluation de la séquence de la figure 2.7.

```
|x| x := Array new: 6.
    x at: 1 put: 'Alain'.
    x at: 5 put: 1945.
    x printOn: Transcript
```

Figure 2.7 : création d'une instance de la classe *Array*

Pour Smalltalk, la première opération de l'évaluation est le traitement de la définition de la variable temporaire *x* (cf 2.8). Une fraction de l'espace disponible va être réservée pour y implanter la représentation de x. La valeur de *x* après cette implantation est la valeur *nil.*

En fait, dans l'espace disponible de Smalltalk, le résultat de l'implantation de *x* est représenté par la figure 2.8. La valeur *nil* est elle-même un objet, unique instance de la classe *UndefinedObject* et la variable *x* a pour valeur interne un *pointeur* qui indique l'adresse de l'emplacement-mémoire qui contient l'objet *nil.* Le programmeur, cependant, ne manipulera jamais ce pointeur : pour lui, tout se passe comme si la valeur de *x* était directement l'objet *nil.*

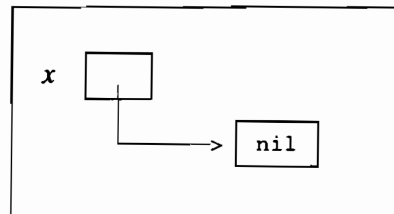


Figure 2.8 : la variable temporaire x après son implantation en mémoire

Ensuite, l'évaluation de la première affectation : `x := Array new: 10` s'effectue en deux temps. Smalltalk évalue d'abord le message *new: 6* adressé à la classe *Array* : il prélève, sur l'espace disponible, le domaine nécessaire pour la représentation d'un tableau de six éléments qui auront tous la valeur *nil*[7]. Smalltalk va ensuite réaliser l'affectation : la valeur de la variable *x* va être modifiée pour désigner le tableau nouvellement créé. Cette valeur sera celle d'un *pointeur* qui désignera l'emplacement du domaine contenant la représentation du tableau. Après cette affectation, la variable x représente donc le tableau créé.

Ensuite, l'évaluation de *x at: 1 put: 'Alain'* (message *at:put:* envoyé à l'objet *x*) va conduire Smalltalk à prélever à nouveau, sur l'espace disponible, le domaine nécessaire à la représentation de la chaîne *'Alain'.* Après cette opération, Smalltalk changera la valeur du premier élément du tableau désigné par *x* : la nouvelle valeur sera un pointeur qui indiquera l'emplacement de la mémoire où est implantée la représentation de la chaîne *'Alain'.*

De même, le traitement du second message *at:put:* envoyé à l'objet *x* modifiera la valeur du cinquième élément du tableau. La figure 2.9 donne une représentation des domaines de l'espace-mémoire utilisés après le traitement de ce second message.

Avant d'étudier comment se terminera l'exécution de la séquence présentée à la figure 2.7, revenons sur le terme de « pointeur », que nous avons utilisé à plusieurs reprises dans les explications précédentes. En programmation, un

---

7 C'est-à-dire qui contiendront tous un pointeur vers l'objet nil.

pointeur est, d'une manière générale, une entité qui désigne l'emplacement d'une information dans l'espace-mémoire occupé par un programme.

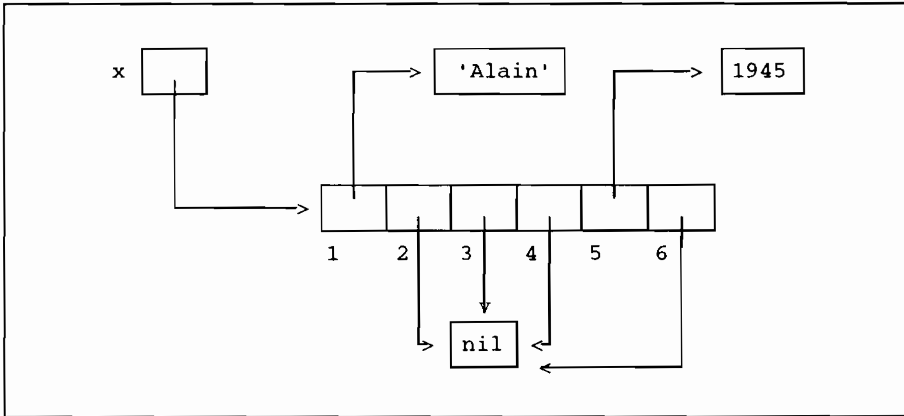


Figure 2.9 : exemple d'utilisation de l'espace-mémoire

A partir de l'exposé que nous venons de faire de l'utilisation de pointeurs par Smalltalk, le lecteur déjà familiarisé avec des langages de programmation classiques tels que Pascal ou C pourrait être amené à penser que la pratique de la programmation orientée « objets » nécessite, pour le programmeur, la manipulation de pointeurs. Il n'en est rien : la gestion de l'espace-mémoire est faite par Smalltalk sans que le programmeur ait à s'en soucier[8]. Et, si nous revenons à l'exemple de la figure 2.9, le programmeur n'a pas à se préoccuper de l'utilisation des pointeurs. Pour lui, l'objet *x* est le tableau représenté sur la figure et, par exemple, l'élément de rang 1 de ce tableau a pour valeur la chaîne *'Alain'*.

Revenons maintenant à la séquence examinée figure 2.7. La dernière expression évaluée :

```
x printOn: Transcript
```

enverra, dans la fenêtre Transcript, la représentation de l'objet *x*. Cette représentation apparaîtra de la manière suivante :

```
('Alain' nil nil nil 1945 nil)
```

Après l'évaluation de cette dernière expression, si l'on suppose que l'évaluation de la séquence de la 2.5 a été déclenchée avec *show it* ou *do it*, le processeur Smalltalk va supprimer la variable temporaire *x* et donc réincorporer à l'espace disponible l'objet que cette variable représentait. Ainsi, le tableau, la constante-chaîne *'Alain'*, la constante numérique *1945* disparaîtront de l'environnement

8 Bien entendu, la manière dont le processeur Smalltalk gèrera la mémoire disponible influera sur la performance des applications. Une gestion efficace impliquera que Smalltalk limite la fragmentation de la mémoire due au fait que les domaines réservés ne sont pas forcément rendus à l'espace disponible selon l'ordre chronologique des réservations (un domaine est rendu à l'espace disponible quand l'objet qu'il représente est détruit). En général, le processeur Smalltalk disposera d'un mécanisme interne de « ramasse-miettes » (en anglais garbage collection) qui permettra de rassembler les fragments disponibles, chaque fois que la grande dispersion de ces fragments pénalisera trop les performances.

Smalltalk. Des cinq objets représentés à la figure 15, seul l'objet *nil*, unique objet de la classe *UndefinedObject*, subsistera à la fin de l'évaluation de la séquence de la figure 2.7.

Ainsi, chaque fois qu'un objet est supprimé de l'environnement Smalltalk, toutes ses variables d'instance (qu'elles soient nommées ou indexées) sont également supprimées.

2.8 - Les variables temporaires

Nous connaissons déjà ces variables que nous avons souvent utilisées pour l'évaluation d'une expression. Ce sont des variables dont le nom commence toujours par une lettre minuscule et qui disparaissent de l'environnement de Smalltalk dès que la séquence dans laquelle elles ont été déclarées a été évaluée. Ces variables peuvent être classées en trois catégories.

2.8.1 Les variables temporaires d'une séquence d'expressions

Ce sont celles qui se déclarent obligatoirement en tête d'une séquence d'expressions, entre une paire de barres verticales (| et |). La séquence pour laquelle ces variables sont déclarées et utilisées peut être une séquence d'expressions saisie dans une fenêtre d'édition de texte ou encore le protocole complet d'une méthode. Quand on déclare une variable temporaire *v*, cette variable désigne, au départ, l'objet *nil*. Pour lui faire désigner un autre objet *o* (i.e. lui « donner une nouvelle valeur »), on doit utiliser une affectation et écrire :

```
v := o
```

Après l'évaluation de cette affectation, la variable *v* désigne l'objet *o* (cf 2.7) et l'on peut, par abus de langage, parler de l'« objet *v* ». On notera à ce propos, que nous rencontrons là l'unique exception à la règle qui veut que, dans Smalltalk tout soit « objet ». Une variable est un *pointeur* sur un objet et le symbole := n'est pas le sélecteur d'une méthode. Ainsi, comme un objet ne peut traiter que les messages qui correspondent à une méthode qu'il reconnaît, on ne peut faire une affectation directe à un objet[9]. Par exemple, si *o1* et *o2* sont deux objets, l'évaluation de la séquence

```
|x|  x := o1.  x := o2.
```

n'aura pas pour effet de « transformer » l'objet *o1* en l'objet *o2*. Dans cette évaluation, x aura désigné successivement l'objet *nil*, l'objet *o1* et enfin l'objet *o2*.

2.8.2 Les arguments d'un bloc

Un bloc est une séquence d'expressions encadrée par une paire de crochets [et]. Un bloc est un objet de la classe *Context* (cf 2.10) et peut comporter des *arguments* qui sont des variables locales déclarées au début du bloc. Un bloc est

9 Voir, à ce propos, au chapitre 3, les exemples donnés avec la classe String, paragraphe 3.3.

un objet syntaxique qui permet la construction de schémas itératifs ou conditionnels. Il contient toujours une séquence d'expressions dont il déclenchera l'évaluation chaque fois qu'il recevra un message correspondant à l'une des trois méthodes *value*, *value:* ou *value:value:*. Examinons par exemple la séquence de la figure 2.10 qui calcule, dans la variable *somme*, la somme des valeurs du tableau d'entiers *tableau*.

```
|tableau somme|
tableau := #(1 10 100 1000).
somme := 0.
tableau do: [:element| somme:= somme + element].
```

Figure 2.10 : exemple d'évaluation d'un bloc à un argument

A la dernière ligne de la séquence présentée à la figure 2.10, l'objet *tableau* reçoit un message de la forme *do: blocAvecUnArgument*. Le traitement d'un tel message déclenche une itération qui évalue *blocAvecUnArgument* pour chaque élément du tableau. Le bloc évalué est ici :

```
[:element | somme := somme + element]
```

Il comprend deux parties encadrées par `[` et `]` et séparées par une barre verticale. La première déclare l'argument *element* qui est une variable temporaire, locale au bloc, qui sera créée au début de l'évaluation du bloc et supprimée en fin d'évaluation. Si *o* est un objet et si le bloc reçoit le message *value: o*, la variable *element* est, lors de sa création, initialisée de manière à désigner l'objet *o*. Ainsi, le traitement du message *do:* de la dernière ligne de la figure 2.10 va amener l'objet *#(1 10 100 1000)* à envoyer au bloc successivement les quatre messages *value: 1*, *value: 10*, *value: 100* et *value: 1000* qui crééront puis supprimeront quatre fois la variable *element* pour lui faire désigner successivement les entiers *1*, *10*, *100* et *1000*. A la fin d'exécution de la séquence de la figure 2.10, la variable *somme* aura la valeur *1111*.

Un argument de bloc est donc une variable temporaire, locale au bloc dans lequel elle est déclarée. Sa déclaration doit figurer en tête du bloc entre les signes `[` et `|` et elle est immédiatement précédée par le caractère : (deux points). Elle a les mêmes caractéristiques que celles des variables temporaires décrites en 2.8.1 mais elle reste liée à l'objet qu'elle représente et qui est l'argument du message *value:* ou *value:value:*[10] envoyé au bloc. En d'autres termes, il n'est pas possible, dans un bloc avec argument, d'affecter directement une valeur à l'argument. Ainsi, un bloc dont la forme est *[:arg|...]* ne peut contenir une expression de la forme *arg := v*. En revanche, l'objet que représente l'argument de bloc peut être éventuellement modifié, si l'on envoie à cet argument (i.e. à cet objet) le message adéquat.

10 Sauf extension apportée par le programmeur au langage, Smalltalk V ne prévoit que des blocs à zéro, un ou deux arguments évaluables avec les messages value (pas d'argument), value: (un argument) ou value:value: (deux arguments).

2.8.3 Les arguments d'une méthode

Un argument d'une méthode *m* est une variable temporaire qui représentera un objet qui interviendra dans l'évaluation de *m* et qui est déclaré, dans l'en-tête de *m*, comme paramètre. Par exemple, dans l'évaluation de :

```
'Bonjour' printOn: Transcript
```

qui fera apparaître la chaîne *'Bonjour'* dans la fenêtre *System Transcript*, la méthode *printOn:*, de la classe *String*, est utilisée, conformément au protocole indiqué à la figure 2.11.

```
printOn: aStream
   "Append the receiver as a quoted string to aStream doubling
    all internal single quote characters."
   aStream nextPut: $'.
   self do:
      [:character |
       aStream nextPut: character.
       character = $' ifTrue: [aStream nextPut: character]
      ].
   aStream nextPut: $'
```

Figure 2.11 : la méthode *printOn:* de la classe *String*

Dans cette méthode, la variable *aStream* est l'argument qui représentera l'objet utilisé en paramètre d'un message *printOn:* envoyé à un objet de la classe *String*. L'objet représenté par l'argument d'une méthode peut être modifié s'il reçoit, au cours de l'exécution du protocole de cette méthode, un message approprié. C'est le cas, pour l'évaluation de *'Bonjour' printOn: Transcript:*. En effet, l'argument *aStream* correspond à la fenêtre *Transcript* et cette fenêtre est modifiée neuf fois, à chaque évaluation d'un message *nextPut:* qui lui est adressé.

On notera également que, comme l'argument d'un bloc, l'argument d'une méthode reste lié à l'objet-paramètre qui lui correspond : une affectation d'un nouvel objet à un tel argument est illicite. Enfin, on remarquera que la figure 2.11 offre également un exemple de l'évaluation d'un bloc avec un argument. En effet, l'objet-récepteur du message *printOn* (ici *'Bonjour'*) déclenchera, avec un message *do:* (cinquième ligne de la figure), une évaluation répétitive du bloc dont l'argument est la variable *character*. Dans cette évaluation répétitive, l'argument *character* prendra successivement les valeurs *$B, $o, $n, $j, $o, $u et $r*. Remarquons également que ce bloc contient lui-même un autre bloc : *[aStream nextPut: character]* qui est l'argument du message *ifTrue:* (cf 2.11.1).

2.9 - Les variables partagées

Les variables partagées sont des variables permanentes qui sont accessibles à plusieurs objets. A la différence des variables temporaires, leur noms commencent toujours par une majuscule. Pour certaines d'entre elles, qui sont définies par

l'utilisateur, la « durée de vie » est déterminée par ce même utilisateur, qui peut, à tout moment, les supprimer de l'environnement de Smalltalk.

D'autres variables partagées sont prédéfinies et ne peuvent être supprimées. Par exemple, la variable *Smalltalk* désigne un objet de la classe *Dictionary* qui répertorie la plupart des objets manipulés par Smalltalk et que nous étudierons au chapitre 3 (cf 3.5). Egalement, la variable *CharacterConstants* désigne un objet qui répertorie les symboles descriptifs qui peuvent être utilisés pour représenter des caractères spéciaux (*Bell*, *Cr*, *Tab*, *MouseButton*, etc, cf 2.9.2).

Les variables partagées peuvent être classées en trois catégories : variables globales, dictionnaires partagés (en Anglais, pool dictionaries) et variables de classe. Nous allons maintenant étudier chacune de ces catégories.

2.9.1 - Les variables globales

Une variable globale désigne un objet qui est accessible à tous les objets[11]. Par exemple, la variable *Transcript*, que nous avons déjà fréquemment utilisée, est un objet de la classe *TextEditor* dont la fenêtre (*System Transcript*) peut faire apparaître des informations envoyées par tout objet de l'environnement Smalltalk.

La variable globale *Transcript* est une variable prédéfinie dans l'environnement de Smalltalk. L'utilisateur peut aussi définir de nouvelles variables globales. Etudions par exemple l'évaluation de la séquence de la figure 2.12.

```
|t|
t := 25.
TauxTVA := #(5.5 18.6 33).
RepertoireCourantDeD := Directory new
                            drive: $D;
                            pathName: '\'.
TauxTVA at: 3 put: t.
```

Figure 2.12 : utilisation de variables globales

Après l'évaluation de l'expression de la deuxième ligne, la variable temporaire *t* désigne l'objet *25*. Pour évaluer la ligne suivante, Smalltalk va procéder en deux temps :

- Ayant constaté que *TauxTVA* ne peut être qu'une variable globale (car son nom commence par un majuscule) et que cette variable est encore inconnue, Smalltalk va interroger l'utilisateur, en utilisant un menu à deux options. Ce menu permet à l'utilisateur, soit de renoncer à la création de la variable globale[12], soit de confirmer cette création.
- Si nous supposons que, dans l'évaluation de la figure 2.12, l'utilisateur confirme la création de *TauxTVA*, Smalltalk réserve alors un emplacement pour cette variable puis l'initialise pour lui faire désigner un

11 Au contraire, les dictionnaires partagés et les variables de classe ne sont accessibles qu'à certaines catégories d'objets (cf 2.9.2 et 2.9.3).

12 Il le fera par exemple si la variable correspond à une variable temporaire dont il aura, par erreur, frappé l'initiale du nom en majuscule.

tableau de trois éléments (qui représentent trois des taux de TVA les plus utilisés en France en 1989).

La poursuite de l'évaluation de la séquence de la figure 2.12 va conduire Smalltalk à créer une nouvelle variable globale, *RépertoireCourantDeD*[13], qui représentera un objet de la classe *Directory* et désignera dans un environnement MS-DOS, le répertoire-racine du disque *D:*.

Enfin, l'évaluation de la dernière expression de la figure 2.12 modifiera le dernier élément de la variable globale *TauxTVA* pour remplacer le taux de TVA de 33% par la valeur de *t* (25%). A l'issue de l'exécution de la séquence complète, la variable temporaire *t* disparaîtra de l'environnement de Smalltalk mais les variables globales *TauxTVA* et *RepertoireCourantDeD* resteront disponibles. On notera que la disparition de la variable *t* n'entraînera pas de modification de la valeur du troisième élément de *TauxTVA*. En effet, l'évaluation de l'expression *TauxTVA at: 3 put: t* (cf figure 2.12) avait affecté au troisième élément du tableau *TauxTVA* l'objet représenté par *t* : cet élément pointe désormais vers l'entier *25*. La disparition de *t* supprime la valeur de *t*, c'est-à-dire le pointeur, présent dans *t*, vers l'entier *25*. Le pointeur vers ce même entier, présent dans le troisième élément de *TauxTVA*, lui, ne sera pas modifié. L'entier *25* restera présent dans l'environnement de Smalltalk tant qu'un pointeur le désignera.

Après l'évaluation de la séquence de la figure 2.12, les variables *TauxTVA* et *RépertoireCourantDeD* subsistent dans l'environnement de Smalltalk. On peut les modifier, par exemple en évaluant la séquence :

```
TauxTVA := #(5.5 14 18.6 25).
RepertoireCourantDeD pathName: '\Marie\Musique'
```

La première expression évaluée affectera un nouvel objet à la variable *TauxTVA*, ajoutant ainsi un taux de TVA intermédiaire de 14% entre 5,5% et 18,6%. A l'évaluation de cette première expression, Smalltalk n'interrogera plus l'utilisateur pour valider la création de la variable globale, puisque cette variable lui est déjà connue. Ensuite, l'évaluation de la deuxième expression modifiera l'objet désigné par *RepertoireCourantDeD* pour désigner le répertoire *\Marie\Musique* sur le disque *D:*.

Comment supprimer une variable globale créée par l'utilisateur ? Smalltalk répertorie toutes les variables globales dans le dictionnaire *Smalltalk*, que nous avons mentionné au début du paragraphe 2.9. Pour supprimer une variable globale, il suffit d'envoyer à ce dictionnaire un message *removeKey:* dont l'argument est un symbole qui représente le nom de la variable[14]. Ainsi, pour supprimer, de l'environnement de Smalltalk, la variable *TauxTVA*, on évaluera :

```
Smalltalk removeKey: #TauxTVA
```

2.9.2 - Les dictionnaires partagés

Un dictionnaire partagé (en Anglais, pool dictionary) est un objet de la classe *Dictionary* qui répertorie des informations que l'on peut partager entre plusieurs classes. Chacune de ces classes doit, dans sa définition, indiquer explicitement

13 Sous réserve, bien entendu, que l'utilisateur confirme son accord pour la création.

14 La gestion des informations d'un dictionnaire sera étudiée au chapitre 3.

une référence à ce dictionnaire partagé. Par exemple, la définition, en Smalltalk V, de la classe *FileStream* :

```
ReadWriteStream subclass: #FileStream
  instanceVariableNames:
    'file pageStart writtenOn lastByte lineDelimiter '
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants '
```

indiquera que cette classe utilise le dictionnaire partagé *CharacterConstants* que nous avons mentionné au début du paragraphe 2.9 et qui répertorie les symboles descriptifs qui peuvent être utilisés pour représenter des caractères spéciaux (*Bell*, *Cr*, *Tab*, *MouseButton*, etc).

2.9.3 - Les variables de classe

Une variable de classe est une variable qui permet le partage d'informations entre toutes les instances d'une même classe. Elle fait partie de la définition de la classe et elle est accessible à la classe ainsi qu'à toutes les instances de la classe.

A titre d'exemple, supposons que, pour une application donnée, nous utilisons une classe *Vehicule*, dont les instances décrivent les véhicules de service d'une entreprise. Pour connaître l'état du parc, on utilisera deux variables de classe. Elles apparaissent dans la définition de la classe qui est donnée, en première partie de la figure 2.13.

Définition de la classe Vehicule

```
Object variableSubclass: #Vehicule
 instanceVariableNames:
   'type annee kilometrage '
 classVariableNames:
   'NombreDeVehicules VehiculesSortis '
 poolDictionaries: ''
```

Deux méthodes de classe de la classe Vehicule

```
ToustAuGarage
  "A la réception de ce message, la classe Vehicule remet à zéro la variable de classe
   VehiculesSortis pour indiquer que tous les véhicules sont au garage"
 VehiculesSortis := 0

VehiculesSortis
  "A la réception de ce message, la classe Vehicule renvoie
   la valeur de la variable de classe VehiculesSortis"
 ^VehiculesSortis
```

Figure 2.13 : deux variables de classe dans la définition d'une classe

La variable de classe *NombreDeVehicules* aura pour rôle d'indiquer le nombre de véhicules du parc de l'entreprise et la variable de classe *VehiculesSortis* indiquera le nombre de véhicules qui sont en train d'effectuer un déplacement. On notera la différence par rapport aux variables d'instance *type*, *année* et *kilometrage* : il n'y a qu'un seul exemplaire de chacune des deux variables *NombreDeVehicules* et *VehiculesSortis* et ces variables sont permanentes. En revanche, chaque instance de la classe *Vehicule* dispose d'un jeu des trois variables *type*, *année* et *kilometrage*, dont la durée de vie est celle de l'instance elle-même. Par exemple, dans l'évaluation de la séquence suivante :

```
|fourgon1|
fourgon1 := Vehicule new.
fourgon1 sort.   "sort est une méthode d'instance de la classe Vehicule"
" ... ici des déplacements avec fourgon1 ... "
fourgon1 rentre   "rentre est une méthode d'instance de la classe Vehicule"
```

trois variables d'instance *type*, *année* et *kilometrage* seront créées au moment de l'évaluation de *fourgon1 := Vehicule new*. Elles disparaîtront de l'environnement, avec la variable temporaire *fourgon1*, à l'issue de l'évaluation de la séquence.

Une variable de classe de la classe C est accessible à la classe C et à ses sous-classes, mais pas aux autres classes. Si V est une variable de classe de la classe C, C peut modifier ou consulter V, par l'intermédiaire de méthodes *de classe*. Par exemple, si nous revenons à la figure 2.13, la deuxième partie de cette figure présente deux méthodes de classe de la classe *Vehicule*. On utilisera par exemple la première, en début d'application, pour remettre à zéro la variable *VehiculesSortis*[15]. La deuxième pourra servir à donner une information sur les véhicules, par exemple en évaluant :

```
Transcript nextPutAll: 'nombre de véhicules à l''extérieur :'.
(Vehicule VehiculesSortis) printOn: Transcript
```

De même, pour permettre à une instance de la classe *Vehicule* de modifier *VehiculesSortis*, il faudra utiliser une méthode *d'instance*. On peut par exemple supposer que le message *sort* envoyé, dans un précédent exemple, à l'instance *fourgon1* correspond à la méthode suivante :

```
sort
  "le véhicule-récepteur sort du garage"
  VehiculesSortis := VehiculesSortis + 1
  . . .  "autres opérations liées à la sortie"
```

Réciproquement, la méthode d'instance *rentre* devra diminuer d'une unité la valeur de *VehiculesSortis*, pour comptabiliser le retour du véhicule.

2.10 - La classe Context et l'évaluation d'un bloc

Nous avons déjà abordé la notion de bloc au paragraphe 2.8.2. Précisons ici ses caractéristiques en les rattachant à la classe *Context*. Les propriétés de cette

---

15 Dans l'environnement de Smalltalk, une variable de classe est une variable permanente qui conserve sa valeur, entre deux exécutions de Smalltalk. Lors de la définition d'une nouvelle classe, toutes les variables d'instance de cette classe sont initialisées à la valeur nil.

classe permettent, en Smalltalk, de définir, dans le protocole d'une méthode, des opérations conditionnelles ou itératives.

2.10.1 - Un bloc est un objet

Distinguons tout d'abord entre séquence d'expressions et bloc. Si nous évaluons, avec *show it*, la séquence :

```
|x y z|
 x:=1. y:=2. z:=3
```

Smalltalk répond en affichant la valeur *3*. Dans l'évaluation d'une séquence, la valeur finale est celle de la dernière expression évaluée (ici, celle de l'entier affecté à la variable temporaire *z*).

Reprenons maintenant la même séquence, mais en encadrant les trois affectations par une paire de crochets [et]. La séquence comprend maintenant une déclaration de trois variables, suivie d'un *bloc* (un bloc est une suite d'expressions, encadrée par des crochets). La séquence ainsi obtenue, que nous appellerons séquence 1, est présentée à la figure 2.14, sur la fenêtre de gauche. Cette fenêtre montre, en inversion vidéo, le résultat de l'évaluation de la séquence 1. Ici, le résultat n'est plus *3* : Smalltalk répond en indiquant que ce résultat est un objet de la classe *Context*. En effet, un bloc est un objet de la classe *Context* : sa valeur est celle de l'objet lui-même, c'est-à-dire la séquence d'expressions entre crochets (et non le résultat de l'évaluation de cette séquence). En fin d'évaluation de la séquence 1, Smalltalk renverra donc la dernière valeur traitée, c'est-à-dire le bloc lui-même.

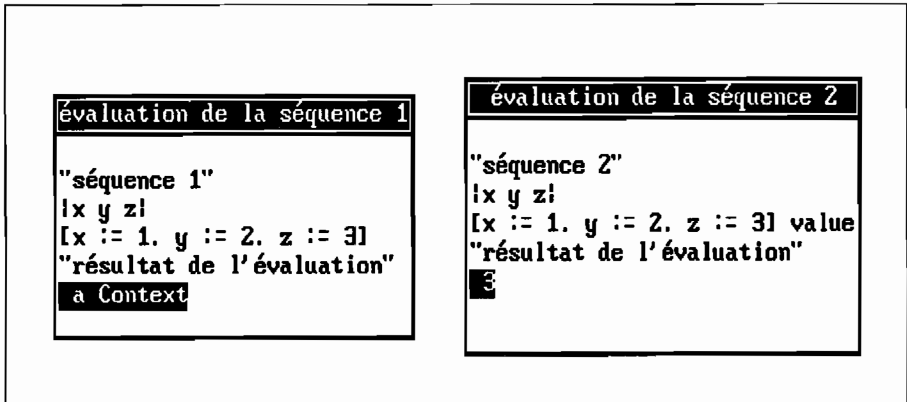


Figure 2.14 : Un bloc s'évalue quand il reçoit le message *value*

Evaluer un bloc, c'est évaluer la séquence d'expressions représentée par ce bloc. Un bloc étant un objet, il est seul habilité à déclencher sa propre évaluation : il le fera quand il recevra le message *value* et la valeur renvoyée sera celle de la dernière expression évaluée. Si nous revenons à la figure 2.14 et si nous examinons la fenêtre de droite, nous constatons que la séquence 2 diffère de la séquence 1 par l'envoi du message *value* au bloc *[x:=1. y:=2. z:=3]*. Le résultat

de l'évaluation de la séquence 2 est donc celui de l'évaluation du bloc : il apparaît en inversion vidéo, au bas de la fenêtre. C'est l'entier *3*, valeur de la dernière expression évaluée dans le bloc.

Notons que, si un bloc fait partie du protocole d'une méthode, il peut contenir une expression de la forme :

^ *expressionFinale*

qui, si elle est évaluée, terminera à la fois l'évaluation du bloc (même si elle n'est pas la dernière expression de ce bloc) et l'exécution de la méthode, en renvoyant la réponse *expressionFinale.*

2.10.2 - Bloc récursif

Un bloc est récursif s'il s'envoie lui-même le message *value*. Cette caractéristique sera utilisée pour la construction de schémas itératifs (cf 2.12). Etudions-la ici, à travers l'exemple suivant.

```
"séquence R"
|entier bloc somme|
entier:=0. somme :=0.
bloc := [entier:=entier+1.
         somme:=somme+entier.
         (entier=4) ifTrue:  [somme]
                    ifFalse: [bloc value]
        ].
bloc value
```

La séquence R utilise trois variables temporaires : *entier*, *somme* et *bloc*. Quand on l'évalue, les variables *entier* et *somme* sont d'abord initialisées avec l'entier *0* puis la variable *bloc* désigne un bloc. Dans ce bloc, les deux premières expressions augmentent *entier* de 1 puis cumulent la valeur ainsi obtenue dans la variable *somme*. La dernière expression du bloc est conditionnelle : si *entier* est différent de 4, elle déclenche une nouvelle évaluation de *bloc* (avec l'évaluation de *bloc value*), sinon elle renvoie la valeur de la variable *somme*. La dernière ligne de la séquence R est celle qui déclenchera la première évaluation de *bloc*. Celle-ci cumulera 1 dans *somme* puis déclenchera une deuxième évaluation de *bloc*, qui, à son tour cumulera 2 dans *somme*, etc jusqu'à ce que la variable *entier* atteigne la valeur 4. L'évaluation de la séquence R sera alors terminée et renverra la valeur 10 de *somme*, qui correspond à la somme des quatre premiers entiers.

Toujours à propos de l'exemple précédent, notons que des blocs peuvent être emboîtés. Le bloc récursif de la séquence R, contient deux autres blocs : *[somme]* et *[bloc value]* dont la présence est imposée par la méthode *ifTrue:ifFalse:* qui exige que ses arguments soient des blocs (cf 2.11).

2.10.3 - Blocs avec arguments

Les blocs que nous venons d'étudier sont des blocs sans arguments. Un bloc peut comporter des arguments qui sont des variables temporaires, déjà présentées en 2.8.2. On les déclare en tête du bloc, entre le crochet gauche [et une barre de

déclaration |, chaque nom d'argument étant immédiatement précédé du caractère : (deux points). Par exemple, un bloc avec un argument aura la structure suivante :

[:*argument* | *sequenceExpressions*]

Un tel bloc s'évaluera quand il recevra le message *value: uneValeur* qui déclenchera l'évaluation de *sequenceExpressions*, dans laquelle la variable *argument* prendra la valeur *uneValeur*. Le lecteur pourra se reporter à l'exemple de la figure 2.10, paragraphe 2.8.2, pour étudier une telle évaluation.

En Smalltalk V, un bloc peut comporter zéro, un ou deux arguments. Un bloc à deux arguments s'évalue quand il reçoit le message :
value: premierArgument value: deuxiemeArgument
qui correspond à la méthode *value:value:* de la classe *Context*.

2.11 - Les classes True et False et la réalisation d'un choix

Dans la hiérarchie des classes, les classes *Boolean*, *True* et *False* rassemblent les objets logiques qui permettent les choix dans l'évaluation d'une séquence d'expressions. La classe *Boolean* est une classe abstraite, classe-mère des deux sous-classes *True* et *False*. La classe *True* est une classe qui possède une seule instance, permanente dans l'environnement de Smalltalk. Cette instance est l'objet *true* qui représente la valeur logique « vrai ». De même, la classe *False* a une seule instance, l'objet *false*. La classe *Boolean* définit des méthodes générales dont héritent *True* et *False*, en particulier la méthode *new*, redéfinie pour éviter l'héritage de la classe *Object*. Cette méthode est, en Smalltalk V, définie de la manière suivante :

```
new
  "Disallow the instantiation of booleans because there is only one true and one false."
  ^self invalidMessage
```

Son exécution provoquera l'apparition d'une fenêtre de diagnostic indiquant que le message *new* n'est pas accepté par la classe : comme on l'imagine, il n'est pas possible de créer de nouveaux objets logiques, distincts de *true* et *false*.

2.11.1 - Schémas conditionnels

Examinons la séquence suivante qui permet de calculer la racine carrée d'un entier positif frappé au clavier par l'utilisateur[16].

```
|entier|
 entier := (Prompter
                prompt: 'je calcule la racine carrée de l''entier :'
                default: '') asInteger.
 entier ≥ 0     ifTrue: [entier sqrt]
```

16 La classe Prompter, utilisée dans cet exemple, offre une interface élémentaire de dialogue avec l'utilisateur. Adressé à cette classe, le message prompt: question default: reponse affiche une question à l'écran et propose une réponse par défaut. Nous reviendrons sur la classe Prompter en 3.5.2. Pour un objet de la classe Prompter, la méthode prompt:default: renvoie toujours le texte frappé par l'utilisateur sous la forme d'un objet de la classe String.

L'évaluation, avec *show it*, de cette séquence fait d'abord apparaître à l'écran une fenêtre de type *Prompter* qui demande à l'utilisateur de frapper un entier et rend cet entier sous la forme d'une chaîne. Cette chaîne reçoit le message *asInteger* et renvoie l'entier correspondant, qui devient la valeur de la variable temporaire *entier*. Ensuite Smalltalk évalue l'expression *entier* ≥ 0 qui, en fonction du signe de l'entier traité, renverra l'objet *true* ou l'objet *false*. Cet objet reçoit alors le message :

```
ifTrue: [entier sqrt]
```

dont le traitement sera différent selon que l'objet récepteur sera *true* ou *false*. Si, par exemple, l'entier choisi était 169, alors l'objet récepteur est *true* et le bloc *[entier sqrt]* est évalué et renvoie[17] la valeur 13.0. Si, au contraire, l'entier choisi était -10, alors l'objet récepteur est *false* et l'évaluation de la méthode *ifTrue:* renvoie *nil*.

On constate que la méthode *ifTrue:*, qui correspond au schéma algorithmique si ... alors ..., provoque un traitement différent selon que l'objet récepteur est *true* ou *false*. Ce polymorphisme de la méthode *ifTrue:* est assuré, pour les sous-classes de *Boolean*, par une définition de la méthode dans chacune des classes *True* et *False*, comme le montre la figure 2.15.

pour la classe True

```
ifTrue: aBlock
    "If the receiver is true, answer the result of evaluating aBlock (with no arguments),
     else answer nil."
    ^ aBlock value
```

pour la classe False

```
ifTrue: aBlock
    "If the receiver is true, answer the result of evaluating aBlock (with no arguments),
     else answer nil."
    ^ nil
```

Figure 2.15 : la méthode *ifTrue:* est définie pour les classes *True* et *False*

On voit donc comment la distinction entre les deux classes *True* et *False* permet de définir simplement un schéma si ... alors .... De la même manière, l'alternative si ... alors ... sinon ... sera représentée par l'envoi, à un objet logique, du message *ifTrue: trueBlock ifFalse: falseBlock* pour lequel la méthode correspondante de la classe *True* renverra *trueBlock value* et celle de la classe *False* renverra *falseBlock value*.

2.11.2 - Opérateurs logiques

Smalltalk définit, pour les classes *True* et *False*, des méthodes correspondant aux opérateurs logiques *non*, *et* et *ou*. Ainsi la méthode unaire *not* renvoie *false* pour la classe *True* et *true* pour la classe *False* : *(3 > 1) not* s'évalue *false*.

17 Quand un entier non négatif reçoit le message sqrt, il renvoie sa racine carrée, sous la forme d'un réel (instance de la classe float).

La conjonction est représentée par deux méthodes qui diffèrent par la nature de leurs arguments :

- la méthode binaire *& aBoolean* prend comme argument un objet de la classe *True* ou *False*. Elle renvoie *aBoolean* pour la classe *True* et *false* pour la classe *False*.
- la méthode à mot-clé *and: aBlock* prend comme argument un bloc, dont le résultat de l'évaluation doit être un objet de la classe *True* ou *False*.

L'exemple suivant montre la différence d'écriture :

```
(3 > 1) & ('abc' reversed = 'cba')      "s'évalue true"
(3 > 1) and: ['abc' reversed = 'cba']   "s'évalue true"
```

Smalltalk V ne fait pas d'autre distinction entre ces deux méthodes que celle qui concerne la nature des arguments[18].

D'une manière analogue, on pourra pour la disjonction non exclusive, utiliser la méthode binaire *| aBoolean* ou la méthode *or: aBlock*. Le ou exclusif correspond à la seule méthode *xor: aBoolean*, dont l'argument n'est pas un bloc mais un objet de la classe *True* ou *False*.

2.12 - La classe Context et les itérations.

Smalltalk dispose d'un mécanisme général d'itération qui est représenté par les méthodes *whileTrue:* et *whileFalse:* de la classe *Context*. La méthode *whileTrue:* est présentée à la figure 2.16.

```
whileTrue: aBlock
  "Repetitively evaluate the receiver block and aBlock, until the result of receiver block
   evaluation is false. Answer nil."
  self value
    ifTrue: [aBlock value.
             self whileTrue: aBlock].
  ^nil
```

Figure 2.16 : la méthode *whileTrue:* de la classe *Context*

Cette méthode correspond au schéma classique *tant que prédicat faire action(s)* et s'utilise dans un énoncé de la forme :

blocCondition `whileTrue:` *blocRépété*

L'objet *blocCondition* reçoit le message *whileTrue*. Cet objet est un bloc, instance de la classe *Context*, dont l'évaluation doit renvoyer l'objet *true* ou l'objet *false*[19].

---

18 En revanche, Smalltalk-80 considère le & comme "évaluatif" (aBoolean est toujours évalué même si le récepteur est l'objet false) et le and: comme non forcément évaluatif.

19 Si l'évaluation de blocCondition ne renvoie pas un objet logique, l'exécution de la méthode whileTrue: affichera une fenêtre de diagnostic avec l'indication d'erreur "receiver is not a boolean" car l'objet renvoyé par l'évaluation de blocCondition reçoit le message ifTrue: (cf figure 2.15).

En effet, le protocole de la méthode *whileTrue:* envoie ensuite à cet objet logique un message *ifTrue:* qui, si l'objet est *true*, déclenche deux opérations successives :

- *blocRépété* est évalué ;
- *whileTrue: blocRépété* est à nouveau envoyé à l'objet *blocCondition* : cette récursivité terminale réalise le mécanisme d'itération.

La méthode *whileFalse:* est conçue selon le même principe, l'itération ne se poursuivant cette fois que tant que l'évaluation de *blocCondition* renvoie *false*.

Avec ces schémas généraux, on peut décrire pratiquement toutes les formes d'itération. En particulier, on peut, dans une itération incluse dans le protocole d'une méthode, provoquer un arrêt de l'itération et une terminaison immédiate de la méthode en évaluant une expression de renvoi de la forme *^valeurRetournée*. A titre d'exemple, définissons, pour la classe *String*, la méthode *estPalindrome* dont le message unaire, adressé à une chaîne, permettra de savoir, si cette chaîne est, ou non, un palindrome, c'est-à-dire si elle se lit de la même manière dans les deux sens (« ici », « elle » sont des palindromes ; le mot « elles » n'en est pas un).

Une manière simple de résoudre le problème revient à observer qu'un palindrome ne change pas si l'on inverse l'ordre de ses caractères. On peut donc utiliser la méthode *reversed* et proposer la définition suivante, pour la méthode cherchée :

```
estPalindrome
  "répond true si la chaîne réceptrice est un palindrome"
  ^ (self = self reversed)
```

Une telle définition est conforme à l'esprit de la programmation orientée objets qui doit être de rechercher systématiquement la réutilisation de code. Ici, on s'appuie sur la méthode *reversed* pour construire la méthode *estPalindrome*. Dans une utilisation pour le prototypage, fréquente avec Smalltalk, on ne peut qu'encourager une telle pratique. Cependant, sur l'exemple précis de la méthode *estPalindrome*, si l'on recherche la rapidité d'exécution, on peut remarquer que pour une chaîne dont le premier et le dernier caractères sont différents, l'utilisation de *reversed* inversera d'abord la chaîne, alors qu'une comparaison des caractères en partant des deux extrémités permettrait de conclure plus rapidement. La deuxième version de *estPalindrome*, présentée à la figure 2.17, est conçue selon ce principe.

```
estPalindrome
  "répond true si le récepteur est un palindrome ou une chaîne vide ;
   version 2 : comparaison caractère par caractère, en partant des deux extrémités"
 |gauche droite|   "désigneront les caractères comparés"
 gauche :=1. droite := self size.
 [gauche  < droite]
    whileTrue: [(self at:gauche) ~= (self at:droite)
                    ifTrue: [^false].
                gauche := gauche + 1.
                droite := droite - 1.].
 ^ true
```

Figure 2.17 : on sort de l'itération dès qu'une différence est constatée

Dans cette version, l'évaluation de ^*false*, dès qu'une différence entre deux caractères est constatée, provoque à la fois la sortie de l'itération et la terminaison de la méthode.

Smalltalk utilise la méthode *whileTrue:* dans de nombreuses méthodes itératives définies pour la classe *Collection* et ses sous-classes, qui permettent d'effectuer un traitement sur l'ensemble des éléments d'une collection ou sur une sélection de ces éléments (cf chapitre 4).

Pour terminer, signalons un dernier schéma itératif, disponible avec la méthode *timesRepeat:* de la classe *Integer*. Cette méthode s'emploie dans une expression de la forme :

```
unEntier timesRepeat: unBloc
```

et déclenchera si *unEntier* est positif, *unEntier* répétitions de l'évaluation de *unBloc*. La figure 2.18 donne le protocole de cette méthode ainsi qu'un exemple d'utilisation.

La méthode timesRepeat:

```
timesRepeat: aBlock
    "Evaluate the block aBlock n number of times, where n is the receiver."
    | anInteger |
    anInteger := self.
    [anInteger > 0]
       whileTrue: [anInteger := anInteger - 1.
                   aBlock value]
```

Exemple d'utilisation

l'évaluation de la séquence suivante

```
"calcul, dans p, de 2 puissance 32"
|p| p:=1. 32 timesRepeat: [p := p*2].
p
```

donne, avec *show it*, le résultat *4294967296*

Figure 2.18 : la méthode *timesRepeat:*

3 - Aller plus loin avec Smalltalk

3.1 - Faire des calculs

Comme tout environnement de programmation, Smalltalk fournit des primitives pour manipuler des nombres. Tous ces nombres sont des instances de sous-classes de la classe *Number*, dont la descendance est représentée à la figure 3.1. La classe *Number* elle-même est une sous-classe de la classe ***Magnitude***, qui rassemble des méthodes permettant de comparer des grandeurs mesurables et que nous avons abordée en 1.6 (Cf figure 1.8).

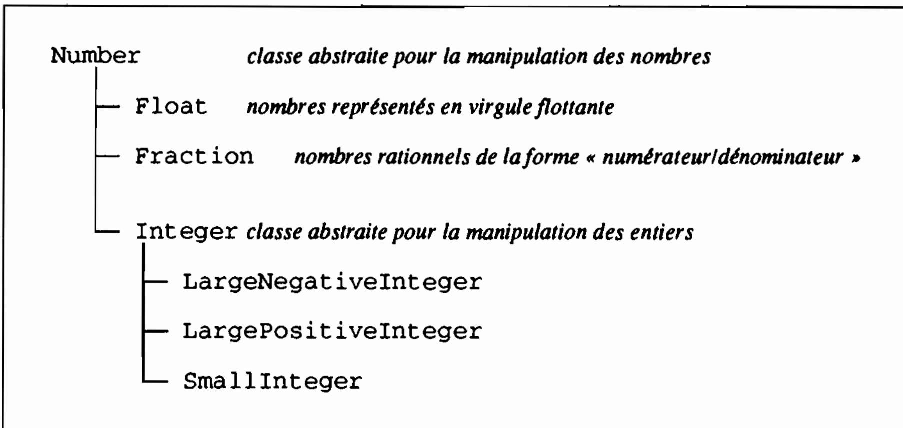


Figure 3.1 : la classe *Number* et sa descendance

Un nombre en Smalltalk est toujours une instance de l'une des classes *Float*, *Fraction*, *LargeNegativeInteger*, *LargePositiveInteger* ou *SmallInteger*. Toutes ces classes héritent de la classe abstraite *Number*, qui rassemble des méthodes générales qui s'appliqueront à toutes les sous-classes comme, par exemple, la méthode *squared* qui renvoie le carré du récepteur :

```
squared
    "Answer the receiver multiplied by the receiver."
    ^ self * self
```

Ces méthodes générales font souvent appel aux opérations arithmétiques de base (addition, soustraction, multiplication et division) qui, elles, sont toutes définies dans la classe *Number* par le protocole *^implementedBySubclass* car elles doivent impérativement être redéfinies pour chaque sous-classe de *Number.* C'est cette redéfinition qui assurera le polymorphisme de méthodes telles que *squared.*

3.1.1 - La représentation littérale des nombres

La manière la plus courante de représenter un nombre en Smalltalk est d'écrire une suite de chiffres décimaux, précédés éventuellement du signe - (moins), et qui peuvent être suivis d'un point et d'une autre suite de chiffres décimaux :

```
5    512    -53    345.12    -5.0
```

On notera qu'un nombre positif s'écrira toujours sans signe car Smalltalk considère systématiquement le signe + comme le sélecteur de la méthode binaire d'addition. Ainsi, Smalltalk répondra par l'indication d'erreur *invalid receiver* à l'évaluation de *+50.* En revanche, Smalltalk distingue entre la méthode binaire de soustraction - et le signe -, en imposant au signe d'être immédiatement suivi d'un chiffre : l'évaluation de *-50* renvoie *-50* mais l'évaluation de l'expression - *50* renvoie *invalid receiver.*

On peut aussi représenter un nombre en utilisant la notation scientifique <m>e<p> où <m> est un nombre tel que ceux définis précédemment et <p> un exposant entier, positif ou négatif. Le nombre représenté est celui obtenu en multipliant la valeur de <m> par la <p>-ième puissance de la base (en général 10) : l'écriture *3.45e-2* représente la valeur 0,0345 ; *12e193* représente l'entier 12×10^{193}. On notera que l'exposant est obligatoirement introduit par la lettre minuscule e car la majuscule E représente le « chiffre » 14 dans les systèmes de numération à base supérieure à 14.

En effet, on peut aussi représenter des nombres en les écrivant dans une base différente de 10. Dans ce cas, la représentation du nombre est précédée du préfixe <b>r (r pour « radix », base en Anglais), l'entier <b> indiquant la base. A l'évaluation, Smalltalk renverra toujours une écriture en base 10. Pour représenter les chiffres supérieurs à 9 (dans une base plus grande que 10), on utilise les majuscules A (pour 10), B (pour 11), etc. Par exemple :

- l'évaluation de *2r111* renverra *7,*
- l'évaluation de *16rA0* renverra *160,*
- l'évaluation de *17r1G* renverra *33,*
- l'évaluation de *16rA0.8* renverra *160.5.*

3.1.2 - Entiers, rationnels et réels approchés

Smalltalk classe les nombres en trois catégories avec, pour chacune, une représentation particulière : les entiers (classe *Integer* et ses sous-classes), les valeurs exactes de rationnels non entiers (classe *Fraction*) et les approximations de réels représentés en virgule flottante (classe *Float*).

Les méthodes applicables aux entiers sont rassemblées dans la classe *Integer*, qui comprend trois sous-classes (Cf figure 3.1). La classe *SmallInteger* est celle des entiers pour lesquels l'ordinateur utilisé dispose de circuits de calcul standard[1]. Les autres entiers sont répartis, selon leurs signes, en deux classes : *LargePositiveInteger* et *LargeNegativeInteger* (qui, en Smalltalk V, permettent de traiter des entiers dont la représentation peut occuper jusqu'à 64Ko). En Smalltalk V, pour tous ces entiers, les méthodes de calcul correspondant aux opérations élémentaires d'addition, de soustraction, de multiplication et de division sont définies dans la classe *Integer* sous la forme de primitives écrites en langage d'assemblage, qui font la distinction entre les classes des entiers manipulés et font appel soit aux instructions-machine de calcul (classe *SmallInteger*) soit à des sous-programmes de calcul (autres sous-classes de *Integer*).

La classe *Fraction* est la classe des rationnels non entiers, qui sont représentés de manière exacte par le quotient *n/d*, *n* et *d* étant deux entiers relatifs premiers entre eux. Ainsi *2/3* est une instance de la classe *Fraction* dont l'approximation *6.666667e-1* est elle-même une instance de la classe *Float* (Cf ci-après, paragraphe 3.1.4, pour cette dernière classe).

Chaque objet de la classe *Fraction* a deux variables d'instance, *numerator* et *denominator*, qui déterminent le rationnel représenté. A la différence de la classe *Integer*, qui redéfinit les méthodes de classes héritées pour interdire la création d'instances nouvelles, la classe *Fraction* dispose de plusieurs méthodes de classes qui permettent de créer des objets de cette classe. En effet si, pour les entiers, on peut considérer que toutes les valeurs sont prédéfinies dans l'environnement et peuvent être représentées par des constantes sous l'une des formes décrites en 3.1.1, chaque objet de la classe *Fraction* doit être créé, afin que ses variables d'instances soient représentées. On peut créer un objet de la classe *Fraction* en utilisant les méthodes de classe *new* ou *numerator:denominator:* ou encore, en envoyant à un entier e_1 le message $/e_2$, dans lequel e_2 n'est pas un diviseur de e_1. La figure 3.2 donne des exemples de création d'instances de la classe *Fraction*.

```
|f1 f2 f3 f4 e r|
 f1 := Fraction new.      "f1 désigne l'instance nil/nil de la classe Fraction"
 f2 := Fraction numerator: 2 denominator: 3.
                          "f2 désigne l'instance 2/3 de la classe Fraction"
 f3 := 2/3.               "f3 désigne l'instance 2/3 de la classe Fraction"
 f4 := 8/12.              "f4 désigne l'instance 2/3 de la classe Fraction"
 e := 8/2.                "e désigne l'entier 4 de la classe SmallInteger"
 r := 8/12.0              "r désigne la valeur 6.6666667e-1  de la classe Float"
```

Figure 3.2 : quatre instances de *Fraction*, un entier, un réel approché

On remarquera, sur cette figure, que la méthode / de la classe *Integer* utilisée avec un argument entier, effectue une simplification éventuelle comme le montre l'affectation de *8/12* à *f4*. La méthode de classe *numerator:denominator:* ne réalise, en revanche, qu'une simple initialisation des variables d'instances.

1 de -32767 à +32768, sur la plupart des postes de travail en 1990.

Constatons-le, en utilisant, dans le paragraphe suivant, une fenêtre d'inspection sur une instance de la classe *Fraction*.

3.1.3 - Inspection d'un objet

Smalltalk permet, chaque fois qu'on le désire, d'examiner un objet pour mieux en connaître la structure ou pour vérifier que l'état de cet objet est bien celui que l'on attendait. Pour cela, il suffit d'envoyer à l'objet que l'on veut examiner le message *inspect*. La méthode correspondante est définie dans la classe *Object* et elle est donc héritée par toutes les classes[2]. L'effet de cette méthode est d'ouvrir, à l'écran, une fenêtre d'inspection sur l'objet-récepteur. Pour nous familiariser avec une telle fenêtre, examinons la figure 3.3 qui présente deux fenêtres d'un écran Smalltalk.

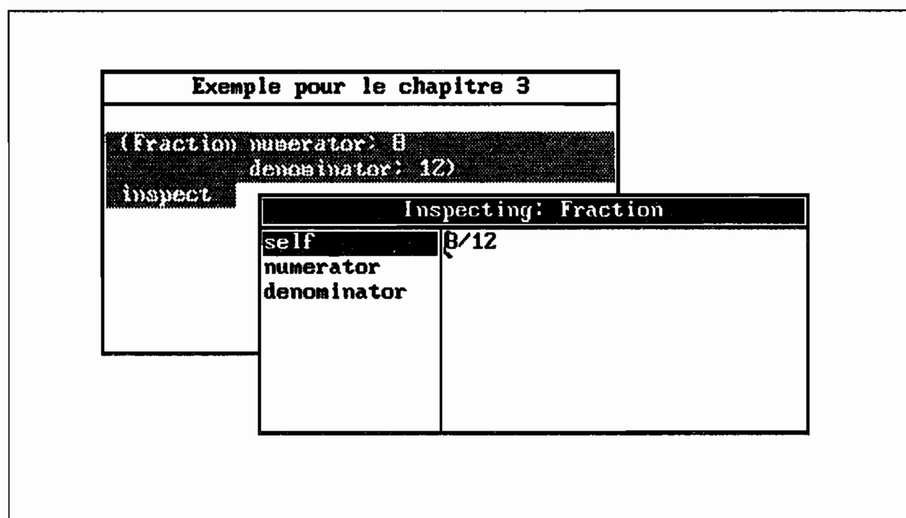


Figure 3.3 : inspection d'une instance de la classe *Fraction*

Pour comprendre les informations qui sont présentées sur cette figure, examinons d'abord la fenêtre en arrière-plan, dont la barre de titre indique « Exemple pour le chapitre 3 ». On a sélectionné, dans cette fenêtre l'expression :

```
(Fraction numerator: 8 denominator: 12) inspect
```

puis fait apparaître le menu de cette fenêtre et déclenché, avec *do it*, l'évaluation de la séquence. Cette évaluation s'est effectuée en deux temps :

1) La classe *Fraction* reçoit le message *numerator:8 denominator:12* et renvoie une nouvelle instance.

2 Sauf pour la classe Dictionary, qui redéfinit cette méthode pour adapter la présentation de la fenêtre d'inspection à la structure d'un dictionnaire (Cf 3.5).

2) Cette nouvelle instance de la classe *Fraction* reçoit le message *inspect* et ouvre une fenêtre d'inspection sur elle-même. C'est cette fenêtre, dont la barre de titre indique « inspecting Fraction », qui est la fenêtre active.

Une fenêtre d'inspection comporte deux panneaux. Le panneau de gauche présente une liste, dont les éléments désignent dans l'ordre, l'objet inspecté (*self*) et ses variables d'instances (désignées par leurs noms ou par leurs numéros s'il s'agit de variables indexées). On peut sélectionner un des éléments de cette liste, en cliquant sur le bouton gauche de la souris. On fait alors apparaître la description complète de l'élément dans le panneau de droite. Sur l'exemple de la figure 3.3, l'objet lui-même est sélectionné (*self*) et l'on voit bien, dans le panneau droit, que l'instance de la classe *Fraction* créée par la méthode *numerator:denominator:* n'a pas été simplifiée, puisque l'instance *8/12* est affichée (au lieu de *2/3*).

3.1.4 - La classe Float

La classe *Float* est celle des nombres dont la représentation interne est en virgule flottante (avec mantisse et exposant) et pour lesquels les calculs nécessitent, soit un coprocesseur de calcul en virgule flottante, soit des sous-programmes d'émulation. Les constantes numériques dont les règles d'écriture ont été données en 3.1.1, sont de la classe *Float* si leur représentation contient un point décimal, sinon elles sont traitées comme des instances de la classe *Fraction* ou *Integer*. Ainsi l'évaluation de

- *1.35 class* renvoie *Float*,
- *15e-130 class* renvoie *Fraction*,
- *15e130 class* renvoie *LargePositiveInteger*.

Pour les opérations arithmétiques élémentaires (Cf 3.1.5), les règles qui déterminent le mode de calcul à appliquer quand une expression contient des valeurs de classes différentes, sont appliquées, en Smalltalk V, directement dans les primitives de calcul définies en langage d'assemblage pour une plus grande efficacité[3].

La classe *Float* comporte deux méthodes de classe qui permettent de créer un nombre représenté en virgule flottante. Ce sont les méthodes :

- *fromInteger: anInteger* qui renvoie la représentation en virgule flottante de l'argument entier *anInteger*,
- *pi* qui renvoie la représentation approchée de π, en virgule flottante, avec la meilleure approximation permise par la machine utilisée.

Comme nous l'avons indiqué au début de 3.1, la classe *Number* rassemble des méthodes générales dont héritent toutes les sous-classes correspondant à des valeurs numériques. Pour la plupart, ces méthodes correspondent à des fonctions numériques comme, par exemple, la méthode *cos*, qui calcule le cosinus du nombre-récepteur et qui est représentée en première partie de la figure 3.4. On constate que cette méthode n'effectue pas elle-même le calcul mais envoie, à la représentation en virgule flottante du récepteur (obtenue par le message *asFloat*),

---

3 Smalltalk-80 implémente ces règles directement en Smalltalk et utilise, pour cela, des méthodes de conversion (« coercing messages »).

le message *cos*. Il ne s'agit nullement d'un appel récursif, car ce dernier message *cos* est envoyé à une instance de la classe *Float* et fait donc appel à la méthode *cos* redéfinie pour cette classe et présentée en deuxième partie de la figure.

méthode cos de la classe Number

```
cos
  "Answer a Float which is the cosine of the receiver. The receiver is an angle measured in
   radians."
  ^self asFloat cos
```

méthode cos de la classe Float

```
cos
  "Answer the cosine of the receiver. The receiver is an angle measured in radians"
   <primitive: 30>
  ^self class floatError
```

Figure 3.4 : les deux méthodes pour le calcul du cosinus

Examinons de plus près cette seconde méthode. L'expression *<primitive: 30>* indique qu'elle s'exécute en faisant appel à une primitive écrite en langage d'assemblage. En effet, si la majeure partie du code de Smalltalk est écrite en Smalltalk, une très faible partie de ce code est implantée sous la forme de primitives réalisées en langage-machine[4].

Dans une méthode, l'appel d'une primitive se note entre les signes < et >. La primitive appelée est conçue pour gérer complètement la plupart des cas du traitement auquel elle correspond. Elle ne retourne à la méthode qu'en cas d'échec. Dans ce dernier cas, le code qui suit l'appel de la méthode est évalué. Pour l'exemple de la figure 3.4, en cas d'échec de la primitive 30, la méthode *cos* renverra la valeur *self class floatError* qui appelle la méthode de classe *floatError* de la classe *Float*. Cette méthode analyse la cause de l'erreur (dépassement de capacité, division par zéro, etc) et affiche une fenêtre avec le message d'erreur correspondant.

Pour en terminer avec l'exemple de la figure 3.4, remarquons que, comme pour toutes les fonctions trigonométriques de base de Smalltalk, les angles sont évalués en radians. Pour les conversions, les méthodes *degreesToRadians* et *radiansToDegrees* sont disponibles dans la classe *Number*. Comme *cos* et la plupart des fonctions numériques, elles s'appuient sur un appel de primitive ou un calcul direct dans la classe *Float*. Nous donnons par exemple la réalisation, en Smalltalk V, des deux méthodes *degreesToRadians*.

```
degreesToRadians        "méthode d'instance de Number"
  "Answer the receiver converted from degrees to radians."
  ^ self asFloat degreesToRadians
degreesToRadians        "méthode d'instance de Float"
  "Answer the number of radians the receiver represents in degrees."
  ^ self * Float pi / 180
```

4 Une centaine de primitives pour Smalltalk V.

A propos des conversions, examinons la manière d'obtenir, par la méthode *asFloat*, la représentation en virgule flottante d'un entier ou d'un rationnel de la classe *Fraction*. La figure 3.5 présente les méthodes utilisées. On peut être, au premier abord, surpris que la classe *Float* dispose également de cette méthode qui ne fait rien d'autre que renvoyer le récepteur. En fait cette méthode est indispensable pour assurer le bon fonctionnement des méthodes numériques héritées de *Number* qui, pour le calcul, effectuent d'abord une conversion en virgule flottante, en envoyant à l'objet récepteur le message *asFloat*, quelle que soit la classe de cet objet.

méthode asFloat de la classe Fraction

```
asFloat
  "Answer the receiver as a floating point number."
  ^numerator asFloat / denominator asFloat
```

méthode asFloat de la classe Integer

```
asFloat
  "Answer the floating point representation of the receiver."
  ^Float fromInteger: self
```

méthode asFloat de la classe Float

```
asFloat
  "Answer the receiver as a floating point number."
  ^self
```

Figure 3.5 : méthodes *asFloat* pour les classes *Fraction*, *Integer* et *Float*

3.1.5 - Opérations arithmétiques élémentaires

Les opérations arithmétiques usuelles (addition, soustraction, multiplication, division) sont, en Smalltalk, représentées par des messages binaires pour lesquels, rappelons-le, l'ordre d'évaluation, dans une expression sans parenthèses, est de gauche à droite. Ainsi l'expression *3 +5 * 5* s'évaluera *40* (et non pas *28*, comme avec la plupart des autres langages où la multiplication a priorité sur l'addition).

Lorsque, dans une expression, les valeurs sont de classes différentes, Smalltalk effectue les conversions nécessaires, comme le montrent les exemples de la figure 3.6. On notera, à propos du deuxième exemple de cette figure, le rôle des parenthèses qui amènent Smalltalk à évaluer d'abord 1/2 dont le résultat est une fraction qui devient l'argument du message + envoyé à 1. Sans les parenthèses l'évaluation de *1 + 1/2* renverrait l'entier *1*. On remarquera également que les exemples de la deuxième et de la troisième ligne de la figure représentent la

même expression : sauf pour le sélecteur[5] -, la présence d'espaces autour d'un opérateur est facultative.

```
1 + 1.5         "renvoie 2.5 de la classe Float"
1 + (1/2)       "renvoie 3/2 de la classe Fraction"
1 + (1 / 2)     "renvoie 3/2 de la classe Fraction"
3/2 + 1         "renvoie 5/2 de la classe Fraction"
3/2 + 1.0       "renvoie 2.5 de la classe Float"
```

Figure 3.6 : quelques expressions arithmétiques avec des valeurs de classes différentes

Les méthodes correspondant à toutes ces opérations arithmétiques élémentaires sont, en Smalltalk V, en général réalisées par des appels de primitives en langage-machine. Lorsque ce n'est pas le cas, la méthode utilise indirectement de telles primitives. Examinons, par exemple, la méthode * de la classe Fraction :

```
* aNumber
  "Answer the result of multiplying the receiver by aNumber."
  ^ (numerator * aNumber numerator)
      /
    (denominator * aNumber denominator)
```

Cette méthode envoie elle-même deux messages correspondant aussi au sélecteur * respectivement aux variables d'instance *numerator* et *denominator* de la fraction. Comme ces variables d'instance sont des entiers, ces deux messages déclencheront l'exécution de la méthode * de la classe *Integer* qui, elle, fera appel à une primitive en langage-machine.

Toujours à propos de la méthode * de la classe *Fraction*, on remarquera que la méthode * effectue apparemment le même traitement quelle que soit la classe de l'argument aNumber. Si cet argument est une autre fraction, on comprend bien le fonctionnement de la méthode, qui correspond à la multiplication de deux fractions. Mais, si aNumber est un entier ou une instance de la classe *Float*, on peut se demander comment il réagira aux messages *numerator* et *denominator* qui lui sont adressés, d'autant que ces deux méthodes sont apparemment absentes des deux classes *Integer* et *Float*. Elles sont, en fait, héritées de Number, où elles sont définies de la manière suivante :

```
numerator         "méthode de la classe Number"
  "Answer the numerator of the receiver.
   Default is the receiver which can be overridden by the subclasses."
  ^ self
denominator       "méthode de la classe Number"
  "Answer the denominator of the receiver.
   Default is one which can be overridden by the subclasses."
  ^ 1
```

Sachant maintenant que pour une instance de la classe *Float* ou *Integer*, les méthodes *numerator* et *denominator* renvoient respectivement le nombre lui-

5 Rappelons en effet (Cf 3.1.1) que seul le sélecteur - ne peut être suivi immédiatement par un chiffre car, sinon, il est interprété comme le signe d'un nombre négatif.

même et l'entier *1*, on comprend comment est assuré le polymorphisme pour la méthode * de la classe *Fraction.*

Pour en terminer avec les opérateurs arithmétiques, décrivons brièvement les opérateurs associés à la division. La méthode / réalise, pour toutes les classes, la division et rend le quotient avec la meilleure approximation possible, compte tenu des classes des nombres participant à la division :

```
12 / 4      "renvoie l'entier 3"
17 / 3      "renvoie la fraction 17/3"
17 / 3.0    "renvoie 5.6666667 de la classe Float"
```

La méthode // permet d'obtenir l'entier immédiatement inférieur au quotient exact. Si *a* et *b* sont deux entiers positifs, *a // b* renvoie donc le quotient euclidien de *a* par *b*. Le reste d'une division peut être obtenu par la méthode \\ dont le résultat, si *a* et *b* sont des entiers relatifs, est obtenu conformément à la règle suivante :

*a \\ b = a - (a // b) *b.*

La figure 3.7 donne quelques exemples d'utilisation de // et \\.

```
27 // 4     "renvoie 6 car 27=4*6.75 et 6 est le plus grand entier inférieur à 6.75"
27 // -4    "-7 car 27=(-4)*(-6.75) et -7 est le plus grand entier  inférieur à 6.75"
```

Figure 3.7 : exemples d'utilisation des méthodes // et \\

3.1.6 - Les intervalles

Bien qu'elle ne soit pas une descendante de la classe *Number*, la classe *Interval* est liée à cette première classe, dans la mesure où ses instances sont des progressions arithmétiques. Une instance de cette classe, est une suite croissante de nombres définie par ses trois variables d'instances :

beginning le premier nombre de la suite,

end la borne supérieure de la suite,

increment l'intervalle entre deux nombres consécutifs (raison de la progression).

Une instance de *Interval* peut être créée avec la méthode de classe *from:to:* ou *from:to:by:* comme le montrent les exemples de la figure 3.8. Sur cette figure, chaque nouvelle instance créée reçoit le message *asArray* et renvoie en réponse un tableau contenant tous les nombres de la progression. Le premier exemple montre que, en l'absence d'indication explicite de la raison de la progression, l'intervalle entre deux nombres consécutifs est de 1. On notera avec le troisième exemple que le dernier nombre d'une progression peut être inférieur à l'argument du mot-clé *to:*. En effet, dans ce troisième exemple, la différence entre la borne supérieure et le premier nombre de la progression n'est pas un multiple de la raison de la progression. Constatons également, avec le dernier exemple de la figure, que toute suite qui ne peut être croissante, est une instance vide de la classe *Interval.*

```
(Interval from: 2 to: 5) asArray.    "renvoie le tableau (2 3 4 5)"
(Interval from: 1 to: 2.5 by: 0.5) asArray.
        "renvoie le tableau (1.0 1.5 2.0 2.5)"
(Interval from: 1 to: 2.2 by: 0.5) asArray.
        "renvoie le tableau (1.0 1.5 2.0)"
(Interval from: 5 to: 2) asArray.    "renvoie le tableau vide ()"
```

Figure 3.8 : quelques instances de la classe *Interval*

Une progression arithmétique, instance de *Interval*, peut aussi être créée avec les méthodes d'instances *to:* et *to:by:* de la classe *Number*, qui sont héritées par toutes les sous-classes de celle-ci. Par exemple, la seconde progression de la figure 3.8 aurait pu être obtenue par l'évaluation de *1 to:2.5 by:0.5*, qui représente l'envoi d'un message à l'entier *1* qui répond en renvoyant la progression (*1.0 1.5 2.0 2.5*).

Indiquons enfin pour terminer que la classe *Interval* est une des classes descendantes de la classe *Collection* (Cf chapitre 4) et qu'à ce titre elle dispose de la méthode *do:*, déjà abordée en 2.8.3. Quand une progression arithmétique non vide, instance de la classe *Interval*, reçoit le message *do: unBloc*, elle déclenche l'évaluation de *unBloc* pour chacun des nombres qui la composent, l'argument de *unBloc* prenant successivement la valeur de chaque nombre de la progression. L'exemple suivant propose une utilisation de cette méthode pour l'affichage d'une table des carrés.

```
"affichage sur System Transcript d'une table des carrés de 1 à 20"
(1 to: 20)
  do: [:entier|
        Transcript nextPutAll: 'le carré de '.
        entier printOn: Transcript.
        Transcript nextPutAll: ' est '.
        entier * entier printOn: Transcript.
        Transcript cr]
```

3.2 - Les tableaux

Nous avons vu dans les précédents chapitres comment créer une constante tableau (instance de la classe *Array*) . Nous montrerons dans celui-ci comment manipuler les instances de cette classe en construisant un environnement facilitant le calcul matriciel. Auparavant, nous allons étudier comment créer un tableau dont les éléments ne sont plus des constantes mais les valeurs de variables.

3.2.1 - Variables et tableaux

Nous pouvons créer un tableau de trois manières différentes :

```
|tab1 tab2 tab3|
tab1 := Array new:3.
tab1 at:1 put:'un';
     at:2 put:'deux';
     at:3 put:'trois'.

tab2 := #('un' 'deux' 'trois').

tab3 := Array with: 'un' with: 'deux' with: 'trois'.
```

Les trois variables *tab1*, *tab2* et *tab3* représentent chacune un tableau contenant les objets *'un'* *'deux'* et *'trois'*. Nous avons vu dans les chapitres précédents l'instanciation par le message *new:* (*tab1*) et l'affectation (*tab2*). La dernière méthode (*with:with:with:*) est héritée de la classe *Collection* dont la descendance est présentée jusqu'à la classe *Array* dans la figure figure 3.9.

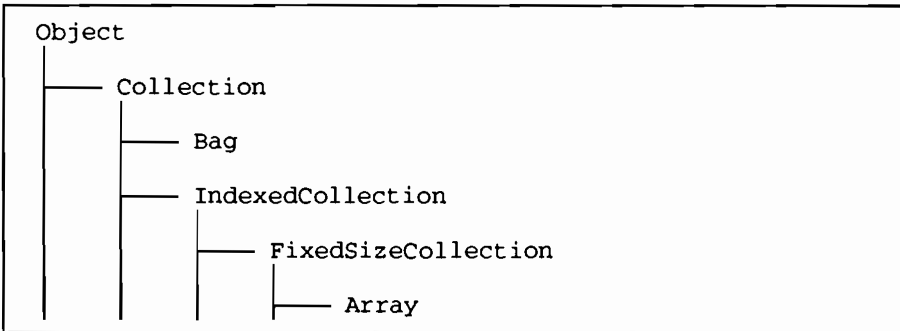


Figure 3.9 : position de la classe *Array* dans la hiérarchie des classes

Ces trois façons de créer un tableau ne sont pourtant pas totalement identiques. L'évaluation par *show it* des séquences du tableau 3.10 illustre cette différence.

La première séquence évaluée renvoie un tableau contenant la lettre *x* et non la valeur *10*. La deuxième séquence renvoie la classe à laquelle appartient cette lettre *x* : il s'agit d'un symbole. La syntaxe de Smalltalk pour la classe *Array* (Cf 2.6) interdit toute utilisation d'une variable à l'intérieur des parenthèses délimitant le tableau. Dans les deux premières séquences, *x* est donc interprété par Smalltalk comme un symbole (un caractère serait précédé du signe $, une chaîne de un caractère serait encadrée par des apostrophes).

Cette syntaxe nous interdit d'intégrer ainsi la valeur d'une variable parmi les éléments d'une constante tableau : il faudra utiliser les méthodes *at:put:* ou *with:*.

La troisième séquence renvoie le nombre *20*, que nous avons chargé en dernier dans le tableau *y*. La méthode *at:put:* ne renvoie pas le tableau complété mais l'objet que l'on ajoute. Par contre, la méthode *with:with:* renvoie le tableau complet (quatrième séquence).

Séquence à évaluer	Résultat
|x| x := 10. #(x 20)	(x 20)
|x y| x := 10. y := #(x 20). (y at:1) class	Symbol
|x y| x := 10. y := Array new:20. y at:10 put:x ; at:20 put:20.	20
|x| x := 10. Array with:x with:20	(10 20)
|x y| x := 10. y := Array with:x with:x. x := 20. y	(10 10)

Tableau 3.10 : Différentes méthodes pour créer une instance de la classe *Array*

La méthode *with:with:* est une méthode de classe utilisée pour créer une instance : par définition, elle doit évidemment renvoyer l'instance créée. La méthode *at:put:* est une méthode d'instance, utilisée pour modifier l'objet auquel on l'applique. Dans le cas de tableaux emboîtés les uns dans les autres, les accès se feront par des messages *at:* ou *at:put:* en cascade : il ne faudra pas perdre de vue que l'objet renvoyé par le message *at:put:* n'est pas le tableau lui-même.

Dans la cinquième séquence, nous voyons que la modification de la variable *x* n'a aucune incidence sur le contenu du tableau *y*, qui a pourtant été défini avec cette variable *x*. Une variable n'est pas un objet : elle n'est qu'un pointeur sur un objet (Cf 2.6). En évaluant la ligne `y := Array with:x with:x.`, Smalltalk n'associe pas les éléments du tableau à la variable *x* mais recopie le pointeur sur l'objet *10* (contenu actuel de la variable *x*) dans chacune des deux cases du tableau. Modifier la variable *x* (donc changer la valeur du pointeur) ne peut avoir d'incidence sur le contenu de *y*, puisque les cases du tableau pointent toujours sur l'objet *10* qui lui, reste intact. Nous voyons là une nouvelle illustration de la philosophie « orientée objets » : l'objet *y* est seul responsable de son intégrité.

Ces différents points peuvent surprendre un programmeur habitué aux langages structurés. Ils imposent en tout cas de porter une attention particulière à la

définition des méthodes construisant des tableaux emboîtés les uns dans les autres, comme nous allons le voir dans le paragraphe suivant.

3.2.2 - Un exemple d'utilisation des tableaux : le calcul matriciel

Le calcul matriciel s'effectue sur des matrices. Cette remarque de bon sens nous conduit à une première interrogation : existe-t-il une classe dont les instances pourraient être des matrices ?

La réponse est oui : la classe *Array* permet de créer un tableau dont les éléments peuvent être à leur tour d'autres tableaux[6]. Nous sentons pourtant que la manipulation de telles instances risque d'être lourde et relativement complexe.

Il est préférable de créer une classe héritant de *Array*. Cette classe utilisera les méthodes de ses classes mères, mais nous pourrons lui ajouter quelques méthodes spécifiques, comme *discriminant* ou *.

Une nouvelle question nous arrête : faut-il créer une classe par type de matrice (2x2, 2x3 etc...) ou regrouper les matrices de toutes dimensions au sein de la même classe ?

Avoir une seule classe offre évidemment plus de souplesse. Mais les méthodes seront plus complexes à établir. D'autre part, il sera nécessaire de connaître les dimensions de chaque instance, pour pouvoir appliquer les calculs à bon escient. Nous définirons donc deux variables d'instance (*ligne* et *colonne*) donnant les grandeurs horizontales et verticales de la matrice.

Créer une nouvelle classe

Pour créer la nouvelle classe *Matrice*, nous allons travailler sur la fenêtre du *Class Hierarchy Browser* (Cf 1.7). Il faut d'abord cliquer sur le nom de la classe mère, *Array*, dans le panneau de gauche (panneau des classes) en faisant défiler si nécessaire la liste des classes avec la souris. Une fois la classe *Array* sélectionnée, une courte pression du bouton droit de la souris provoque l'affichage d'un menu. Cliquer sur l'option *add subclass* permet d'ajouter une nouvelle classe. Après avoir entré son nom, il faut préciser son type (dans notre cas, nous sélectionnerons le type *variableSubclass*[7]). Le panneau inférieur affiche alors la définition provisoire de notre classe (figure 3.11).

```
Array variableSubclass: #Matrice
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

Figure 3.11 : définition provisoire de la classe *Matrice*

6 Nous pouvons assimiler une matrice à un tableau de colonnes (ou de lignes).

7 Nous verrons au paragraphe 3.3 la signification des différents types de sous-classes.

Ce panneau fonctionne comme une fenêtre d'édition de texte. Nous allons modifier la ligne

```
instanceVariableNames: ''
```

en ajoutant deux variables d'instance entre les parenthèses :

```
instanceVariableNames: 'ligne colonne'.
```

Pour enregistrer les modifications, il faut encore appeler le menu de ce panneau (en cliquant rapidement avec le bouton droit de la souris) et choisir l'option *save*.

Accéder aux variables d'instance

La plupart des méthodes que nous allons construire pour faire du calcul matriciel devrait connaître les dimensions des matrices. Nous définirons donc quatre méthodes d'instance permettant de définir et de consulter les variables *ligne* et *colonne* (figure 3.12).

Les variables d'instance *ligne* et *colonne* prendront définitivement leurs valeurs à la création de la matrice. Les méthodes *ligne:* et *colonne:* seront donc appelées par la méthode de classe chargée de créer de nouvelles instances. Les méthodes *ligne* et *colonne* seront utilisées par toutes les méthodes d'instance nécessitant les dimensions de la matrice pour fonctionner.

```
ligne
  "Répond la valeur de ligne."
  ^ligne

ligne: uneValeur
  "Fixe la valeur de ligne."
  ligne := uneValeur

colonne
  "Répond la valeur de colonne."
  ^colonne

colonne: uneValeur
  "Fixe la valeur de colonne."
  colonne := uneValeur
```

Figure 3.12 : méthodes d'accès aux variables d'instance

Créer une nouvelle matrice

Une matrice étant une succession de lignes, nous pouvons l'assimiler à un tableau de tableaux. Créer une nouvelle instance revient alors à créer un tableau contenant autant de positions que de lignes dans la matrice (méthode de classe *deDimensions:*), puis à mettre à chaque position un tableau dont le nombre

d'éléments sera le nombre de colonnes de la matrice (méthode d'instance *ligne:colonne:*). La méthode *deDimensions:* (figure 3.13) crée cet objet.

```
deDimensions : unTableau
  "crée une nouvelle instance de Matrice dont les dimensions sont précisées
   dans unTableau"
  ^(self new:(unTableau at:1)) ligne:(unTableau at:1)
                               colonne:(unTableau at:2)
```

Figure 3.13 : méthode de classe *deDimensions:*

La méthode *deDimensions:* telle que nous la décrivons figure 3.13 utilise un argument : celui-ci est un tableau contenant les deux dimensions de la matrice. Une matrice 2x3 sera créée en évaluant *Matrice deDimensions:#(2 3)*.

La première évaluation de la méthode *deDimensions:* consiste à créer un tableau de la classe *Matrice* qui contiendra autant d'éléments que de lignes dans la matrice (*new:(unTableau at:1)*). Le nombre de lignes de la matrice est extrait du tableau donné en argument par l'expression (*unTableau at:1*). Cette nouvelle instance reçoit ensuite le message *ligne:colonne:*.

```
ligne:i colonne:j
  "structure la matrice en attribuant leurs valeurs aux variables ligne et  colonne"
  self ligne:i;
       colonne:j.
  1 to:ligne do:[:lig| self at:lig
                            put:(Array new:colonne)].
  ^self
```

Figure 3.14 : méthode d'instance *ligne:colonne:*

La méthode *ligne:colonne:* (figure 3.14) fixe d'abord les valeurs des variables d'instance *ligne* et *colonne* puis affecte à chaque élément du tableau receveur un tableau vide de taille *colonne*. La matrice est prête à recevoir ses valeurs numériques.

Charger les valeurs numériques dans une matrice

Nous pouvons charger les valeurs d'une matrice élément par élément. Cette opération risque d'être fastidieuse. Nous pouvons aussi les charger ligne par ligne (ou colonne par colonne). Enfin, nous pouvons transférer le contenu d'un tableau dans une matrice, pour peu que nous respections des conventions préétablies.

Dans tous les cas, nous aurons besoin d'accéder à un élément précis de la matrice, défini par ses coordonnées en ligne et colonne.

La méthode *auxCoordonnees:* décrite figure 3.15 effectue un accès en deux étapes pour accéder à l'élément dont les coordonnées sont précisées dans l'argument *unTableau*. Le premier appel de *at:* renvoie la ligne en position (*unTableau at:1*) ; l'appel suivant extrait l'élément de position (*unTableau at:2*) dans cette ligne.

```
auxCoordonnees:unTableau
  "renvoie la valeur du récepteur aux positions indiquées dans unTableau"
  ^(self at:(unTableau at:1)) at:(unTableau at:2)
```

Figure 3.15 : méthode *auxCoordonnees:* pour accéder à un élément de la matrice

La méthode symétrique *auxCoordonnees: unTableau met: uneValeur* (figure 3.16) remplace la valeur de position *unTableau* par *uneValeur*.

```
auxCoordonnees:unTableau met:uneValeur
  "remplace la valeur du récepteur aux positions indiquées dans unTableau
   par uneValeur"
  ^(self at:(unTableau at:1)) at:(unTableau at:2)
                              put:uneValeur
```

Figure 3.16 : méthode *a:met:*

La modification est réalisée dans le tableau de la ligne considérée (ligne de numéro *(unTableau at:1)*) par la méthode *at:put:*.

Entrer les éléments de la matrice un par un n'est pas pratique. La méthode *prendLesValeurs: tableValeurs* (figure 3.17) charge en une seule fois toutes les valeurs de la matrice stockée dans *tableValeurs*, instance de *Array*.

```
prendLesValeurs:tableValeurs
  "répartit les valeurs de tableValeurs dans la matrice ligne par ligne. La taille
   de tableValeurs doit être cohérente avec les dimensions de la matrice"
  |j|
  j:=1. "la variable j marquera la position courante dans tableValeurs"
  (ligne * colonne = tableValeurs size)
  ifFalse:[^self error:'Hors dimension de la matrice'].
  "la taille du tableau est cohérente avec les dimensions de la matrice, on remplit
   donc celle-ci ligne après ligne avec l'extrait correspondant du tableau tableValeurs"
  1 to:ligne do:[:i|
       self at:i
            put:(tableValeurs copyFrom:j
                              to:(j+colonne - 1)).
       j:=j+colonne].
  ^self
```

Figure 3.17 : méthode *prendLesValeurs:*

La méthode *prendLesValeurs:tableValeurs* (figure 3.17) nous permet d'introduire une nouvelle méthode d'instance de *Array* : *copyFrom:firstIndex to: lastIndex.* Comme son nom l'indique, cette méthode renvoie une copie du récepteur comprise entre ses éléments de rang *firstIndex* et *lastIndex.* Les variables d'instance *ligne* et *colonne* sont utilisées dans *prendLesValeurs* pour

découper en *ligne* tableaux de *colonne* éléments la liste de valeurs *tableValeurs*. Pour créer une matrice, nous pouvons maintenant évaluer :

```
|x|
x := Matrice nouveau:#(4 2).
x prendLesValeurs:#(1 0 2 0 1 0 1 0 ).
```

Le résultat affiché à l'écran après évaluation avec l'option *show it* est :

```
Matrice((1 0)(2 0)(1 0)(1 0))
```

La représentation de la matrice à l'écran peut être améliorée en redéfinissant la méthode *printOn:* dans la classe *Matrice*. La méthode *printOn:* est appelée lors de l'évaluation d'une séquence par *show it*. Son rôle est d'afficher le contenu du récepteur (en l'occurrence notre matrice) dans un éditeur de texte (la fenêtre de travail)[8]. Nous proposons dans la figure 3.18 une redéfinition possible de cette méthode qui permet d'obtenir la matrice précédente sous la forme :

```
(1 0
 2 0
 1 0
 1 0)
```

La méthode *printOn:* faisant appel aux flux, son contenu ne sera pas étudié dans le cadre de ce chapitre.

```
printOn:aStream
  "Imprime les valeurs de la matrice présentées sous forme habituelle"
  aStream cr.
  aStream nextPut:$(.
  1 to:ligne do:[:lig|
       1 to: colonne do:[:col|
               (self auxCoordonnees:(Array with:lig
                                            with:col))
                      printOn:aStream.
                      "printOn: est adressé à l'entier de position (lig col)
                       dans la matrice réceptrice. Il est donc fait appel à la
                       méthode printOn: définie dans la classe Integer"
                aStream tab].
       (lig = ligne) ifTrue:[aStream nextPut:$)].
       aStream cr
                ].
  ^self
```

Figure 3.18 : amélioration de l'affichage des matrices à l'écran

8 La méthode printOn: envoie une représentation imprimable du récepteur dans un flux, qui peut être récupérée par une fenêtre d'éditions de texte. Cette méthode sera abordée dans le chapitre 5, consacré aux flux.

Multiplier deux matrices

Nous pouvons maintenant construire une méthode permettant de multiplier deux matrices (figure 3.19).

```
* uneMatrice
 "renvoie le produit de la matrice réceptrice par la matrice argument, en vérifiant que
  leurs dimensions sont compatibles"
 | resultat somme |
 (self colonne = uneMatrice ligne)
 ifFalse:[^self error:'Impossible de multiplier deux
                       matrices de ces dimensions'].
 "la matrice résultante sera stockée dans resultat, qui est donc créée aux bonnes
  dimensions"
 resultat := Matrice deDimensions:
                      (Array with:self ligne
                             with:uneMatrice colonne).
 1 to:self ligne do:[:i|
 "i est l'indice des lignes de la matrice de gauche"
    1 to:uneMatrice colonne do:[:j|
    "j est l'indice des colonnes de la matrice de droite"
       somme := 0.
       "somme stocke temporairement la valeur à mettre dans la matrice résultat"
       1 to:self colonne do:[:k|
       "k est l'indice de la valeur à multiplier"
          somme := somme +
                ((self auxCoordonnees:
                       (Array with:i with:k))
                * (uneMatrice auxCoordonnees:
                       (Array with:k with:j)))].
          "fin du calcul de la valeur à charger. Il reste à placer cette valeur dans
           la matrice resultat"
       resultat auxCoordonnees:(Array with:i with:j)
                met:somme]
    "fin d'une ligne de la matrice resultat"
 ]
 "fin de toutes les lignes de la matrice resultat"
 ^resultat
```

Figure 3.19 : multiplier deux matrices

Cette méthode est particulièrement démonstrative des difficultés que le débutant peut rencontrer lorsqu'il doit donner un tableau en argument d'une méthode. La matrice *resultat* est créée par l'envoi du message *deDimensions:unTableau* à la classe *Matrice* (*resultat := Matrice deDimensions: (Array with: self ligne with: uneMatrice colonne)*). Dans notre cas, nous ne connaissons pas à l'avance la taille de la nouvelle matrice à créer. *unTableau* contient donc deux variables (*self ligne* et *uneMatrice colonne*). Mais nous avons

vu qu'il est impossible de créer directement une instance de la classe *Array*, comme nous serions tentés de le faire avec :

```
#(self ligne uneMatrice colonne)
```

L'évaluation de cette ligne renverra un tableau contenant quatre symboles (*self*, *ligne*, *uneMatrice* et *colonne*)[9]. Il faut donc passer par la classe *Array*, lui envoyer un message (*with:with:*) qui acceptera en paramètre le résultat des messages *self ligne* et *uneMatrice colonne*.

Dans ce cas précis, l'objet paramètre *unTableau* n'a que deux éléments. Nous pouvons donc utiliser directement la méthode *with:with:* pour créer et affecter des valeurs à un tableau en une seule opération. Mais la classe *Collection* (et *Array* par voie de conséquence) ne dispose que de quatre méthodes de ce type (*with:*, *with:with:*, *with:with:with:*, *with:with:with:with:*). Pour des tableaux ayant plus de quatre éléments, il faudra couper l'opération en :

```
unTableau := Array new: 5.
unTableau at:1 put:objet1;
          at:2 put:objet2;
          at:3 put:objet3;
          at:4 put:objet4;
          at:5 put:objet5.
```

La méthode * fait aussi appel à *auxCoordonnees:* et *auxCoordonnees:met:* qui prennent en paramètre un tableau dont les éléments sont les variables locales *i*, *j* et *k*. Nous utilisons de nouveau la séquence `Array with:i with:j` pour obtenir un tableau ayant comme éléments les objets pointés par les variables *i* et *j*.

Pour vérifier le fonctionnement de la méthode, évaluons avec l'option *show it* :

```
|x y|
x := Matrice deDimensions:#(4 2).
x prendLesValeurs:#(1 0 2 0 1 0 1 0 )
y := Matrice deDimensions:#(2 3).
y prendLesValeurs:#(1 0 0 0 1 0).
x * y
```

Smalltalk renverra :

```
(1   0   0
 2   0   0
 1   0   0
 1   0   0)
```

3.3 - Les chaînes de caractères

Dans la hiérarchie des classes, *String* est située au même niveau que la classe *Array* (figure 3.20) et hérite des mêmes classes que celle-ci.

---

9 Smalltalk considère que le contenu des parenthèses suivant le signe # est la représentation exacte du tableau. A part une vérification syntaxique, il ne lance aucune évaluation de l'expression comprise entre les parenthèses.

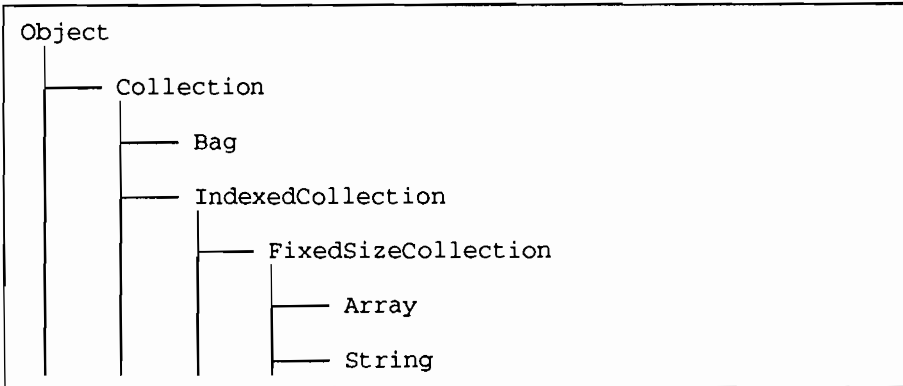


Figure 3.20 : la classe *String* dans la hiérarchie des classes

Nous avons vu dans les chapitres précédents quelques exemples de chaînes de caractères. Nous allons présenter dans ce chapitre de nouvelles méthodes utilisables dans la classe *String*. Nous nous servirons ensuite de cette classe pour apprendre à créer, modifier, détruire ou récupérer une sous-classe et ses méthodes associées.

3.3.1 - Quelques manipulations de chaînes de caractères

Traiter des informations dans une chaîne de caractères[10]

Outre les méthodes héritées de ses classes mères (que nous verrons au chapitre 4), *String* permet de travailler sur les chaînes pour en extraire un tableau de mots[11] (*asArrayOfSubstrings*), un nom de fichier MS-DOS (*fileName*) ou le suffixe d'un tel nom (*fileExtension*).

L'évaluation par *show it* de :

```
('nomfic.ext  23456  23/12/90  12:00:00'
  asArrayOfSubstrings     "transforme la chaîne en tableaux de sous-chaînes"
     at:1)                "renvoie le premier élément du tableau, donc la
                            première sous-chaîne"
  fileName                "renvoie le champ correspondant au nom du fichier
                           MS-DOS"
  asUpperCase.            "renvoie le nom du fichier en majuscules"
```

extrait, d'une ligne analogue à celle que l'on pourrait obtenir avec la commande MS-DOS *dir*, le nom d'un fichier converti en majuscules (*NOMFIC*).

---

10 L'exemple sera tiré du système d'exploitation MS-DOS et de Smalltalk V.

11 Ce ne sont pas des mots au sens linguistique du terme, mais une suite de caractères compris entre deux espaces de la chaîne d'origine.

A partir de ces méthodes, l'utilisateur peut se construire un environnement personnalisé, capable de traiter les chaînes de caractères de ses fichiers texte (recherche et remplacement de mots...), tableur, base de données etc.

Améliorer la présentation

withCrs remplace dans une chaîne toutes les occurrences du caractère \ par un code lf (retour de ligne). Cette méthode est intéressante car elle permet de noter sur une seule ligne une suite de mots à disposer en colonne. Toutes les méthodes d'affichage des menus l'utilisent.

Ainsi `'Remove\New\Create\Exit' withCrs` présentera la chaîne de la manière suivante :

```
Remove
New
Create
Exit
```

trim: aCharacter renvoie la chaîne réceptrice élaguée des caractères dont le code ASCII est inférieur ou égal à celui de *aCharacter*. La suppression s'effectue à la fois au début et à la fin de la chaîne et s'interrompt au premier caractère dont le code est strictement supérieur à celui de *aCharacter*.

Ainsi
```
'PLANTEZ CE CLOU A DEUX METRES',
      ('DE LA BASE' trim:$E)
```

renvoie 'PLANTEZ CE CLOU A DEUX METRES LA BAS', ce qui n'est pas la même chose[12]. Nous voyons que les deux lettres *A* (dont le code est inférieur à celui de *$E*) de *LA BAS* ont été maintenues dans la chaîne renvoyée après évaluation du message *trim:$E*. La suppression des caractères s'est interrompue au *L* en début de chaîne et au *S* en fin de chaîne.

trimBlanks supprime les blancs de début et de fin.

edit crée une fenêtre éditeur de texte sur l'écran et y affiche la chaîne.

outputToPrinter imprime la chaîne sur l'imprimante.

printOn:aStream charge la chaîne dans le flux aStream (voir le chapitre 5).

Et, comme curiosité, la figure 3.21 illustre quelques possibilités d'affichage supplémentaires.

---

12 Nous verrons au chapitre 4 les méthodes de la classe Collection. Le sélecteur , (virgule) correspond à une de ces méthodes. Il permet d'accoler la chaîne donnée en paramètre à la chaîne réceptrice pour en faire une chaîne unique.

Figure 3.21 : affichage d'une chaîne à l'écran

3.3.2 - Problèmes d'héritage liés à la classe String

La notion d'héritage incite le programmeur à créer une nouvelle sous-classe lorsqu'il désire modifier une méthode d'une classe donnée. En effet, les classes sont des objets partagés par tous les utilisateurs. Il est pratiquement certain que la modification d'une classe existante, même mineure, aura des conséquences imprévisibles sur le déroulement d'autres méthodes. L'héritage permet alors au programmeur de protéger l'environnement commun, tout en créant des objets qui bénéficieront de cet environnement.

Pourtant, le respect de cette règle risque de donner bien des soucis comme nous allons l'illustrer par un exemple tiré de la classe *String.*

Supposons que nous voulions manipuler des mots de la langue française. Toutes les opérations de comparaison nécessiteront probablement le passage en majuscules (pour se libérer des choix typographiques propres à chaque opérateur) : mais nous avons vu précédemment (Cf 1.6) que la fonction *asUpperCase* proposée dans Smalltalk V ne sait traiter qu'un alphabet anglo-saxon. Elle ignore le traitement particulier des caractères accentués dans la langue française. Nous pouvons le vérifier une nouvelle fois avec :

```
'Huîtres à Noèl, oeufs à pâques et pâture en été'
asUpperCase
```

qui affiche après évaluation par *show it* :

```
HUÎTRES à NOëL, OEUFS à PâQUES ET PâTURE EN éTé
```

Nous allons donc ajouter une sous-classe à la classe *String* : *MotFrancais* par exemple[13] (option *variableByteSubclass*[14]) dans laquelle nous ajouterons la version française de *asUpperCase*.

Dans *String* (et donc dans *MotFrancais* qui en hérite), les méthodes de classe permettant de créer un objet sont peu pratiques. Héritées de la classe *collection*, elles limitent la taille des mots à quatre lettres (méthodes de classe *with:* à *with:with:with:with:*) ou imposent une écriture rebutante au programmeur (méthode de classe *new:* qui nécessite de connaître au moins approximativement la taille de la chaîne à créer, suivie d'autant de messages *at: position put: lettre* qu'il y a de caractères dans la chaîne).

En réalité, dès qu'il s'agit d'affecter un objet chaîne à une variable, le moyen le plus immédiat est d'utiliser une constante :

```
|chaine|
chaine := 'Les sanglots longs des violons de l''automne'.
```

Smalltalk associe automatiquement la variable *chaine* à un nouvel objet de la classe *String* de valeur *Les sanglots longs des violons de l'automne* (de la même façon, il interprétera automatiquement un nombre comme une instance de la classe *Integer*).

Mais comment créer simplement une instance de *MotFrancais* ? Employer le symbole d'affectation de Smalltalk ne donnera qu'une instance de *String* :

```
|mot|
mot := MotFrancais new:12.
mot := 'mot français'.
mot class
```

ne renverra jamais *MotFrancais* mais *String*, car la deuxième affectation interprétée par Smalltalk associe obligatoirement le contenu des apostrophes à la classe *String*.

Les méthodes de classe héritées des classes mères de *String* (voir chapitre 4) sont d'un emploi trop fastidieux. Il est donc nécessaire de disposer d'une méthode de classe qui puisse récupérer une chaîne existante (de la classe *String*) pour la dupliquer dans une instance de *MotFrancais*. Nous allons créer la méthode de classe *aPartirDe:* dans *MotFrancais* (figure 3.22).

---

13 Notre souci de préserver la pureté de la langue française aurait dû nous inciter à écrire cette nouvelle classe MotFrançais et non MotFrancais. Malheureusement, l'origine anglo-saxonne de Smalltalk V limite la compréhension de son analyseur syntaxique aux seuls caractères anglo-saxons. L'évaluation de x := MotFrançais new:12 s'interromprait sur ç en signalant 'should be selector'.

14 Nous verrons la signification de cette option au paragraphe suivant (3.3.4), consacré aux manipulations de classes.

```
aPartirDe: uneChaine
  "crée une nouvelle instance de MotFrancais à partir de uneChaine "
  |phrase|
   phrase := MotFrancais new: uneChaine size.
   1 to: (uneChaine size)
     do:[:indice| phrase at:indice
                         put:(uneChaine at:indice)].
   ^phrase
```

Figure 3.22 : la première méthode de *MotFrancais*

L'évaluation avec l'option *show it* de l'expression :

```
(MotFrancais aPartirDe:'Noël à Pâques') class
```

donne bien *MotFrancais.* Notre conversion a réussi. Le programmeur pourrait maintenant construire les méthodes particulières au traitement des mots français (méthode *enMajuscules* etc.).

La seconde difficulté que rencontre l'utilisateur lorsqu'il crée une sous-classe de *String* réside dans la manière particulière dont est redéfinie la méthode = pour la classe *IndexedCollection* et ses sous-classes (voir chapitre 4). Non seulement le contenu de la collection est comparé objet par objet, mais il est nécessaire que les deux instances appartiennent à la même classe. Autrement dit :

```
(MotFrancais aPartirDe:'identique')='identique'
```

rendra *false.*

Il faut donc redéfinir la méthode = dans la classe *MotFrancais* de manière à ce qu'elle se contente de comparer les caractères composant les deux chaînes, sans faire intervenir leurs classes (figure 3.23).

Malgré cette précaution, l'évaluation avec *show it* de :

```
|mot word|
word := 'waterloo'.
mot := MotFrancais aPartirDe 'waterloo'.
word = mot
```

ne donnera ni *'austerlitz'* ni le mot de Cambronne mais *false.*

L'expression *word = mot* représente l'envoi du message = *mot* à l'objet *word,* instance de *String.* C'est donc la méthode = de *String* qui est utilisée, où l'appartenance à une même classe est une condition nécessaire pour renvoyer *true.*

```
= unMot
"renvoie true si le récepteur et unMot sont composés des mêmes caractères,
 false sinon, indépendamment de la classe mère du récepteur et de unMot"
| taille position |
"taille va stocker la longueur de la chaîne, position la position courante du caractère
 examiné"
((taille:=self size)~= unMot size) ifTrue:[^false].
"si les chaînes n'ont pas la même taille, alors elles sont différentes"
position := 1. "on se place au premier caractère"
"tant qu'il y a des caractères dans la chaîne:"
[position <= taille] whileTrue:[
   ((self at:position) = (unMot at:position))
     ifFalse:[^false] "au moins un caractère est différent"
     ifTrue:[position:=position+1] "le dernier caractère étudié est
                                    identique, on avance d'une
                                    position"].
^true " tous les caractères sont identiques, les deux chaînes le sont donc aussi"
```

Figure 3.23 : redéfinition de la méthode = dans la classe *MotFrancais*

Nous pourrions bien sûr modifier à son tour la méthode = dans la classe *String*. Ce serait contraire à notre souci d'intégrité des classes d'origine. Pour ces opérations de comparaison, une méthode de conversion renvoyant une instance de *String* à partir d'une instance de *MotFrancais* peut être utile (figure 3.24).

```
asString "méthode d'instance de MotFrancais"
 "renvoie la chaîne de caractères composant le récepteur dans une instance de String"
   |chaine|
    chaine := String new: self size.
    1 to: (self size) do:[:indice|
        chaine at:indice
               put:(self at:indice)].
   ^chaine
```

Figure 3.24 : traduire une instance de *MotFrancais* en instance de *String*

```
|mot word|
word := 'waterloo'.
mot := Motfrancais aPartirDe 'waterloo'.
word = mot asString
```

renvoie maintenant *true*.

L'examen d'une fenêtre d'édition des classes (*Class Hierarchy Browser*) montre d'ailleurs que de nombreuses méthodes de conversion sont disponibles dans la classe *String*, qui permettent de traduire une chaîne en entier, en majuscule, en minuscule, en symbole, en date, en heure, en flux (classe *Stream* étudiée au chapitre 5). Nous y trouvons même la méthode *asString* : elle n'est toutefois pas utili-

sable pour traduire les instances de *MotFrancais*, puisque son seul effet est de renvoyer le récepteur tel quel.

D'une manière générale, le programmeur doit rester attentif à la nature des objets qu'il manipule. La construction de méthodes comparant des chaînes de caractères nécessite la plupart du temps l'appel à des méthodes de conversion (en majuscules, en symbole, en chaîne).

Il faut donc que ces méthodes de conversion soient présentes dans toutes les classes contenant les chaînes comparées. C'est ce que nous avons fait pour *MotFrancais* en lui ajoutant la méthode *asString* (les autres sont héritées de *String*). Le polymorphisme de la méthode *asString* impose d'ailleurs qu'elle soit définie *aussi* dans la classe *String*. L'expression *objet asString* est toujours comprise et son évaluation renvoie systématiquement une instance de *String*, et ce, qu'*objet* soit une instance de *String* ou non.

3.3.3 - Les arguments d'une méthode : variables ou objets ?

Nous avons abordé au chapitre 2 les notions élémentaires de la programmation orientée objets et leur implémentation dans Smalltalk. Nous avons vu la distinction à faire entre un objet et une variable. Ces conceptions surprennent parfois le programmeur formé aux langages structurés. Nous allons revenir dans ce paragraphe sur quelques difficultés qu'il peut rencontrer.

Un programme Smalltalk débute en général par une séquence d'instructions comparable à celle-ci :

```
|v|
v := o
```

Nous avons signalé, au début de cet ouvrage, l'abus de langage que nous commettions en parlant de l'objet *v* alors que *v* n'était qu'une variable désignant un objet *o*. La plupart du temps, ce manque de rigueur ne prête pas à conséquence. Les exercices qui suivent montrent que ce n'est pas toujours le cas.

Une variable peut désigner des objets différents dans une même séquence

Nous avons vu que la séquence suivante est acceptée par Smalltalk :

```
|mot|
mot := MotFrancais new:12.
mot := 'mot français'
```

mot étant une variable, elle peut recevoir successivement plusieurs valeurs. Ces valeurs sont des objets dont l'existence est liée à la durée de vie de la variable qui les pointe (l'instance de *MotFrancais*, vide de caractères, sera détruite lorsque la variable *mot* désignera *'mot français'* ; l'instance de *String* *'mot français'* sera supprimée une fois la variable *mot* disparue de l'environnement, en fin d'évaluation). L'instance de *MotFrancais* ne change pas de classe pour appartenir à *String* à la suite de la deuxième affectation : elle est détruite.

De même, la séquence suivante est tout à fait valide :

```
|mot|
mot := 'Le jour '.
mot := mot,'le plus long'
```

Pourtant, le signe *mot* n'est pas interprété exactement de la même façon par Smalltalk selon qu'il est à droite ou à gauche du signe d'affectation. Dans la dernière expression, c'est l'objet désigné par la variable *mot* (à droite du signe :=) qui reçoit le message binaire *,'le plus long'* alors que c'est à la variable *mot* (à gauche du signe :=) que l'on affecte le résultat de ce message binaire.

Une variable transmise en argument d'une méthode est considérée comme un objet

Nous avons présenté au paragraphe 3.3.1 la méthode *printOn:* dont le rôle est d'ajouter le récepteur (en l'occurrence une chaîne de caractères) au flux donné en argument.

Essayons de construire dans la classe *MotFrancais* une méthode comparable qui ajoutera le récepteur à un texte (instance de *String*) donné en argument. La figure 3.25 présente le corps de cette méthode (*ajouteA:*) qu'un programmeur expérimenté en langages classiques pourrait être tenté de construire.

Nous espérons ainsi pouvoir évaluer une séquence semblable à :

```
| x y |
x := MotFrancais aPartirDe:'peau d'âne'.
y := 'Mots dont l'orthographe est modifiée : '
x ajouteA:y.
```

x et *y* sont deux variables auxquelles nous affectons temporairement la valeur de deux objets (instances de la classe *MotFrancais* et de la classe *String*). Nous les assimilons à des objets.

```
ajouteA:unTexte
  "ajoute le contenu du récepteur  à la fin de unTexte"
  unTexte := unTexte,self
```

Figure 3.25 : exemple de méthode illégale

Il nous est impossible de sauvegarder cette nouvelle méthode : l'analyse effectuée par Smalltalk s'interrompt et un message d'erreur nous signale que l'affectation *unTexte :=* est impossible dans ce contexte.

La méthode *ajouteA:* réalise pourtant un travail qui ne pose aucun problème si nous l'effectuons directement dans une fenêtre de travail :

```
| x y |
x := MotFrancais aPartirDe:'peau d'âne'.
y := 'Mots dont l'orthographe est modifiée : '
y := y,x.
y
```

L'évaluation de cette séquence par l'option *show it* montre que la variable *y* contient maintenant le texte *'Mots dont l'orthographe est modifiée : peau d'âne'*.

L'échec de la méthode *ajouteA:* vient de ce que Smalltalk lui-même considère les variables x et y comme des objets (dont il doit respecter l'intégrité) à certains moments :

- *y*, paramètre donnée du message *ajouteA:y*, ne représente plus la variable *y* (donc le pointeur sur l'objet désigné par *y*) mais l'objet lui même dès que Smalltalk évalue la séquence de messages contenue dans la méthode *ajouteA:*
- l'analyse effectuée par Smalltalk rejette donc toute affectation susceptible de modifier l'intégrité des objets manipulés dans la méthode.
- de la même manière, l'expression *self* := serait refusée.

Nous verrons plus loin qu'il existe pourtant des méthodes (qui font appel à des primitives systèmes) qui permettent de forcer la réaffectation d'un objet[15].

En langage orienté « objets », l'objet est strictement responsable de son intégrité : l'objet argument d'un message ne peut être modifié sans « donner son accord ». Nous pouvons vérifier que la modification de l'argument (*aStream*) est réalisée par l'envoi d'un message à cet argument (*aStream nextPut: character*). En définitive, c'est bien l'objet lui-même qui s'est modifié.

3.4 - Créer une classe, sauvegarder et récupérer ses méthodes

3.4.1 - Quel type de classe créer ?

Lorsque nous avons créé *MotFrancais*, sous-classe de *String*, en utilisant les menus d'une fenêtre d'édition des classes (*Class Hierarchy Browser*), il nous a fallu préciser quel type de sous-classes nous voulions définir (Cf 3.3.2).

Nous avons remarqué la grande variabilité des objets gérés par Smalltalk. Certaines classes, comme *Character* ou *Integer* ont des instances de taille fixe et prévisible. D'autres, comme *String* ou *ByteArray* ont des instances de taille fixe mais différentes entre elles. Certaines, comme *Bag*, ont des instances de taille variable.

Il est donc utile de savoir quel type d'instance une classe va contenir, afin de préparer au mieux la mémoire de la machine à la création d'objets de cette classe.

La stratégie d'utilisation de la mémoire prend quatre formes sous Smalltalk :

- les objets sont prédéfinis, et il n'y a pas lieu de créer de nouvelles instances de la classe : c'est le cas de *Character* ;
- les objets sont considérés comme prédéfinis, mais n'apparaissent réellement en mémoire que si la nécessité l'impose : c'est le cas des sous-classes de la classe *Integer*, pour laquelle la création d'instance est impossible[16] ;

---

15 C'est le cas de la méthode become:, qui sera abordée en 4.9.3.

16 Leur utilisation est bien entendu possible. Lorsque le programmeur fait appel à un objet de cette classe (comme dans l'expression |x y| x:=12345. y:=x*x), l'instance est créée en mémoire (dans notre exemple, la variable x pointera sur un objet de la classe Integer et de valeur 12345 qui n'existait pas réellement en mémoire avant l'évaluation de l'expression).

- l'objet a une taille fixe qui est déterminée à sa création par le message *new:n*. Smalltalk réserve alors un espace mémoire de *n* cases pour y stocker les *n* éléments de l'objet (octets dans le cas de la classe *ByteArray*, caractères dans le cas de *String*). Il est impossible de modifier la taille de l'objet après sa création. Des méthodes d'instance permettent ensuite d'accéder à telle ou telle case dans l'espace réservé à l'instance. On dit des classes qui possèdent ce type d'objet qu'elles autorisent l'emploi de « variables d'instance indexées » : ces variables sont numérotées de 1 à *n* et désignent le contenu de chacune des cases mémoire de l'instance. La méthode *at:put:* de *String* est une méthode typique de ce genre de classe, puisqu'elle accède directement au contenu d'une case de l'instance ;
- l'objet est composé de plusieurs éléments (*n* au maximum[17]) de taille variable. Smalltalk réserve un espace mémoire de *n* cases pour y stocker les *n* pointeurs (éventuels) sur les éléments constitutifs de l'objet. La valeur des pointeurs est initialisée au fur et à mesure du chargement des éléments[18] (ex.: classe *Bag*) ;

Nous voyons que Smalltalk permet de gérer des classes contenant :

- des objets de taille fixe (byte ou word[19]) ;
- des objets composés d'une suite d'objets de taille fixe (byte ou word);
- des objets composés d'une suite d'objets de taille variable.

Dans le dernier cas, le processeur range les objets composant l'instance dans des endroits à part, et stocke uniquement les pointeurs sur ces objets dans l'instance.

Il est nécessaire de signaler à Smalltalk le type d'objet qu'une classe contiendra, pour préparer le processeur à traiter correctement ses instances. Dans le cas de Smalltalk V[20], il existe quatre types de sous-classes :

subclass — ses instances ont une taille fixe. Elles n'ont pas de variables indexées (car elles ne sont pas composées d'une suite d'éléments) mais peuvent avoir des variables nommées[21]. Ce n'est pas la valeur elle-même de l'instance qui est conservée dans l'espace mémoire alloué par le processeur, mais son adresse.

variableSubclass — ses instances ont des variables indexées (elles sont composées d'une suite de cases correspondant chacune aux éléments qui les composent) et peuvent avoir des variables nommées. Chaque case (accessible par une variable d'instance indexée) contient un pointeur sur un élément de l'instance (ou sur

17 Nous verrons au chapitre 4 qu'il est possible de mettre plus de n éléments dans une instance initialement prévue pour en recevoir au maximum n (message new:n) : Smalltalk crée une copie de l'instance en voie de saturation en calculant plus largement sa taille.

18 Les pointeurs non affectés désignent l'objet nil.

19 Octet ou mot machine.

20 C'est aussi le cas pour Smalltalk 80.

21 Nous avons vu au chapitre 1 le rôle de ces variables d'instance nommées.

l'objet *nil* si cette case n'est pas encore attribuée à un élément).

variableByteSubclass
ses instances n'ont que des variables indexées (aucune variable nommée), et la valeur de chaque élément (un octet, soit huit bits) est directement stockée dans la case mémoire correspondante.

variableWordSubclass
ses instances n'ont que des variables indexées (aucune variable nommée), et la valeur de chaque élément (un mot, soit seize bits) est directement stockée dans la case mémoire correspondante.

La différence entre les deux dernières catégories correspond aux particularités du processeur des machines pour lesquelles ce logiciel a été conçu[22]. Les octets permettent de représenter tous les caractères du code ASCII (256 valeurs différentes). Les mots de deux octets sont utilisés pour coder les entiers longs (*LargePositiveInteger*, *LargeNegativeInteger*)[23].

Pour bien faire la distinction entre ces différents types de classe, nous allons essayer de créer une sous classe de type *variableSubclass* dans *String* (qui est définie comme *variableByteSubclass*). Un message d'erreur s'affiche, signalant *Superclass is non-pointers*. En effet, les instances de *String* sont composées d'une suite d'octets. Des instances composées d'une suite d'adresses (*variable Subclass*) ne peuvent hériter de la classe *String*[24].

Si nous essayons de nouveau en choisissant *subclass*, un message nous signale que le processeur nous imposera un type de classe autorisant les variables indexées (puisque *String*, la classe mère, a déjà ce type de variables). Après accord, le message d'erreur *Superclass is non-pointers* s'affiche de nouveau : *subclass* associe des adresses à ses instances, et non des valeurs.

Enfin, si nous essayons avec *variableWordSubclass*, un autre message d'erreur apparaît : *Superclass is Bytes*.

Un examen attentif d'une fenêtre d'éditions des classes (*Class Hierarchy Browser*) montre que les classes de type *subclass* sont les seules qui permettent de définir n'importe quel type de sous-classe.

---

22 Cette distinction disparaît dans la version Smalltalk V 286 pour processeur Intel 80286.

23 Ils forment les briques élémentaires du codage de ces entiers longs. Smalltalk V permet ainsi de coder un entier dont la représentation binaire atteint une taille de 64 kilo-octets (32768 mots machine) !

24 De même, variableByteSubclass ne peut descendre de variableSubclass. Les variables indexées ne sont pas en cause, les deux classes les autorisant. Mais la classe mère traite des adresses, la sous-classe des valeurs.

3.4.2 - Comment sauvegarder une classe et ses méthodes ?

Le menu de gestion des classes du *Class Hierarchy Browser* offre une option qui permet de sauvegarder la description d'une classe (définition et méthodes seulement : les instances existantes ne sont pas sauvegardées). Il s'agit de "*file out*" .

Cliquer sur cette option amène l'ouverture d'une fenêtre réclamant le nom du fichier de sauvegarde. La figure 3.26 reprend un extrait du fichier de sauvegarde de la classe *Array*.

```
FixedSizeCollection variableSubclass: #Array
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!Array class methods ! !


!Array methods !

printOn: aStream
        "Append the ASCII representation
         of the receiver to aStream."
    (RecursiveSet includes: self)
        ifTrue: [^self printRecursionOn: aStream].
    RecursiveSet add: self.
    aStream nextPut: $(.
    1 to: self size - 1 do: [ :element |
        (self at: element) printOn: aStream.
        aStream space].
    self isEmpty
        ifFalse: [
            self last printOn: aStream].
    aStream nextPut: $).
    RecursiveSet remove: self ifAbsent: []!
...
```

Figure 3.26 : début du fichier de sauvegarde de la classe *Array*

Dans ce fichier sont énumérées les définitions de la classe et de ses méthodes telles qu'elles apparaissent dans les fenêtres du *Class Hierarchy Browser*, à une exception près : les paragraphes sont délimités par le signe !.

Ce signe ! est la marque qui permet à Smalltalk de sectionner le fichier en modules homogènes, correspondant chacun à une méthode ou une définition.

3.4.3 - Restaurer une classe

Il existe dans Smalltalk une classe spécialisée dans la manipulation de fichiers : *FileStream*. Cette classe présente un certain nombre de méthodes, dont

une bonne proportion est réservée à l'interprétation de fichiers-sources de Smalltalk.

Pour récupérer une classe, ses méthodes et ses descendants il suffit d'utiliser la méthode *fileIn*, comme dans l'exemple proposé ci-dessous :

```
(File pathName: 'd:\array.cls') fileIn; close.
```

La méthode *pathName:* appliquée à la classe *File* renvoie une instance de *FileStream* ouverte sur le fichier dont le nom a été donné en paramètre. La méthode *fileIn* s'occupe de charger et mettre à jour les classes décrites dans le fichier qui sera ensuite refermé par le message *close*.

3.5 - Les dictionnaires de Smalltalk

Nous allons maintenant aborder une classe qui permet de gérer une collection d'informations avec un accès associatif. Cette classe permet d'accéder à chaque information dès lors qu'on en connaît la partie qui sert de critère d'accès et que l'on appelle la *clé*. Il s'agit de la classe *Dictionary*, dont chaque instance rassemble des informations qui sont elles-mêmes des instances de la classe *Association*. La classe *Dictionary* est souvent utilisée, dans de nombreuses applications, y compris par Smalltalk lui-même, qui gère l'ensemble des classes et des variables globales à l'aide d'un dictionnaire (Cf 3.5.7).

3.5.1 - La classe Association

Cette classe permet de représenter des couples d'informations *clé/valeur*, en vue d'un accès associatif : si on connaît *clé*, on peut obtenir *valeur*. Une instance de la classe *Association* peut être créée avec la méthode de classe *key:value:* comme le montre l'évaluation de la première expression de la figure 3.27. Chaque objet de la classe *Association* possède deux variables d'instances *key* et *value* qui désigneront respectivement la clé et la valeur de l'association. Dans l'exemple de la figure 3.27, l'objet *a1* est l'association *France* ⇨ *Paris* (le symbole ⇨ est habituellement utilisé par Smalltalk pour exprimer la relation clé/valeur ; dans les écrans de Smalltalk V, il est représenté par la combinaison de caractères ==>).

```
|a1 a2 a3|
a1 := Association key: 'France'  value: 'Paris'.
        "a1 est l'association 'France' ⇨ 'Paris'"
a2 := Association key: 'Italie'.
        "a2 est l'association 'Italie' ⇨ nil"
a3 := Association new.
        "a3 est l'association nil ⇨ nil"
```

Figure 3.27 : création d'instances de la classe Association

Trois méthodes de classe permettent de créer des instances de la classe *Association*. Nous venons d'étudier la première d'entre elles (sélecteur *key:value:*).

La seconde est la méthode *key:* qui permet de créer une instance dont la valeur reste indéterminée, comme le montre l'instanciation de *a2* à la figure 3.27. La dernière est la méthode *new*, héritée de la classe *Object*, qui permet de créer une association dont la clé et la valeur sont indéterminées (Cf instanciation de *a3*, sur la figure).

La classe *Association* est une sous-classe de la classe *Magnitude* (Cf 1.6) et, à ce titre, elle hérite des méthodes <, <=, =, >, >= qui permettent de comparer entre elles deux associations. Dans cette classe, ces méthodes sont toutes redéfinies pour exprimer la comparaison sur les clés (et non sur les valeurs), ce qui est conforme au principe de l'utilisation de la clé pour l'accès associatif. Ainsi, la méthode < est-elle définie de la manière suivante :

```
< anAssociation
    "Answer true if the receiver key is less than anAssociation key, else answer false."
    ^ key < anAssociation key
```

Les autres méthodes d'instance permettent notamment d'obtenir ou de modifier la clé d'une association (*key* et *key:*), d'obtenir ou de modifier la valeur d'une association (*value* et *value:*). Par exemple, si l'on souhaitait compléter la séquence de la figure 3.27 pour affecter à *a2* la valeur manquante, il suffirait d'ajouter *a2 value: 'Rome'*.

3.5.2 - Un tableau pour enregistrer des associations

Examinons la séquence de la figure 3.28. L'évaluation de la séquence qu'elle présente peut être décomposée en quatre étapes. La première crée un tableau de cinq éléments, appelé *repertoire*, et y enregistre des associations pays ⇨ capitale pour cinq pays méditerranéens.

La seconde étape demande à l'utilisateur de donner le nom d'un pays. Le nom frappé par l'utilisateur deviendra la valeur de la variable *paysChoisi*. Ce bref dialogue avec l'utilisateur est effectué à l'aide d'un objet de la classe *Prompter* qui est créé par la méthode de classe *prompt:default:* de cette classe. Si *v* est une variable, pour évaluer le message

v `:= Prompter prompt:` *question* `default:` *reponseParDefaut*

Smalltalk effectue les opérations suivantes :

- Une fenêtre (objet de la classe *Prompter*) est ouverte à l'écran. Sa barre de titre indique le texte donné avec l'argument *question*. Son panneau de texte propose *reponseParDefaut*.
- L'utilisateur peut alors modifier à sa guise le texte proposé puis valider la réponse qu'il choisit, en appuyant sur la touche « Entrée ».
- La fenêtre « *Prompter* » disparaît de l'écran (l'instance correspondante est détruite) et la réponse fournie par l'utilisateur est renvoyée par la fin du traitement du message. Cette réponse devient la valeur de la variable *v*.

```
|repertoire paysChoisi index|
"on enregistre, dans repertoire, les capitales de cinq pays méditerranéens"
repertoire := Array new: 5.
repertoire
  at:1 put: (Association key: 'Grèce' value: 'Athènes').
repertoire
  at:2 put: (Association key: 'Egypte' value: 'Le Caire').
repertoire
  at:3 put: (Association key: 'Maroc' value: 'Rabat').
repertoire
  at:4 put: (Association key: 'Espagne' value: 'Madrid').
repertoire
  at:5 put: (Association key: 'Albanie' value: 'Tirana').
"maintenant, on demande à l'utilisateur d'indiquer un pays"
paysChoisi := Prompter prompt: 'indique un pays'
                       default: 'Grèce'.
"ici, on recherche le pays indiqué par paysChoisi dans repertoire"
index := 1.
[(index < 5) and: [paysChoisi ~= (repertoire at: index) key]]
   whileTrue: [index := index+1].
"ici, on répond, en fonction du résultat de la recherche"
(paysChoisi = (repertoire at: index) key)
  ifTrue:  [Transcript
               nextPutAll: paysChoisi, ' capitale : ',
                           (repertoire at: index) value;
               cr]
  ifFalse: [Transcript
               nextPutAll: paysChoisi,
                           ' : pays absent du répertoire';
               cr]
```

Figure 3.28 : utilisation d'un tableau d'associations

Après l'enregistrement, dans *paysChoisi*, du pays indiqué par l'utilisateur, la troisième étape explore le tableau *repertoire* pour rechercher la capitale correspondante. A l'issue de la recherche, la quatrième étape fournit la réponse à l'utilisateur, en fonction du résultat trouvé (pays présent ou non dans *repertoire*) au moyen d'un affichage dans la fenêtre *Transcript*.

On peut trouver plusieurs inconvénients au traitement proposé par l'exemple de la figure 3.28 :

- l'objet utilisé pour l'enregistrement des informations est un tableau et ce choix ne facilite pas la mise à jour, ni la croissance du volume d'informations. Comment supprimer une association, comment en ajouter une sixième ?
- pour traiter un large volume d'informations (chargées par exemple dans le tableau, à partir d'un fichier), il sera nécessaire de prévoir dès le départ un tableau de grandes dimensions et, plus le nombre

d'informations sera grand, plus la recherche d'une capitale sera longue. Cette recherche s'effectue en effet par une itération d'exploration du tableau.

En fait, pour un tel traitement, un objet de la classe *Dictionary* est bien mieux adapté, comme nous allons le constater au paragraphe suivant.

3.5.3 - Un dictionnaire pour enregistrer des associations

La figure 3.29 présente le même traitement que celui que nous avons étudié au paragraphe précédent, mais réalisé en utilisant un *dictionnaire*, objet de la classe *Dictionary*, pour la variable *repertoire*. Un objet de cette classe, sous-classe de la classe *Set*[25], est une collection d'associations clé ⇨ valeur, dont toutes les clés sont distinctes les unes des autres. Les associations sont rangées et retrouvées rapidement en utilisant une technique de hachage[26] dont l'utilisateur n'a pas à se préoccuper : elle fait partie des techniques de base utilisées par Smalltalk.

```
|repertoire paysChoisi|
"on enregistre, dans repertoire, les capitales de cinq pays méditerranéens"
repertoire := Dictionary new.
repertoire add: (Association key: 'Grèce' value: 'Athènes').
repertoire add: (Association key: 'Egypte' value: 'Le Caire').
repertoire add: (Association key: 'Maroc' value: 'Rabat').
repertoire add: (Association key: 'Espagne' value: 'Madrid').
repertoire add: (Association key: 'Albanie' value: 'Tirana').
"maintenant, on demande à l'utilisateur d'indiquer un pays"
paysChoisi := Prompter prompt: 'indique un pays'
                       default: 'Grèce'.
"ici, on recherche le pays indiqué par paysChoisi dans repertoire"
(repertoire includesKey: paysChoisi)
  ifTrue:  [Transcript
              nextPutAll: paysChoisi, ', capitale : ',
                          (repertoire at: paysChoisi);
              cr]
  ifFalse: [Transcript
              nextPutAll: paysChoisi,
                          ' : pays absent du répertoire';
              cr]
```

Figure 3.29 : exemple d'utilisation d'un dictionnaire

Par ailleurs, la taille d'un dictionnaire est variable : la méthode new permet de créer un dictionnaire comportant un nombre d'emplacements prédéfini (quatre par

25 La classe Set est étudiée au chapitre 4, dans le cadre duquel on revient également sur les particularités de la classe Dictionary.

26 Le hachage s'appuie sur une règle de calcul qui permet ici, par une simple transformation, d'obtenir, à partir d'une association, l'adresse de rangement de cette association (Cf chapitre 4, 4.9.2).

défaut, avec Smalltalk V) et chaque enregistrement d'une association dans le dictionnaire va comparer le nombre d'emplacements effectivement utilisés par rapport au nombre total d'emplacements. Pour maintenir les performances du hachage, un agrandissement sera automatiquement déclenché[27] dès que la proportion d'emplacements disponibles sera inférieure à un seuil prédéfini. L'utilisateur peut donc ajouter des informations à un dictionnaire, sans avoir à se soucier de la capacité du dictionnaire[28].

Si nous reprenons l'exemple de la figure 3.29, nous pouvons donc dire que le nombre d'associations pays ⇨ capitale qui peuvent être ajoutés au dictionnaire n'est plus limité à 5, comme il l'était avec un tableau dans l'exemple de la figure 3.28.

On remarquera que chaque nouvelle association est ajoutée au dictionnaire par la méthode *add:* dont l'argument doit être une association. On aurait pu utiliser la méthode *at:put:* pour écrire, par exemple, directement

```
repertoire  at:'Grèce'  put:'Athènes'
```

car la méthode *at:put:* telle qu'elle est définie pour la classe *Dictionary*, fait appel à la méthode *add:*. En effet, son protocole est le suivant :

```
at: aKey put: anObject
  "If the receiver dictionary contains the key/value pair whose key equals aKey,
   then replace the value of the pair with anObject, else add the aKey/anObject pair."
  self add: (Association key: aKey value: anObject).
  ^ anObject
```

Enfin, pour terminer la comparaison entre les deux techniques présentées dans les figures 3.28 et 3.29, remarquons que, si l'accès à l'information exige, avec un tableau, un parcours séquentiel d'autant plus long que le tableau est grand, ce même accès est immédiat avec un dictionnaire. En effet, cette opération est à la figure 3.29 représentée par l'évaluation de :

```
repertoire includesKey: paysChoisi
```

qui recherchera, par un hachage, la clé indiquée par *paysChoisi* et renverra *true* ou *false* selon le résultat de la recherche.

3.5.4 - Un exemple de dictionnaire analogique

Pour nous familiariser davantage avec les objets de la classe *Dictionary*, étudions maintenant un exemple, qui restera très élémentaire dans ce paragraphe, mais qui sera repris et enrichi au chapitre 6. Notre but est de construire une application permettant de gérer un dictionnaire analogique. Un tel dictionnaire peut être utilisé par le rédacteur d'un texte qui souhaite disposer d'un outil pour faciliter l'expression de ses idées. Ce dictionnaire permet d'obtenir, à partir d'un mot représentant un objet ou une idée, une liste de mots voisins ou apparentés. Par exemple, un tel dictionnaire fournira pour l'adjectif *fragile* les analogies suivantes

fragile : frêle, faible, grêle, mince, menu, branlant, précaire, inconsistant.

---

27 Cf chapitre 4, méthode grow, paragraphe 4.9.3.

28 Pour autant, bien sûr, que l'espace-mémoire géré dynamiquement par Smalltalk soit suffisant.

Nous souhaitons réaliser une application permettant à la fois de consulter et d'enrichir un dictionnaire analogique conservé de manière permanente dans un fichier et accessible à travers une interface graphique. L'application complète sera construite au chapitre 6. Nous allons ici nous contenter de définir les objets et les méthodes de base sur lesquels s'appuiera cette application.

La nature même du problème suggère de représenter la relation entre un mot *m* et la liste *l* des mots apparentés par une association dont la clé est *m* et la valeur est *l*. Dans une telle association, nous choisirons de représenter la liste *l* par une chaîne de caractères dans laquelle les mots associés à *m* apparaîtront séparés les uns des autres par un espace :

amour ⇨ *'passion amitié tendresse'*

Il paraît également bien adapté au problème posé, de rassembler ces associations dans un objet de la classe *Dictionary*, qui sera créé en début d'application et immédiatement initialisé à partir du fichier permanent de stockage du dictionnaire d'analogies. En fin d'application, cet objet, s'il a été modifié, enverra les informations qu'il contient dans ce fichier permanent.

Nous laisserons de côté, ici, la liaison avec le fichier (qui sera étudiée au chapitre 6) pour nous intéresser uniquement au nécessaire enrichissement de la classe *Dictionary* pour disposer de méthodes adaptées aux besoins de notre application, comme par exemple l'enregistrement d'une relation d'analogie entre deux mots, qui ne correspondra pas forcément à l'utilisation directe de la méthode *add:* de la classe *Dictionary*.

En fait plutôt que de modifier l'environnement de la classe *Dictionary*, nous allons créer une classe, que nous appellerons *DicoAnalogies* et qui sera reprise et enrichie au chapitre 6. Nous avons étudié, en 3.4, les techniques de création d'une classe. Indiquons ici simplement que nous définissons cette classe comme sous-classe de *Dictionary*, sans variables d'instances nommées :

```
Dictionary variableSubclass: #DicoAnalogies
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

Bien entendu, dès à présent, la classe *DicoAnalogies* dispose de l'ensemble des méthodes héritées de *Dictionary* et nous pouvons les utiliser, pour réaliser à travers une fenêtre d'inspection (Cf 3.1.3), une interface-utilisateur rudimentaire. Evaluons, par exemple, la séquence suivante avec *do it* :

```
|a1 a2 a3 dictionnaire|
a1 := Association key:'amour'
                  value:'passion amitié tendresse'.
a2 := Association key:'colère'
                  value:'rage fureur'.
a3 := Association key:'oubli'
                  value:'étourderie distraction omission'.
dictionnaire := DicoAnalogies with: a1 with: a2 with: a3.
dictionnaire inspect
```

La séquence précédente enregistre trois relations d'analogie, pour les mots *amour*, *colère* et *oubli*, respectivement dans les trois variables temporaires *a1*, *a2* et *a3*. Puis elle utilise la méthode *with:with:with:*[29] pour créer, dans la variable *dictionnaire*, un dictionnaire analogique qui répertorie les trois relations. Enfin, la dernière expression de la séquence envoie le message *inspect* au dictionnaire. Comme nous l'avons vu en 3.1.3, cette méthode, définie dans la classe Object, est disponible pour toutes les classes et permet d'examiner l'objet qui reçoit le message correspondant. Signalons que la classe *Dictionary* (et donc également *DicoAnalogies*) redéfinit cette méthode pour l'adapter à l'examen d'un dictionnaire. La figure 3.30 montre le résultat de l'évaluation de l'expression *dictionnaire inspect* et va nous permettre de préciser les caractéristiques de cette redéfinition.

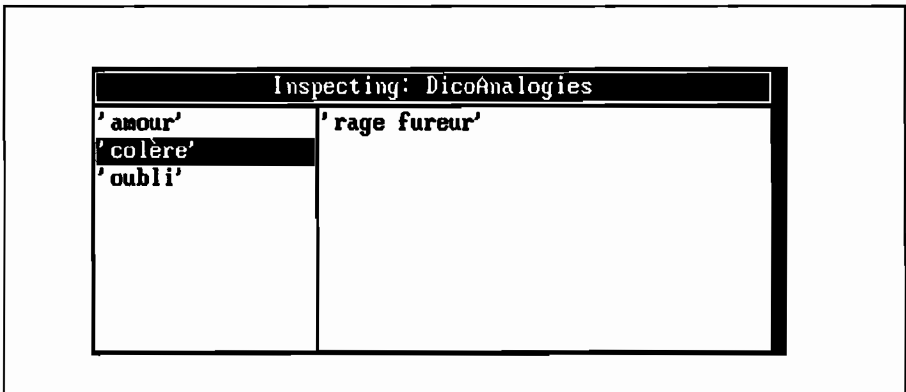


Figure 3.30 : inspection d'un dictionnaire

Comme les fenêtres d'inspection habituelles, cette fenêtre est divisée en deux panneaux. Cependant, le panneau de gauche n'indique pas la liste des variables d'instances mais la liste des clés des associations enregistrées dans le dictionnaire. Dans ce panneau, la clé *'colère'* est sélectionnée : la valeur correspondante (c'est-à-dire la chaîne formée par les mots apparentés) est affichée dans le panneau de droite. D'une manière générale, cette fenêtre d'inspection est bien adaptée à la consultation et même à la mise à jour d'un dictionnaire. En sélectionnant, par un simple clic de la souris, une clé du panneau de gauche, on fait immédiatement apparaître, en regard, dans la fenêtre de droite, la valeur correspondante. Cette valeur peut être modifiée. Si, dans la fenêtre représentée à la figure 3.30, on clique dans le panneau de droite, entre *rage* et *fureur*, on fait apparaître le curseur-texte et l'on peut, par exemple frapper le mot *exaspération* qui s'insèrera entre les deux mots précédents. Pour enregistrer la modification de l'analogie dans le dictionnaire sur lequel est ouverte la fenêtre d'inspection, il suffit de cliquer ensuite avec le bouton droit de la souris pour faire apparaître le menu d'édition de texte, dans lequel on sélectionnera l'option *save* de ce menu.

29 Cette méthode de classe est héritée de la classe Collection et permet de créer une instance comprenant trois éléments.

Bien que la méthode *inspect* redéfinie pour la classe *Dictionary*, soit particulièrement adaptée aux caractéristiques des instances de cette classe, on peut cependant se demander comment il serait possible d'effectuer, sur une instance de cette classe, une inspection « standard » telle qu'elle est définie dans la classe *Object* et qui permettrait d'examiner les détails internes d'une telle instance et non plus seulement les informations qu'elle répertorie. Nous allons maintenant répondre à cette question.

3.5.5 - La variable super

Examinons la fenêtre de gauche, intitulée « Inspection propre à Dictionary », de la figure 3.31. Elle résulte, à peu de choses près[30], de l'évaluation de la séquence suivante :

```
|d|
d := Dictionary with: (Association key: 'Lille' value: 59).
d inspect.
```

Cette séquence crée, dans la variable *d*, un dictionnaire avec une seule association (*'Lille'* ⇨ *59*) puis fait appel à la méthode *inspect* de la classe *Dictionary* pour ouvrir une fenêtre d'examen de ce dictionnaire. Cette fenêtre montre bien l'unique association enregistrée.

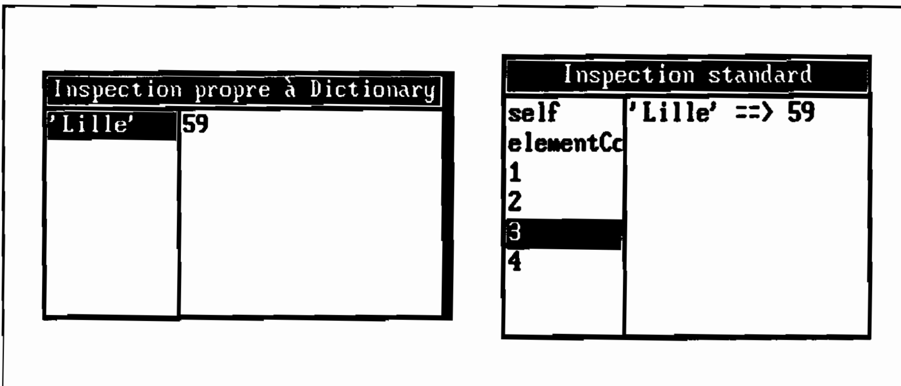


Figure 3.31 : deux fenêtres d'inspection sur un même objet

La fenêtre de droite de la même figure, intitulée « Inspection standard », est le résultat de l'évaluation de :

```
|d|
d := Dictionary with: (Association key: 'Lille' value: 59).
d inspectionStandard
```

30 A peu de choses près, car la barre de titre qui devrait indiquer « Inspecting: Dictionary », a été modifiée, après création de la fenêtre, pour la clarté de la présentation dans la figure. Cette modification est obtenue avec l'option « label » du menu de la barre de titre.

Cette séquence crée le même dictionnaire que la séquence précédente mais fait appel, pour son examen, à la méthode *inspectionStandard.* Comme nous allons le voir, cette méthode déclenche la méthode *inspect* de *Object* et non la méthode de même nom de *Dictionary.* La fenêtre montre alors le détail du dictionnaire[31] et nous pouvons constater que celui-ci possède une variable d'instance nommée, dont le nom *elementCount* n'est, compte tenu de la dimension de la fenêtre, que partiellement apparent. Cette variable indique le nombre d'associations enregistrées : si l'on cliquait sur cette variable dans le panneau de gauche, on verrait apparaître la valeur 1 dans le panneau de droite : il n'y a, dans le dictionnaire, que l'association *'Lille'* ⇨ *59.*

La fenêtre « Inspection standard » montre également que le dictionnaire possède des variables d'instance indexées numérotées de 1 à 4 (car la méthode *with:* crée un dictionnaire pouvant, au départ, contenir quatre associations[32]). Ces variables servent à répertorier les associations du dictionnaire et la fenêtre montre que l'unique association enregistrée est dans la troisième variable d'instance. On pourrait vérifier, en cliquant sur chacune des autres variables d'instances indexées, qu'elles désignent toutes l'objet *nil*, indiquant ainsi qu'elles restent inutilisées.

Expliquons maintenant le principe de fonctionnement de la méthode *inspectionStandard.* Précisons tout de suite qu'il ne s'agit pas d'une méthode prédéfinie en Smalltalk mais que c'est une méthode que nous avons ajoutée à la classe *Dictionary* avec le protocole suivant :

```
inspectionStandard
  "fait appel à la méthode inspect de Object"
  super inspect
```

On constate que cette méthode ne fait rien d'autre qu'envoyer à l'« objet » *super* le message *inspect.* En fait, *super* est une variable accessible dans toute méthode et qui représente toujours l'objet récepteur du message qui correspond à la méthode. La variable *super* joue donc un rôle tout à fait comparable à celui de *self*, mais avec une exception notable :

> Lorsque, dans une méthode de la classe *C* un message correspondant à une méthode d'instance *m* de la classe *C* est envoyé à la variable *super*, la méthode *m* de la classe *C* n'est pas utilisée. Smalltalk recherche, en remontant dans la hiérarchie des classes, dont *C* est l'héritière, la méthode *m* à exécuter.

Ainsi, dans la méthode *inspectionStandard*, la méthode *inspect*, qui est exécutée par l'évaluation de *super inspect*, n'est pas celle de Dictionary mais celle que Smalltalk trouvera, dans la classe *Object*, en remontant la hiérarchie : *Dictionary* → *Set* → *Collection* → *Object.* En effet, la méthode *inspect* n'est définie que dans les classes *Dictionary* et *Object.*

Remarquons que, avec l'utilisation de *super*, dans l'exploration ascendante de la hiérarchie, Smalltalk s'arrête dès qu'il trouve une méthode qui convient : si

---

31 On reviendra au chapitre 4, en 4.10, sur la structure interne d'une instance de la classe Dictionary.

32 Ce dictionnaire pourra s'agrandir (Cf 4.10.1) au fur à mesure qu'on lui ajoutera de nouvelles associations.

plusieurs méthodes peuvent convenir, il choisira donc celle de la classe la plus proche de la classe de départ. Par exemple, supposons que nous n'ayons pas défini la méthode *inspectionStandard* dans la classe *Dictionary*, mais que nous l'ayons ajoutée à la classe *DicoAnalogies* avec le même protocole :

```
inspectionStandard
  "méthode définie dans la classe DicoAnalogies"
  super inspect
```

Alors l'évaluation de *DicoAnalogies new inspectionStandard*, entraînera l'apparition du même type de fenêtre que « Inspection propre à Dictionary » (Cf figure 3.31) car Smalltalk, dans la recherche d'une méthode *inspect* au dessus de la classe DicoAnalogies, s'arrêtera à la classe Dictionary, qui contient cette méthode.

Insistons enfin sur un dernier détail du mécanisme associé à l'utilisation de *super* et supposons :

- que la méthode *inspectionStandard* est définie par *super inspect*, dans la classe *Dictionary*,
- que cette même méthode *inspectionStandard* n'est pas redéfinie dans la sous-classe *DicoAnalogies*.

Alors, l'évaluation de *DicoAnalogies new inspectionStandard* fera bien appel à la méthode *inspect* de *Object* car Smalltalk qui, dans cette évaluation, travaille sur une instance de *DicoAnalogies*, commencera la recherche de la méthode *inspect* non pas dans la classe-mère de *DicoAnalogies* mais dans la classe *Set*, mère de *Dictionary*. En effet, le message *inspectionStandard*, adressé à une instance de *DicoAnalogies*, fait appel à la méthode *inspectionStandard* que Smalltalk trouvera dans *Dictionary*. Quand Smalltalk évaluera, dans cette méthode, l'expression *super inspect*, il remontera à la classe-mère de *Dictionary* pour rechercher la méthode *inspect* qui convient. L'utilisation de *super* déclenche la recherche ascendante en commençant par la classe qui se trouve au-dessus de celle qui définit la méthode dans laquelle *super* est utilisée.

3.5.6 - Enrichir le dictionnaire des analogies

Revenons à la classe *DicoAnalogies* et ajoutons-lui les moyens d'enrichir une de ses instances en y enregistrant une nouvelle relation de parenté entre deux mots de sens voisins. Auparavant, pour nous permettre de vérifier, sur la fenêtre *System Transcript*, les opérations effectuées sur un tel dictionnaire, ajoutons, à la classe *DicoAnalogies*, les méthodes d'instances *affichePour: unMot* et *afficheTout* qui permettent de faire apparaître sur la fenêtre *System Transcript*, les analogies correspondant à *unMot* ou l'ensemble des analogies enregistrées. La réalisation de ces méthodes ne présentant pas de difficulté particulière, nous les présentons, sans autre commentaire, à la figure 3.32.

```
affichePour: unMot
  "affiche sur Transcript, s'il en existe, les analogies enregistrées pour unMot dans le
   dictionnaire des analogies récepteur"
 (self includesKey: unMot)
   ifTrue: [Transcript nextPutAll: 'analogies pour ', unMot;
                       cr;
                       nextPutAll: ' ', (self at: unMot);
                       cr]
   ifFalse: [Transcript nextPutAll: 'pas d''analogies pour ',
                                    unMot;
                        cr]

afficheTout
  "envoie sur Transcript toutes les analogies"
 Transcript nextPutAll: 'contenu complet du dictionnaire';
            cr.
 self associationsDo:
   [:couple "désignera chaque analogie enregistrée" |
    Transcript nextPutAll: 'analogies pour ', couple key;
               cr;
               nextPutAll: '  ', couple value; cr]
```

Figure 3.32 : deux méthodes pour l'affichage d'analogies

Nous pouvons maintenant nous intéresser à l'addition d'une analogie au dictionnaire. Il s'agit d'enregistrer le fait que deux mots sont de sens proches, par exemple *colère* et *exaspération*, *calme* et *tranquillité*. Bien entendu, la proximité de sens est une relation symétrique et il faudra que le dictionnaire enregistre aussi bien *calme* ⇨ *tranquillité* que *tranquillité* ⇨ *calme*. Cependant, dans un premier temps, nous allons considérer que la relation est dissymétrique et privilégier l'un des deux mots qui servira de clé d'accès au dictionnaire. Construisons donc la méthode

```
ajoute: motProche  pour: motCle
```

qui ajoutera aux analogies déjà enregistrées pour *motCle*, le mot *motProche*.

Deux cas sont à considérer :

1) Aucune analogie n'est encore enregistrée pour *motCle* dans le dictionnaire. Dans ce cas, il faut créer l'association correspondante.
2) Le dictionnaire contient déjà une association dont *motCle* est la clé. Dans ce cas, il faut compléter la valeur de cette association avec le nouveau mot.

La figure 3.33 décrit la méthode qui correspond à ces opérations. Les commentaires qui accompagnent le code sont suffisants pour que le lecteur puisse comprendre le protocole de cette méthode. Remarquons seulement que, lorsque *motCle* est déjà une clé du dictionnaire, l'opération qui détermine si *motProche* est déjà une analogie enregistrée pour *motCle* est, en Smalltalk, exprimée de manière parlante et concise par :

```
listeMotsProches asArrayOfSubstrings includes: motProche
```

Dans l'expression précédente, la variable *listeMotsProches* contient la chaîne de caractères formée des mots qui constituent des analogies déjà connues de *motCle*. Le lecteur qui a déjà une expérience de programmation pourra établir un parallèle avec des langages classiques dans lesquels la même opération se décrirait de manière bien moins aisée. Dans ces langages, le programmeur sera en général obligé de construire lui-même la primitive de transformation d'une chaîne en tableau de sous-chaînes ainsi que la primitive de recherche d'un mot dans un tableau de chaînes. Cet exemple met en évidence aussi bien le confort apporté par la programmation orientée objets, qui facilite la réutilisation de code, que la richesse de Smalltalk, dont la très large bibliothèque de classes fournit de puissantes primitives de traitement.

```
ajoute: motProche pour: motCle
  "ajoute l'analogie motCle ⇨ motProche dans le dictionnaire des analogies récepteur :
   crée l'association si motCle n'était pas répertorié,
   renvoie true en cas de succès et false si l'analogie était déjà enregistrée. "
   |listeMotsProches|
   (self includesKey: motCle)
      ifTrue: "motCle est déjà une clé enregistrée dans le dictionnaire"
              [listeMotsProches := self at: motCle.
                "listeMotsProches contient maintenant les analogies de motCle"
                "on va regarder si motProche est déjà une analogie de motCle"
                (listeMotsProches asArrayOfSubstrings
                                  includes: motProche)
                   ifTrue:  [^false "car on a trouvé motProche"]
                   ifFalse: [self at: motCle
                                  put: listeMotsProches,
                                       ' ', motProche.
                            "on vient d'ajouter motProche à la liste des
                            analogies déjà existantes de motCle"
                            ^ true]
              ]
      ifFalse: "motCle n'est pas une clé déjà enregistrée dans le dictionnaire"
               [self add: (Association key: motCle
                                       value: motProche).
                ^ true]
```

Figure 3.33 : enregistrement dissymétrique d'une analogie

Pour contrôler le fonctionnement de la méthode *ajoute:Pour:*, créons un dictionnaire d'analogies vide au départ et ajoutons-lui quelques analogies avant d'utiliser la méthode *afficheTout* pour présenter le contenu du dictionnaire sur la fenêtre *System Transcript*. La figure 3.34 montre le résultat obtenu. La séquence évaluée apparaît dans la fenêtre en arrière-plan. Le résultat de l'évaluation est affiché dans *System Transcript*.

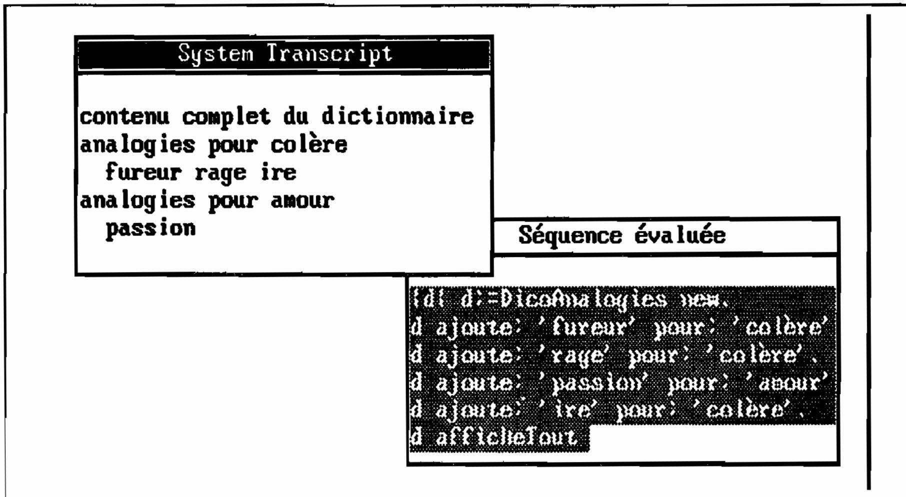


Figure 3.34 : utilisation de la méthode *ajoute:Pour:*

Nous pouvons maintenant définir la méthode qui permettra d'enregistrer une relation d'analogie de manière symétrique. Cette méthode s'écrira simplement en faisant deux appels à la méthode *ajoute:Pour:* et nous pouvons en donner le description suivante :

```
ajouteAnalogieEntre: mot1 et: mot2
  "ajoute, si elles n'y sont pas déjà, les analogies mot1/mot2 et
   mot2/mot1 au dictionnaire récepteur,
   renvoie false si l'une des deux analogies était déjà présente, sinon renvoie true"
  |reponse "indiquera si tout s'est bien passé" |
  reponse := self ajoute: mot1 pour: mot2.
  ^ (self ajoute: mot2 pour: mot1) & reponse
```

Cette méthode étant ajoutée à la classe *DicoAnalogies*, on peut désormais considérer la méthode *ajoute:pour:* comme une méthode privée, réservée à la seule utilisation de la méthode *ajouteAnalogieEntre:et:*, de manière à garantir, avec l'emploi exclusif de cette dernière méthode pour l'ajout de relation d'analogie, la symétrie d'un dictionnaire d'analogies.

3.5.7 - Le dictionnaire de Smalltalk

Smalltalk répertorie les variables globales (Cf 2.9.1.), les dictionnaires partagés (Cf 2.9.2.) ainsi que les classes, dans un dictionnaire, de nom *Smalltalk*. Ce dictionnaire est la seule instance de la classe *SystemDictionary* et c'est une instance permanente. Le dictionnaire *Smalltalk* contient, comme tout dictionnaire, des associations, c'est-à-dire des couples clé/valeur.

La clé est toujours le symbole qui correspond au nom de l'objet répertorié (nom de la variable globale, nom du dictionnaire partagé, nom de la classe). On notera, à ce propos, que la classe *SystemDictionary* redéfinit les méthodes *add:* et *at:put:* pour vérifier que la clé de toute nouvelle association est bien un symbole.

Quand une association répertoriée dans le dictionnaire *Smalltalk* désigne une variable globale, la valeur de cette association est la valeur de la variable. Par exemple, si l'on a affecté à la variable globale *Pays* la chaîne *'France'*, l'évaluation, avec *show it*, de *Pays* renverra *'France'* et l'évaluation de

```
Smalltalk  at: #Pays
```

renverra la même valeur. On notera que, dans l'expression précédente, l'argument du message *at:* est le symbole *#Pays*. En effet, l'évaluation de *Smalltalk at: Pays* renverra une fenêtre de diagnostic indiquant l'erreur *Key is missing*, car Smalltalk rechercherait, dans le dictionnaire *Smalltalk*, la valeur de la variable *Pays* et non le symbole *#Pays*. En revanche, si *France* est une autre variable globale, l'évaluation, avec *show it*, de :

```
Pays := #France.
France := 'pays de la CEE'.
Smalltalk at: Pays
```

renverra la valeur *'pays de la CEE'*.

Quand une association répertoriée dans le dictionnaire *Smalltalk* désigne un dictionnaire partagé, la valeur de cette association est le dictionnaire lui-même. Ainsi, l'évaluation de

```
(Smalltalk at: #CharacterConstants) inspect
```

ouvrira une fenêtre d'examen sur le dictionnaire partagé *CharacterConstants* (Cf 2.9.2. pour des informations sur ce dictionnaire).

Quand une association répertoriée dans le dictionnaire *Smalltalk* désigne une classe, la valeur de cette association est la classe elle-même[33]. On peut vérifier que c'est bien la classe qui est accessible par l'intermédiaire de la valeur de l'association en évaluant par exemple, avec *show it*, l'expression :

```
(Smalltalk at: #Array) with: 'Paris' with: 'France'
```

on obtient le même tableau (*'Paris' 'France'*), que celui que l'on aurait obtenu, en évaluant *Array with: 'Paris' with: 'France'*.

On ne peut supprimer une classe en adressant un message au dictionnaire *Smalltalk*. Nous examinerons, au chapitre 7, les opérations nécessaires à la suppression d'une classe définie par l'utilisateur[34]. On peut en revanche supprimer une variable globale définie par l'utilisateur, en utilisant la méthode *removeKey*, héritée de Dictionary. Ainsi, pour supprimer la variable globale *Pays*, il suffira d'évaluer l'expression *Smalltalk removeKey: #Pays*.

Le dictionnaire *Smalltalk* est bien entendu utilisé par Smalltalk lui-même pour de très nombreuses opérations internes. L'utilisateur peut lui aussi le consulter et en extraire des informations. A titre d'exemple, la séquence suivante permet, quand elle est évaluée, d'obtenir, sur la fenêtre *System Transcript*, la liste des noms de variables globales et des dictionnaires partagés.

---

33 En fait, c'est un pointeur sur la description de la classe qui est stocké dans le dictionnaire Smalltalk. On se reportera au chapitre 7 pour davantage d'explications sur la représentation des classes.

34 Les classes de bases, prédéfinies, ne peuvent être supprimées car elles font partie de l'environnement de Smalltalk.

```
Smalltalk associationsDo:
  [:element | "désignera chaque association enregistrée dans le dictionnaire"
   (element value isKindOf: Class)
      ifFalse: [element key printOn: Transcript.
               Transcript cr.]
  ]
```

Cette séquence évalue répétitivement (méthode *associationsDo:*) un bloc, pour chaque association répertoriée dans le dictionnaire *Smalltalk*. Dans ce bloc, chaque association sera désignée par l'argument *element*. La valeur de chaque association (*element value*) reçoit le message *isKindOf: Class* et renverra la valeur *true* ou *false* selon que cette valeur représente ou non une classe. Chaque fois que la valeur renvoyée est *false*, l'objet répertorié n'est pas une classe et ne peut être qu'une variable globale ou un dictionnaire partagé : on affiche alors la clé de l'association (c'est-à-dire le nom de l'objet représenté) dans la fenêtre *System Transcript*.

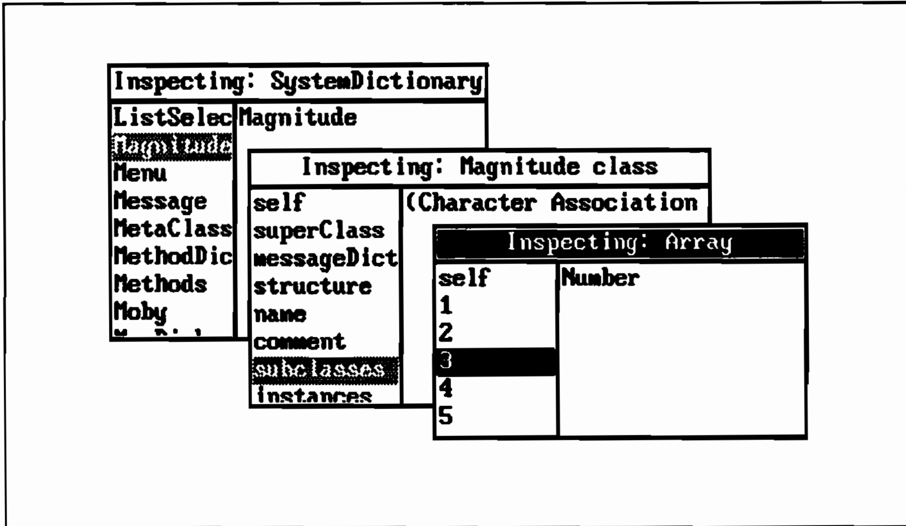


Figure 3.35 : trois inspections successives

Pour terminer, examinons la figure 3.35 qui montre comment on peut également utiliser le dictionnaire Smalltalk pour examiner la structure interne d'un objet. Cette figure a été obtenue par la succession d'opérations suivantes :

- On a déclenché l'évaluation de *Smalltalk inspect* qui a fait apparaître la fenêtre d'inspection figurant en arrière-plan et dont la barre de titre est : « Inspecting SystemDictionary ».
- Dans le panneau de gauche, on a fait défiler la liste des symboles jusqu'à sélectionner la classe *Magnitude*.
- On a alors cliqué dans le même panneau avec le bouton droit de la souris pour faire apparaître le menu correspondant ; dans ce menu, on a sélectionné l'option *inspect* qui a déclenché l'ouverture d'une nouvelle fenêtre d'inspection sur la sélection (*Magnitude*).

- Cette deuxième fenêtre est la fenêtre « Inspecting Magnitude Class ». Elle donne, dans son panneau de gauche, la liste des variables d'instances de l'objet qui représente la classe. Dans cette liste, la variable *subclasses* est sélectionnée, et l'on voit apparaître, en regard, dans le panneau de droite, le début d'un tableau qui répertorie les sous-classes de *Magnitude*.
- On a enfin, dans le panneau de gauche de cette deuxième fenêtre, activé l'option *inspect* du menu, qui a déclenché l'ouverture de la troisième fenêtre « Inspecting: Array ». Cette dernière fenêtre inspecte la variable *subclasses*. Dans le panneau gauche de cette dernière fenêtre, la troisième variable d'instance indexée est sélectionnée : elle désigne la sous-classe *Number* de *Magnitude*.

4 - La classe Collection et sa descendance

4.1 - Collection et ses sous-classes

```
'Chaîne',
'Ligne contenant des mots',
(0 F F F C 4 3 D 5 A),
('poulet' 'vermicelles' 'raton laveur' 'ordinateur'),
('fichier.ext'   12345    (Nov 14, 1990)  14:31:09),
#(4 5 6 7)...
```

A l'instar de Prévert, Smalltalk V regroupe toutes ces énumérations sous la même classe mère. En effet, qu'il s'agisse de nombres, d'octets, de mots, de lettres ou de date, les mêmes traitements sont nécessaires : ajouter un élément à la liste, en enlever un, tester la présence de tel ou tel, réunir deux listes, les vider etc.

C'est le rôle de la classe *Collection* de regrouper tous les types d'énumérations[1] ou de listes imaginables et de leur fournir quelques méthodes de base.

Il est bien évident que certaines énumérations appellent des traitements spécifiques. Une suite de mots nécessitera des opérations différentes de celles d'un tableau de chiffres ou d'octets. La classe *Collection* est donc divisée en plusieurs sous-classes, selon une hiérarchie basée sur :

- l'accès direct à un élément de la collection (par une clé ou un indice) ;
- l'ordre imposé (ou non) aux éléments de la collection ;
- la possibilité de redondances dans la collection ;
- la nature des éléments stockés.

La figure 4.1 donne le détail de cette hiérarchie. La première sous-classe, *Bag*, est une sorte de fourre-tout dans laquelle les éléments n'ont aucun type imposé (nombre, caractère, symbole, tableau, etc.) et peuvent être dupliqués autant de

---

1 Par la suite, nous parlerons indifféremment de « collection » ou d'énumération pour désigner les instances de la classe Collection, et d'éléments pour désigner les objets qui composent ces énumérations.

fois que nécessaire. Nous verrons un peu plus loin quelques méthodes utilisables sur les instances de cette classe.

La seconde, *IndexedCollection*, regroupe toutes les collections dont chaque élément peut être repéré par sa position. Contrairement à *Bag*, la classe *IndexedCollection* est une classe abstraite : il ne peut en exister aucune instance. C'est au niveau de ses sous-classes que les instances peuvent être créées. Une classe abstraite regroupe un certain nombre de méthodes utilisables par plusieurs classes apparentées. Mais ces méthodes sont insuffisantes pour qu'une instance puisse être directement créée, ou manipulée.

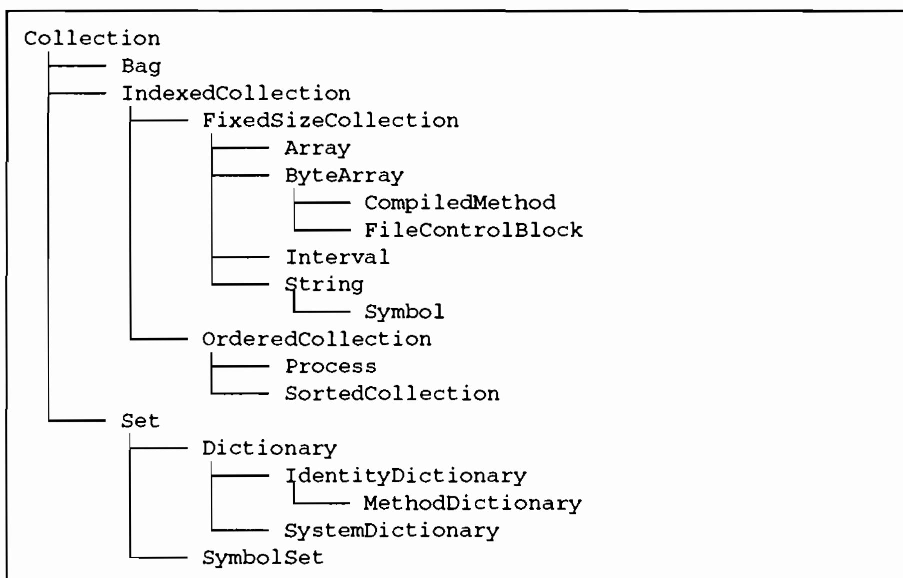


Figure 4.1 : la classe *Collection* et sa descendance

La troisième, *Set*, contient des collections dont les éléments sont uniques (alors qu'ils peuvent être présents en plusieurs exemplaires dans les instances de *Bag*). Contrairement à *IndexedCollection*, on ne peut accéder aux éléments d'une instance par leur position (les éléments d'une instance de *Set* ne sont pas ordonnés). Certaines des sous-collections de *Set* (*Dictionary*) permettent toutefois d'accéder à leurs éléments par une clé (un index plus exactement). *Set* n'est pas une classe abstraite.

Nous expliquerons dans la première partie (4.2 à 4.6) comment utiliser les différentes sous-classes de *Collection*, à l'aide d'exemples. Nous aborderons en deuxième partie (4.7 à 4.10) le contenu des méthodes, et la manière dont les concepteurs de Smalltalk V ont implémenté cette classe et sa descendance.

4.2 - La classe Bag : aperçu de quelques méthodes

4.2.1 - Créer une instance de la classe Bag

Prenons l'exemple d'une ménagère s'approvisionnant au marché. Nous simulerons son panier à provisions avec une instance de la classe *Bag*, ce qui lui permettra d'ajouter autant d'exemplaires que nécessaire d'un même objet.

Définissons donc *panierDeCourse* comme une instance de la classe *Bag*, en envoyant le message *new*.

```
|panierDeCourse|
panierDeCourse := Bag new
```

et vérifions qu'il est vide avec le message *size* :

```
panierDeCourse size
```

L'évaluation par l'option *show it* donne *0*.

4.2.2 - Remplir une instance de la classe Bag

Il reste à remplir notre panier, avec le message *add: unObjet*, en ajoutant à notre évaluation précédente les lignes :

```
panierDeCourse add: 'chou'.
panierDeCourse add: 'salade'.
panierDeCourse add: 'gros sel'.
panierDeCourse size
```

Le dernier message nous permet de vérifier que notre panier contient maintenant trois éléments.

Si notre ménagère décide soudain de racheter d'autres salades, rien de plus simple :

```
panierDeCourse add:'salade'.
```

Nous pouvons vérifier avec le message *size* que *panierDeCourse* contient un élément de plus. Les deux occurrences de *'salade'* apparaissent bien comme deux objets distincts dans l'instance de *Bag*.

Si les tomates particulièrement bon marché tentent notre ménagère, elle pourra en acheter une douzaine avec le message :

```
panierDeCourse add: 'tomate' withOccurrences:12
```

4.2.3 - Explorer le contenu d'une instance de la classe Bag

Et pour afficher le contenu de notre panier ?

Il se trouve que la classe *Bag* n'a pas de méthode naturelle pour un tel affichage, contrairement à la plupart des autres sous-classes de *Collection.* Nous allons créer *panierDeCourse* comme illustré figure 4.2.

```
|panierDeCourse|
 panierDeCourse := Bag new.
 panierDeCourse add:'salade'.
 panierDeCourse add:'poireau'.
 panierDeCourse add:'salade'
 panierDeCourse add:'radis' withOccurrences:4.
```

Figure 4.2 : création de *panierDeCourse*

Explorons les méthodes d'instance de *Bag* qui pourraient nous aider à détailler le contenu de notre panier.

Avec *occurrencesOf: unObjet*, nous pouvons interroger le récepteur pour qu'il nous renvoie le nombre de *unObjet* qu'il contient. Ajoutons par exemple l'instruction suivante à la séquence de la figure 4.2 avant d'évaluer le tout avec *show it* :

```
panierDeCourse occurrencesOf: 'salade'
```

Nous obtenons *2*.

Complétons ensuite la figure 4.2 avec la ligne suivante, avant d'évaluer le tout avec *show it* :

```
panierDeCourse includes:'radis'
```

La réponse est *true*. L'objet *'radis'* fait bien partie de *panierDeCourse.*

La méthode *includes: unObjet* (héritée de la classe *Collection*) nous permet donc de tester la présence de *unObjet* dans notre sac. Mais elle ne nous donne pas le nombre d'occurrences de cet objet.

On le voit, il n'est pas très pratique d'examiner le contenu de notre *panierDeCourse.* Une dernière méthode, *do: unBloc*, pourrait nous aider. Cette méthode exécute le bloc *unBloc* pour chacun des éléments de l'objet recevant le message. Dans notre cas, nous pourrions demander l'affichage à l'écran des éléments de *panierDeCourse.*

```
panierDeCourse do:[:legume| legume printOn: Transcript.
                            Transcript cr]
```

Le bloc d'instructions est évalué pour chacun des éléments de l'objet *panierDeCourse.* Il a pour effet d'afficher sur la fenêtre *System Transcript* tous les légumes de *panierDeCourse*, tout en revenant à la ligne après chaque affichage d'un légume.

Supprimer des éléments d'une instance de la classe Bag

Une fois rentrée à la maison, la cuisinière se précipite sur notre *panierDeCourse* et en retire les ingrédients. Pour le mettre à jour, la méthode *remove:ifAbsent:* est toute indiquée. La figure 4.3 montre la suppression du premier radis. On remarquera que nous avons légèrement amélioré le rendu de notre exemple en obtenant l'affichage de la taille avant le message *remove: ifAbsent:*

```
|panierDeCourse|
 panierDeCourse := Bag new.
 panierDeCourse add:'salade'.
 panierDeCourse add:'poireau'.
 panierDeCourse add:'salade'
 panierDeCourse add:'radis' withOccurrences:4.
 'Mon panier contient' printOn:Transcript.
 panierDeCourse size   printOn:Transcript.
 '   legumes '         printOn:Transcript.
 panierDeCourse remove:'radis'
             ifAbsent:[self error:
               'pas de radis dans le  panier'].
 panierDeCourse size printOn:Transcript.
```

Figure 4.3 : enlever un élément de *panierDeCourse*

Si nous tentons maintenant d'enlever un légume qui n'existe pas (ou n'existe plus) dans *panierDeCourse*, le bloc paramètre de *ifAbsent:* est évalué.

```
System Transcript
'Mon panier contient '7'  légumes '

pas d'asperge dans le panier
[] in UndefinedObject>>DoIt

Découvrir Sm
           |panierDeCourse|
panierDeCourse := Bag new.
panierDeCourse add:'salade'.
panierDeCourse add:'poireau'.
panierDeCourse add:'salade'.
panierDeCourse add:'radis' withOccurrences:4.
'Mon panier contient ' printOn:Transcript.
panierDeCourse size printOn:Transcript.
'  légumes ' printOn:Transcript.
panierDeCourse remove:'asperge'
  ifAbsent:[self error:'pas d''asperge dans le panier'].
```

Figure 4.4 : erreur lors d'une suppression d'objet dans *panierDeCourse*

Une fenêtre d'erreur s'affiche, dont le titre est la chaîne que nous avons donnée en paramètre du message *error: uneChaine.* C'est le cas illustré figure 4.4, où nous avons tenté d'enlever *'asperge'* au lieu de *'radis'*.

4.3 - Les classes Set et Dictionary

4.3.1 - Des méthodes de conversion présentes dans Collection

Pour extraire de *panierDeCourse* les différents types de légume que nous y avons mis, sans nous préoccuper de leur nombre, il suffit de transposer les éléments constituants *panierDeCourse* dans une instance de la classe *Set*, puisque une instance de la classe *Set* ne peut contenir plusieurs exemplaires d'un même objet. Il existe une méthode, définie au niveau de *Collection*, qui permet un tel transfert : c'est la méthode *asSet.*

La figure 4.5 donne un exemple de conversion d'une instance de *Bag* en instance de *Set.*

```
|panierDeCourse typeDeLegume|
 panierDeCourse := Bag new.
 panierDeCourse add:'patate'.
 panierDeCourse add:'poireau'.
 panierDeCourse add:'patate'.
 panierDeCourse add: 'radis' withOccurrences:4.
 panierDeCourse size printOn:Transcript.
 typeDeLegume := panierDeCourse asSet.
 typeDeLegume size printOn:Transcript
```

Figure 4.5 : passage d'une instance de *Bag* à une instance de *Set*

L'évaluation de la séquence de la figure 4.5 provoque l'affichage dans la fenêtre *Transcript* de la taille de *panierDeCourse* (7) puis de celle de l'instance de *Set* qui en est tirée (3) : tous les doublons ont été éliminés lors du transfert des éléments constituant *panierDeCourse* dans *typeDeLegume.*

D'autres méthodes permettent de transférer des instances d'une sous-classe de *Collection* en instance de *Bag* (*asBag*), de *Array* (*asArray*), de *SortedCollection* (*asSortedCollection*), de *IndexedCollection* (*asIndexedCollection*) etc.

Nous aurions aussi pu créer directement *typeDeLegume* avec la méthode *with: with: with:*, héritée de *Collection* qui crée et charge immédiatement une instance de la classe réceptrice avec les objets donnés en paramètres. Des méthodes comparables existent pour mettre un, deux ou quatre objets dans l'instance.

```
|typeDeLegume|
typeDeLegume := Set with:'patate'
                    with:'poireau'
                    with:'radis'.
```

4.3.2 - Une sous-classe importante de Set : Dictionary

Nous avons utilisé cette classe au chapitre 3 (Cf 3.5). Nous allons décrire ses méthodes de manière plus exhaustive dans ce paragraphe.

Construite sur le principe des dictionnaires, cette classe permet de stocker des associations. Elle est donc particulièrement utile dès que l'on veut retrouver un objet par la clé qui lui est associée. Les problèmes suivants peuvent tous être résolus en utilisant les possibilités de cette classe :

- rechercher la traduction anglaise d'un mot français,
- trouver le prix d'un article de supermarché,
- afficher le code constructeur d'une pièce de rechange,
- retrouver un individu par son numéro de Sécurité Sociale,

etc.

Comme n'importe quelle instance de *Set*, un dictionnaire ne doit contenir qu'un exemplaire de chaque association. La règle est en fait plus rigoureuse, pour une meilleure utilisation de cette classe, car un dictionnaire ne doit pas contenir plus d'un exemplaire de chaque clé. Ainsi, (*Charles* ⇨ *De Gaulle*) et (*Charles* ⇨ *Péguy*), s'ils cohabitent dans l'histoire, ne peuvent être contenus dans le même dictionnaire, car ils ont la même clé (*Charles*).

4.3.3 - Les méthodes de Dictionary classées par fonctions

Le stockage des éléments d'un dictionnaire étant basé sur la clé de l'association et non sur l'association elle-même, nombreuses sont les méthodes d'accès qui sont redéfinies dans la classe *Dictionary*.

Nous allons les parcourir rapidement, avant d'en utiliser quelques unes pour résoudre un problème classique de gestion de personnel.

Ajouter un objet au dictionnaire

add: anAssociation ajoute l'association *anAssociation* au dictionnaire.

at: key put: value ajoute l'association (*key* ⇨ *value*) au dictionnaire.

Tester la présence d'un objet

includes: value répond vrai si *value* correspond à une clé (quelle qu'elle soit) existant dans le dictionnaire, faux sinon.

includesKey: aKey répond vrai si *aKey* fait partie des clés du dictionnaire, faux sinon.

includesAssociation: anAssociation
répond vrai si l'association *anAssociation* fait partie du dictionnaire, faux sinon.

Ces trois méthodes permettent donc de tester la présence d'une association, de sa clé ou de sa valeur dans un dictionnaire.

Récupérer un objet

à partir de sa clé :

at: aKey renvoie la valeur associée à la clé *aKey*

at: aKey ifAbsent: aBlock
renvoie la valeur associée à *aKey*, exécute le bloc *aBlock* si la clé n'existe pas dans le dictionnaire.

associationAt: aKey
comme *at:*, mais renvoie l'association.

associationAt: aKey ifAbsent: aBlock
comme *at:ifAbsent:*, mais renvoie l'association complète.

à partir de la valeur :

keyAtValue: value
renvoie la première clé associée à la valeur *value*

keyAtValue: value ifAbsent: aBlock
comme le précédent, mais exécute *aBlock* si *value* n'existe pas dans le dictionnaire.

Ces deux dernières méthodes sont délicates à manipuler, car elles ne renvoient que la première clé correspondant à l'argument. Or nous avons vu que si les clés sont uniques pour un dictionnaire, ce n'est pas le cas des valeurs qui leur sont associées. Ces deux méthodes peuvent donc donner des résultats inattendus si l'on oublie cette propriété.

Il existe des méthodes plus exhaustives :

keys renvoie une instance de *set* contenant toutes les clés

values renvoie une instance de *Bag* contenant les valeurs correspondant aux clés du dictionnaire

Nous revoyons la particularité des dictionnaires dans la différence d'implémentation des dernières méthodes : *keys* peut renvoyer une instance de *Set* car les clés sont uniques, alors qu'il faut une instance de *Bag* pour les valeurs qui peuvent être dupliquées.

Pour savoir combien de clés différentes mènent à la même valeur, on pourra utiliser la méthode *occurrencesOf: value.*

Traiter tous les objets d'un dictionnaire

La méthode *do:* de la classe *Set* présente trois variations dans la classe *Dictionary*, pour exploiter totalement la richesse de cette classe.

associationDo: aBlock
évalue *aBlock* en lui fournissant chaque association en paramètre (c'est donc l'équivalent de la méthode *do:* de la classe *Set*).

keysDo: aBlock
évalue *aBlock* en lui fournissant la clé de chaque association en paramètre.

do: aBlock évalue *aBlock* en lui fournissant la valeur de chaque association en paramètre alors que la méthode *associationDo:* donne l'association (clé et valeur) en paramètre du bloc.

Enlever un objet

removeKey: aKey enlève l'association de clé *aKey* du dictionnaire.

removeKey: aKey ifAbsent: aBlock
comme la précédente, mais exécute *aBlock* si la clé *aKey* est absente du dictionnaire.

removeAssociation: anAssociation
enlève l'association *anAssociation* du dictionnaire[2]. Cette méthode fait appel à *removeKey*.

4.3.4 - Un exemple d'utilisation de dictionnaire : la paye du personnel

Monsieur Leriche, nouveau chef du personnel de la huitième chaîne de télévision, veut travailler sur une liste du personnel classée par catégories socio-professionnelles. Le montant du salaire risquant de suivre les résultats de sa chaîne, il veut aussi pouvoir modifier facilement le salaire correspondant à chaque catégorie socio-professionnelle.

Soit *personnel* le dictionnaire stockant pour chaque individu sa catégorie, et *indice* celui dans lequel Monsieur Leriche indiquera la correspondance entre la catégorie et le salaire.

Monsieur Leriche évalue le contenu de la figure 4.6. La chaîne est récente, et Monsieur Leriche n'a pas encore de secrétaire ni de balayeur. Outre le fait qu'il espère en recruter bientôt, rajouter ces deux catégories nous permettra de présenter un éventail plus large de méthodes utilisables dans *Dictionary*.

Pour obtenir le nombre de catégories réellement présentes, il ajoute au chargement des dictionnaires la ligne suivante avant d'évaluer tout le paragraphe avec *show it.*

```
personnel values asSet size.
```

La méthode *values* appliquée à *personnel* rend une instance de *Bag* contenant les catégories professionnelles, *asSet* en élimine les duplicata et *size* donne le résultat attendu.

---

2 Il n'existe pas de méthode permettant de supprimer toutes les associations correspondant à une valeur donnée. La méthode remove:ifAbsent:, de libellé trop imprécis, est d'ailleurs redéfinie dans la classe Dictionary de manière à signaler qu'elle est invalide dans cette classe.

```
|personnel indice|
personnel := Dictionary new.
personnel add:(Association key:'François'
                           value:'Directeur);
          at:'Michel' put:'Adjoint';
          at:'Philippe' put:'Présentateur';
          at:'Jack' put:'Animateur';
          at:'André' put:'Présentateur';
          at:'René' put:'Cameraman';
          at:'Paul' put:'Cameraman'.
indice := Dictionary new.
indice at:'Directeur' put:20000;
       at:'Adjoint' put:10000;
       at:'Présentateur' put:180000;
       at:'Animateur' put:360000;
       at:'Cameraman' put:5000;
       at:'Secrétaire' put:7000.
       at:'Balayeur' put:4000.
```

Figure 4.6 : chargement des dictionnaires *indice* et *personnel*

Les bas salaires disposant d'aides de l'état, Monsieur Leriche désire obtenir la liste des catégories concernées :

```
(indice select:[:salaireMensuel|salaireMensuel < 8000])
keysDo:[:categorie|categorie printOn:Transcript.
                   Transcript cr].
```

La méthode *select:* construit un nouveau dictionnaire en retenant toutes les associations du dictionnaire récepteur pour lesquelles le bloc argument renvoie *true*. L'évaluation du bloc argument se fait en utilisant l'association entière comme paramètre. Définie au niveau de la classe *Dictionary*, cette méthode est tout à fait adaptée à ce genre de problème. Comme elle renvoie un dictionnaire, nous appliquons la méthode *keysDo:* pour obtenir l'affichage sur la fenêtre *System Transcript* de chaque catégorie de personnel :

```
Cameraman
Secrétaire
Balayeur
```

Les feuilles de paye vont être stockées dans une nouvelle variable, *salaire*. Pour la créer, nous compléterons le contenu de la figure 4.6 avec les expressions suivantes :

```
|personnel indice salaire|
...... "définition et chargement du dictionnaire personnel"
...... "définition et chargement du dictionnaire indice"
salaire := Dictionary new.
personnel associationsDo:[:agent|
              salaire at:(agent key)
                     put:(indice at:(agent value))].
```

Nous avons utilisé la méthode *associationsDo:* (qui évalue le bloc avec l'association complète en paramètre) pour recréer un nouveau dictionnaire associant les noms (*agent key*) récupérés dans le dictionnaire *indice* à partir des catégories professionnelles (*indice at:(agent value)*) au salaire.

La huitième chaîne ne cesse de descendre dans les sondages. Monsieur Leriche doit se débarrasser des salaires les plus lourds. Pour cela, il évalue :

```
(salaire reject:[paye|paye>20000])
keysDo:[:nom|nom printOn:Transcript. Transcript cr]
```

et obtient la liste du personnel restant après licenciement :

```
Philippe
Jack
André
```

La méthode *reject: aBlock* est basée sur le même principe que *select: aBlock*, mais ne retient que les éléments pour lesquels l'évaluation de *aBlock* donne le résultat *false*. Il reste une dernière et douloureuse opération à réaliser : calculer la somme totale versée en salaires. Pour cela, il évalue avec *show it* :

```
salaire inject:0 into:[:somme :paye | somme:=somme+paye].
```

La méthode *inject: initialValue into: aBinaryBlock* évalue successivement le bloc binaire *aBinaryBlock* en lui donnant comme premier argument *initialValue* et comme deuxième argument la valeur d'une association du dictionnaire. Le bloc est évalué pour chaque association du dictionnaire : le premier argument prend *initialValue* comme valeur initiale, puis sa valeur recalculée le cas échéant pour les évaluations suivantes ; le deuxième argument prend la valeur de chacune des associations du dictionnaire.

Dans notre exemple, la variable *somme* a pris la valeur *0* à la première évaluation, puis *0+paye* à la seconde, etc.

4.4 - Classe IndexedCollection

Classe mère d'une descendance touffue, *IndexedCollection* est aussi la seule dont les éléments sont naturellement accessibles par un index.

Cet index correspond à la position de l'objet dans l'énumération. Il est donc entier, et compris entre 1 et le nombre d'objets de la collection. La plupart des méthodes qui s'ajoutent à celles héritées de *Collection* concerne évidemment la manipulation de cet index.

Dans la classe *FixedSizeCollection* ces nouvelles méthodes sont plutôt orientées vers un accès à une position absolue (je veux accéder à l'objet n° 3, au n° 13, au n° 1) alors qu'elles permettent des accès à des positions relatives dans la sous-classe *OrderedCollection* (je veux accéder au premier objet, au dernier, au suivant, au précédent, au troisième à partir d'ici).

IndexedCollection est une classe abstraite : il n'en existe et ne peut en exister aucune instance. *Collection*, classe abstraite elle aussi, définit plusieurs méthodes (*add:* par exemple) tout en précisant qu'il est nécessaire de les implémenter dans chaque sous-classe : pratiquement aucune de ces méthodes n'est redéfinie dans

IndexedCollection. Il faudra descendre dans ses sous-classes avant de voir une implémentation de *add:*.

Le but de *IndexedCollection* est donc de rassembler quelques méthodes communes à toutes les classes ayant un index numérique.

4.4.1 - Quelques méthodes pour manipuler les index

first — renvoie le premier objet de la collection réceptrice.

last — renvoie le dernier objet de la collection.

findFirst: aBlock — renvoie l'index du premier objet répondant vrai à l'évaluation de *aBlock*.

findLast: aBlock — renvoie l'index du dernier objet répondant vrai à l'évaluation de *aBlock*.

detect: aBlock — renvoie le premier objet du récepteur répondant vrai à l'évaluation de *aBlock* (cette méthode est héritée de la classe *Collection*).

indexOf: anObject — renvoie l'index de *anObject* (de sa première occurrence dans l'énumération plus exactement) ou *0* si *anObject* est absent du récepteur.

indexOf: object ifAbsent:aBlock
idem. avec exécution de *aBlock* si *object* est absent de l'énumération.

4.4.2 - Traiter la collection (méthodes définies dans indexedCollection)

reversed — renvoie une énumération partant du dernier objet du receveur et terminant par le premier.

, aCollection — sélecteur , (virgule) : renvoie une nouvelle collection composée du receveur suivi de *aCollection*.

= aCollection — renvoie vrai si le receveur et *aCollection* appartiennent à la même classe et correspondent à la même collection d'objets.

Cette dernière définition est très importante : la méthode = (égale) renvoie faux si les deux collections ne font pas partie de la même classe (même si leurs contenus sont identiques). Nous avons abordé au chapitre 3.3 les difficultés que cette implémentation introduisait dans la manipulation des chaînes de caractères.

4.4.3 - Modifier des objets de la collection

replaceFrom: start to: stop with: aCollection
remplace les éléments du receveur compris entre les index *start* et *stop* par les éléments de la collection *aCollection*. Il faut, bien sûr, que *aCollection* contienne exactement le nombre d'éléments nécessaires.

replaceFrom: start to: stop with: aCollection startingAt: beg
comme la précédente, mais prend les objets de remplacement à partir de l'index *beg* dans *aCollection*.

replaceFrom: start to: stop withObject: object
remplace tous les éléments du receveur entre les index *start* et *stop* par *object*.

atAll: aCollection put: object
aCollection est un ensemble d'index (tous compris entre 1 et la taille du récepteur). Ce message a pour effet de remplacer tous les objets du receveur situés aux positions énumérées dans *aCollection* par l'objet *object*.

atAllPut: object remplace tous les éléments du receveur par *object*.

4.4.4 - Copier la collection

copyWith: object renvoie une copie du receveur à laquelle est ajouté *object* (en dernière position).

copyWithout: object
renvoie une copie du receveur dans laquelle toutes les occurrences de *object* ont été supprimées.

copy

copyFrom: start to: stop

copyReplaceFrom: start to: stop with: aCollection
ces méthodes renvoient une copie du receveur (entier - tronqué entre *start* et *stop* - avec les éléments de *aCollection* remplaçant les objets du receveur entre *start* et *stop*).

4.4.5 - Effectuer des traitements sur tous les objets d'une collection

De même que dans toutes les autres classes issues de *Collection*, il est possible d'exécuter un bloc pour chaque élément de la collection (méthode *do:aBlock*).

reverseDo: aBlock id. que *do:*, mais commence par le dernier objet du receveur pour aller vers le premier.

with: aCollection do: aBinaryBlock
exécute le bloc à deux variables *aBinaryBlock* en prenant comme premier paramètre un élément du receveur et comme

deuxième paramètre un élément de *aCollection*. Le receveur et *aCollection* doivent avoir la même taille.

collect: aBlock renvoie une collection composée des résultats de l'évaluation de *aBlock* en prenant successivement chacun des éléments du receveur comme paramètre.

L'absence de la méthode *size* confirme le caractère abstrait de la classe *IndexedCollection* : en effet, le calcul de la taille de la collection dépend du type de sous-classe auquel nous avons affaire :

- les instances ont une taille fixe dès leur création (*FixedSizeCollection*),
- les instances peuvent s'accroître, mais dans un sens précis (*orderedCollection*).

Nous allons étudier chacune de ces deux sous-classes en réalisant quelques exercices, qui nous permettront ainsi de mettre en pratique les méthodes que nous venons d'énumérer.

4.5 - La classe FixedSizeCollection

La classe *FixedSizeCollection* étant une classe abstraite, seules ses sous-classes permettent de créer des objets. Celles-ci sont classées selon le type d'objet énuméré par leur instance :

- *String* pour les caractères,
- *Array* pour les éléments d'un tableau (quels qu'ils soient),
- *ByteArray* pour des octets,
- *Interval* pour des nombres.

De ces quatre sous-classes, *Array* est celle qui se rapproche le plus de la conception générale de *FixedSizeCollection*, puisqu'elle permet d'énumérer n'importe quel objet. Elle n'a d'ailleurs que deux méthodes supplémentaires (*printOn:* et *storeOn:*) permettant de traduire ses instances en suite de caractères imprimables.

Un exemple de manipulation de tableau

Newton observait des pommes et découvrit la théorie de la gravité. Einstein griffonnait sur des cabs et découvrit la théorie de la relativité. L'exercice que nous vous proposons vous permettra d'étudier les planètes tout en croquant des pommes et sans avoir besoin de rayer votre carrosserie de voiture pour prendre des notes. Qui sait la théorie que vous découvrirez alors !

Observant le ciel étoilé par une belle nuit d'été, l'envie vous prend de noter les horaires de lever et de coucher des astres les plus connus. Avec la classe *Array* de Smalltalk, rien n'est plus simple.

Soit *saisie* un tableau dans lequel nous allons mémoriser le nom de l'étoile, son heure et son sens d'apparition ou de disparition et *newton* le tableau dans lequel nous allons conserver toutes ces informations.

Il reste à évaluer, à chaque mouvement planétaire, les lignes suivantes :

```
|newton saisie|
saisie := #('lune' $L (Time dateAndTimeNow)).
newton := Array with:saisie.
```

Smalltalk V offrant une classe *Time*, autant s'épargner la lecture fastidieuse du chronomètre (*Time dateAndTimeNow* donne l'heure et le jour au moment de l'évaluation). Le sens de déplacement est indiqué par le caractère (*$L* pour lever et *$C* pour coucher[3]).

Et ainsi de suite. La procédure est un peu lourde si l'on veut saisir plusieurs observations pour les stocker dans le tableau *newton* (nous avons vu en 3.2 qu'il est laborieux de créer une instance de *Array* contenant plus de quatre éléments). Nous allons simplifier notre travail en créant la classe *Astre*, sous-classe de *Array*, dont les instances seront les tableaux concernant une mesure (variable *saisie* de notre exemple).

Dans *Astre*, nous allons ajouter la méthode de classe *entre*, décrite figure 4.7, pour saisir une observation de manière interactive.

```
entre
  "crée et initialise un tableau avec planete, heure et sens"
  | planete heure sens |
  planete := Prompter prompt:'Quelle planete ? '
                      default:''.
  "planete mémorise le nom de l'astre observé"
  heure := Time dateAndTimeNow.
  "heure récupère la date et l'heure courante de la machine
  [#('L' 'C') includes: sens] whileFalse:
    [sens := Prompter prompt:'Lever (L) ou Coucher (C) ? '
                      default:'L'].
  "tant que l'opérateur n'a pas répondu un caractère correct, la question reste posée.
  sens mémorise ensuite le déplacement observé."
  ^(self with:planete with:(sens at:1) with:heure)
  "(sens at:1) renvoie un caractère"
  "la méthode renvoie un tableau chargé des trois informations nécessaires"
```

Figure 4.7 : méthode *entre* pour la classe *Astre*

Cette méthode permet de créer une nouvelle instance de *Astre* en introduisant au clavier les informations nécessaires (appel au *Prompter*).

Il suffit d'évaluer

```
|newton|
 newton := Array with:(Astre entre) with:(Astre entre)
```

pour initialiser *newton*. Pour tester la méthode, nous avons saisi deux observations concernant le soleil et la lune. L'évaluation par l'option *show it* a donné le tableau suivant :

```
(('soleil' $C (Nov 13, 1990 16:58:00))
 ('lune' $L (Nov 13, 1990 16:58:03)))
```

3 Nous aurions aussi pu coder le mouvement par une chaîne ('lever'/'coucher') ou un entier (+1/-1) : la classe Array permet d'énumérer les objets de n'importe quelle classe (y compris Array).

Pour ajouter une nouvelle ligne à notre tableau, nous pourrions utiliser la méthode , (virgule) qui renvoie une concaténation du récepteur et de l'argument :

```
newton := newton,(Array with:(Astre entre))
```

Pour mieux simuler notre expérience, nous allons construire une méthode qui permette de noter les mouvements planétaires en temps réel, et laisse la possibilité à l'opérateur de cesser l'observation quand il le désire. Cette méthode, *observe*, sera définie comme une méthode de classe dans *Astre* (figure 4.8).

```
observe
  "permet de noter au coup par coup les mouvements planétaires  "
  |galilee|
 galilee := Array with:(Astre entre).
  "on suppose qu'il y a au  moins un mouvement planétaire à noter"
  [(Prompter prompt:'D''autres observations (oui/non) ?'
            default:'oui') = 'oui']
  "tant qu'il y a des planètes à  observer. . . "
 whileTrue:[galilee:=galilee,
                    (Array with:(Astre entre))]
  "on ajoute une entrée à la variable tableau galilee" .
  ^galilee
```

Figure 4.8 : méthode de classe *observe*

Cette méthode simplifie l'écriture de la procédure à lancer pour mémoriser les observations.

```
|newton|
newton := Astre observe
```

Cette séquence très simple, évaluée par l'option *show it*, suffit pour lancer la procédure de mesure. Pour en sortir, il suffira de répondre '*non*' à la question '*D'autres observations (oui/non) ?*'. Pour pouvoir effectuer différents traitements sur les observations stockées dans le tableau *newton*, nous allons plutôt utiliser une variable globale (sa valeur sera maintenue même en fin d'évaluation) pour entrer nos observations. La séquence à évaluer au début de la nuit devient alors :

```
Newton := Astre observe
```

Après une nuit d'observation attentive, nous voulons obtenir la liste des planètes qui ont été aperçues. Nous utiliserons la méthode *collect*, qui permet de rassembler dans une nouvelle collection le résultat du traitement de chaque objet de l'énumération, soit :

```
Newton collect:[:planete|(planete at:1) asUpperCase]
```

qui donnera par exemple (avec *show it*):

```
('SOLEIL' 'LUNE' 'VENUS' 'JUPITER' 'SOLEIL')
```

La méthode *asUpperCase*, qui s'adresse à des chaînes de caractères transforme celles-ci en majuscules (Cf 3.3). Elle nous permet d'obtenir des noms de planètes

homogènes, quelle que soit la manière dont ces noms ont été entrés par l'observateur.

Pour vérifier la cohérence de nos données, nous allons extraire du tableau les deux seules lignes concernant le soleil. Pour cela, nous utiliserons la variable *liste* pour récupérer la place du mot *'SOLEIL'* dans le tableau (méthodes *findFirst* et *findLast* qui nous donnent l'index de la première et la dernière position de *'SOLEIL'*). Evaluons avec l'option *show it* la séquence de la figure 4.9.

```
|liste soleil|
 Newton := Astre observe.
 "on suppose que nous avons fait cinq mesures, la première et la troisième concernant le
  soleil"
 liste := Newton collect:[:planete|
                         (planete at:1) asUpperCase].
 soleil := Array with:(Newton at:
          (liste findFirst:[:nom|nom='SOLEIL']))
               with:(Newton at:
          (liste findLast:[:nom|nom='SOLEIL'])).
```

Figure 4.9 : extraction des mesures concernant le soleil

Dans notre cas, nous avons obtenu :

```
(('soleil' $C (Nov 14, 1990 09:51:15))
 ('soleil' $L (Nov 14, 1990 09:51:25))).
```

Nous en déduisons que nous sommes près du pôle ou que notre chronomètre est très mal en point.

Compliquons un peu l'exercice. Après une seconde nuit à scruter les étoiles, nous avons noté huit mouvements planétaires (figure 4.10).

```
|liste|
 Newton := Astre observe."nous ferons huit observations"
 liste := Newton collect:[:planete|
                         (planete at:1)asUpperCase].
 " liste contient le nom des planètes observées, en majuscule" .
```

Figure 4.10 : initialisation des variables *Newton* et *liste*

Complétons la séquence de la figure 4.10 par le message *liste printOn:Transcript* avant d'évaluer le tout par l'option *show it*. Nous voyons alors s'afficher sur la fenêtre *System Transcript* une liste semblable à:

```
('SOLEIL' 'PLUTON' 'LUNE' 'PLUTON' 'VENUS' 'MARS' 'JUPITER' 'SOLEIL')
```

Au vu de cette liste, l'astronome se désespère : Pluton est invisible à l'oeil nu. L'observateur devra corriger la liste en complétant la séquence précédente par :

```
liste copyWithout:'PLUTON'.
```

qui renvoie après évaluation du tout :

```
('SOLEIL' 'LUNE' 'VENUS' 'MARS' 'JUPITER' 'SOLEIL')
```

Ce n'est pas suffisant : le tableau *Newton* n'a pas été corrigé. Connaissant la position des deux observations litigieuses, et mettant en doute l'éveil de son observateur pendant cette période, l'astronome préfère supprimer toutes les mesures comprises entre les deux mouvements de Pluton pour ne garder qu'un tableau plus fiable, *Einstein*[4]. Il va donc évaluer :

```
Einstein := Newton copyReplaceFrom:2 to:4
                   with:#('Douteux').
```

Il obtiendra ainsi un tableau dans lequel les trois entrées litigieuses ont été supprimées au profit d'un seul objet : 'Douteux' :

```
(('soleil' $C (Nov 14, 1990 10:10:38))
 'Douteux'
 ('venus' $L (Nov 14, 1990 10:10:59))
 ('mars' $L (Nov 14, 1990 10:11:04))
 ('jupiter' $C (Nov 14, 1990 10:11:12))
 ('soleil' $L (Nov 14, 1990 10:11:15)))
```

L'observateur ayant expliqué qu'il a confondu Pluton et Saturne, l'astronome préfère alors corriger le tableau *Newton* avec le message *atAll: aCollection put: Object* qui va remplacer les éléments du récepteur aux indices énumérés dans *aCollection* par le nouvel élément *object*.

La position des entrées concernant Pluton est obtenue en examinant la première sortie de la variable *liste* (Pluton est aux indices 2 et 4). L'astronome évalue alors avec *show it*:

```
Einstein := Newton copy. "Einstein est maintenant une copie de l'ancienne
                          version de Newton"
Newton atAll:#(2 4) put:(Astre with:'saturne'
                               with:nil
                               with:nil).
"il faudra récupérer le sens et l'heure dans Einstein"
```

Il obtiendra un tableau comparable à : :

```
(('soleil' $C (Nov 14, 1990 10:10:38))
 ('saturne' nil nil)
 ('lune' $L (Nov 14, 1990 10:10:47))
 ('saturne' nil nil)
 ('venus' $L (Nov 14, 1990 10:10:59))
 ('mars' $L (Nov 14, 1990 10:11:04))
 ('jupiter' $C (Nov 14, 1990 10:11:12))
 ('soleil' $L (Nov 14, 1990 10:11:15)))
```

---

4 Défini comme variable globale.

Il ne restera plus qu'à récupérer l'heure et le sens dans la copie de *newton* avant correction (*Einstein*) pour corriger les lignes '*saturne*'.

Il est temps d'interpréter les données. Pour cela, nous allons calculer le temps écoulé entre le coucher et le lever du soleil (figure 4.11) en utilisant les méthodes *detect:* (pour récupérer la première observation correspondant au soleil) et *reversed detect:* (pour récupérer la dernière).

```
|lever coucher|
 coucher := Newton detect:[:planete|
             ((planete at:1) asUpperCase = 'SOLEIL')
              and:[(planete at:2) = $C]].
 lever := (Newton reversed) detect:[:planete|
             ((planete at:1) asUpperCase = 'SOLEIL')
              and:[(planete at:2) = $L]].
```

Figure 4.11 : détermination des heures de lever et de coucher du soleil

Nous avons renforcé le test évalué par *detect* en vérifiant qu'il s'agit bien du coucher (*$C*) et du lever (*$L*) au cas où un observateur affaibli par le mal de l'altitude aurait multiplié les apparitions nocturnes du soleil. Pour calculer le temps écoulé, complétons la séquence de la figure 4.11 par :

```
(((lever at:3) at:2) asSeconds) - (((coucher at:3) at:2) asSeconds)
```

qui donne 27, une fois évaluée par *show it*.

Le temps que nous avons obtenu de l'horloge de la machine se présente sous la forme d'un tableau contenant la date et l'heure. Il faut d'abord extraire ce temps de la ligne concernée (*lever at:3*) puis isoler l'heure (*at:2*). Le message *asSeconds* permet de traduire l'heure au format normal en un nombre de secondes absolu depuis le début de la journée.

Le plus dur est fait : il ne reste plus qu'à interpréter ce chiffre de vingt-sept secondes. Le lecteur n'oubliera pas ensuite de détruire les variables globales *Newton* et *Einstein* en évaluant :

```
Smalltalk removeKey:#Newton; removeKey:#Einstein.
```

4.6 - La classe OrderedCollection

La seconde sous-classe de *IndexedCollection* n'impose pas de connaître la taille de ses instances dès leur création. Au contraire, elles doivent pouvoir s'agrandir et diminuer à loisir, puisque leur conception les rapproche des listes chaînées.

L'utilisation la plus fréquente de cette classe concerne la simulation de piles, de files d'attente ou de tableaux dynamiques.

Une instance de *OrderedCollection* mémorise non seulement les éléments, mais aussi les liaisons qui les ordonnent. De nombreuses méthodes ont donc pour but de gérer ces liaisons.

Nous verrons d'abord une liste assez complète des nouvelles méthodes de cette classe avant d'en utiliser quelques unes pour simuler une file d'attente chez un médecin.

4.6.1 - Principales méthodes de OrderedCollection

Ajouter des éléments dans la liste

add: object ajoute *object* en fin de liste.

add: newObject after: oldObject
insère *newObject* immédiatement après *oldObject.*

add: object afterIndex: i
insère *object* juste après l'objet en position *i.*

add: newObject before: oldObject
insère *newObject* juste avant *oldObject.*

add: object beforeIndex: i
insère *object* juste avant l'objet en position *i.*

addAllFirst: aCollection
insère tous les éléments de *aCollection* avant le premier objet de la liste.

addAllLast: aCollection
ajoute tous les objets de *aCollection* en fin de liste.

addFirst: object insère *object* en première position.

addLast: object ajoute *object* en dernière position.

at: index put: object
héritée, toujours utilisable.

replaceFrom: i to: j with: aCollection
remplace les éléments de la liste compris entre les positions *i* et *j* par ceux de *aCollection.*

Consulter la liste

at: index toujours utilisable.

after: object rend l'élément suivant *object* (ou une erreur en cas d'absence).

after: object ifNone: aBlock
permet de traiter l'erreur due à l'absence de *object* dans la liste.

before: object rend l'élément précédant *object.*

before: object ifNone: aBlock
id. avec traitement possible de l'absence de *object.*

Enlever un élément

remove: object ifAbsent: aBlock
enlève *object* de la liste ou évalue *aBlock* si *object* n'existe pas dans la liste.

removeFirst enlève le premier élément.

removeIndex: i enlève l'élément d'index *i* de la collection.

removeLast enlève le dernier élément de la liste.

4.6.2 - Un exemple d'utilisation de OrderedCollection : une salle d'attente.

Le docteur Knock a perdu son carnet de rendez-vous. Un pugilat ayant opposé ses patients qui prétendaient tous être arrivés le premier, le médecin décide de les enregistrer désormais dans leur ordre d'arrivée.

La variable *salle* représentera la file d'attente de ses clients. La variable *position* sera utilisée dans la suite pour repérer la place d'un client particulier[5].

```
|salle position|
salle := OrderedCollection new.
```

Au premier coup de sonnette, le docteur Knock évalue :

```
salle add:'Argan'.
```

Lorsque la famille Fenouillard entre, avant qu'il ait eu le temps de terminer d'examiner Argan, il évalue :

```
salle addAllLast:#('Lise' 'Michel' 'Guy' 'Hubert' 'Aline'
                   'Florence').
```

Au moment où Argan sort, Noah entre. Il est blessé et souffre visiblement beaucoup. Le docteur Knock décide de le voir en priorité, soit :

```
salle removeFirst. "Argan part"
salle addFirst:'Noah'. "Noah est inséré en première position"
```

Pendant ce temps, Hubert Fenouillard implore sa maman pour qu'elle le laisse sortir quelques minutes.

```
salle remove:'Hubert' ifAbsent:[nil].
"le paramètre de ifAbsent ([nil]) signifie qu'aucun traitement spécifique n'est réclamé si
Hubert est absent de la liste"
```

5 Dans tout ce paragraphe, on supposera que les séquences données en exemple sont évaluées en partant de la déclaration des variables salle et position.

Mais quand il revient, une bonne demi-heure plus tard, plusieurs clients sont passés :

```
salle add:'Becker';
      add:'Papin';
      add:'Devos';
      add:'Fignon'.
```

Il est hors de question de le rajouter en fin de liste, d'où :

```
salle add:'Hubert' after:'Aline'.
```

Et pour être sûr que l'insertion a fonctionné, on peut évaluer l'expression :

```
salle before:'Hubert'
```

qui renverra '*Aline*'.

Le docteur Knock a remarqué la présence de Monsieur Devos en introduisant un nouveau patient. Il consulte sa liste pour connaître sa position :

```
position := salle indexOf:'Devos'
```

(méthode héritée de IndexedCollection).

Et comme il meurt d'envie de s'amuser, il va subrepticement modifier l'ordre de passage de son chansonnier favori :

```
salle add:'Devos' beforeIndex:(position - 1);
      "ajoute Devos avant la personne qui le précédait"
      removeIndex:(position + 1).
      "supprime l'ancien Devos qui se retrouve maintenant décalé
       d'une position en arrière"
```

Dans cet exemple, nous avons utilisé la notion de file d'attente, très employée en programmation. Outre la simulation d'événements (caisses de super-marché, guichets de banque, péages d'autoroute), l'informatique utilise les files d'attente pour gérer ses propres ressources : l'impression des fichiers sur une imprimante, le lancement de plusieurs processus sur l'ordinateur, la gestion des accès simultanés à un disque font appel à des files d'attente.

Le principe est simple : lorsqu'une ressource est occupée, les nouvelles requêtes pour cette ressource s'accumulent dans une file d'attente. Elles seront prises en charge dans leur ordre d'arrivée (« premier arrivé - premier sorti ») ou dans l'ordre inverse (« dernier entré - premier sorti »). Ce dernier cas s'apparente d'ailleurs au mode de fonctionnement d'une pile, notion également très utilisée en informatique. Une pile s'apparente à la manipulation d'une pile d'assiettes. Celles-ci sont entassées les unes sur les autres, mais seule la dernière empilée reste accessible. La mémoire de l'ordinateur peut aussi être gérée comme une pile.

Les instructions indispensables pour utiliser une pile sont « empiler », « dépiler », « consulter le sommet de la pile ». Les messages Smalltalk correspondants sont *add* (ou *addLast*), *removeLast* et *last*.

Pour simuler une file d'attente de discipline « premier entré - premier sorti », il est nécessaire de disposer au moins des instructions « s'ajoute à la file », « sort de la file », « prochain à sortir ? » qui pourront être respectivement réalisées en Smalltalk avec les méthodes *add*, *removeFirst* et *first*.

4.6.3 - La classe SortedCollection

Sous-classe de *OrderedCollection*, son examen dans le *Class Hierarchy Browser* peut paraître un peu inquiétant : encore une énumération fastidieuse de nouvelles méthodes ! Il n'en est rien. Une grande partie des méthodes créées dans la classe mère *OrderedCollection* n'est pas applicable dans cette classe : toutes ces méthodes sont redéfinies dans *SortedCollection* pour signaler une erreur.

En effet, chaque instance de *SortedCollection* ordonne elle-même ses éléments selon une règle de classement (contenue dans un bloc à évaluer) qui lui a été donnée à sa création (méthode de classe *sortBlock: aBlock* qui crée l'instance et détermine sa règle de classement) ou par la suite (méthode d'instance *sortBlock: aBlock*). La méthode *sortBlock* renvoie le bloc contenant la règle de classement de l'instance réceptrice.

Comme on peut s'en douter, une variable d'instance permet de stocker le bloc de classement. Elle s'appelle évidemment *sortBlock.*

Pour illustrer le fonctionnement de cette classe, prenons l'exemple d'une école. L'instituteur désire obtenir une liste alphabétique de ses élèves. Mais le directeur lui impose de noter aussi le prénom et l'âge de l'enfant concerné. L'instituteur choisit d'écrire les informations concernant un élève dans un tableau. Il évalue avec *do it* la séquence de la figure 4.12.

```
|classe|
 classe := SortedCollection
                     sortBlock:
                     [:eleve1 :eleve2 |
                     (eleve1 at:1) < (eleve2 at:1)].
                   "le bloc de tri compare le premier champ de l'élément qui
                     sera dans notre cas le nom de famille"
 classe add:#(Spangheroo Walter 12);
        add:#(Bidochon  Paul 11);
        add:#(Vautry Raoul 14);
        add:#(Vergniaud Leon 13);
        add:#(Johannes Jules  10)
 classe
```

Figure 4.12 : création d'une liste d'élèves ordonnée alphabétiquement

Le bloc contenant la règle de classement est placé en argument du message *sortBlock:*. Ce doit être un bloc binaire, dont les variables locales représentent deux éléments de l'instance de *SortedCollection*. Pour être placée avant la deuxième variable (*eleve2)* dans l'instance de *SortedCollection*, la première variable (*eleve1*) doit respecter la condition donnée dans la séquence d'instructions (*(eleve1 at:1) < (eleve2 at:1)*).

Le résultat de l'évaluation précédente est *SortedCollection ((Bidochon Paul 11) (Johannes Jules 10) (Spangheroo Walter 12) (Vautry Raoul 14) (Vergniaud Leon 13)).*

Un nouveau ministre impose de conserver des listes par âge décroissant : rien de plus facile. Il suffit d'ajouter le message :

```
classe sortBlock:[:eleve1 :eleve2 |
                    (eleve1 at:3)>(eleve2 at:3)]
```

La variable *classe* désigne alors *((Vautry Raoul 14) (Vergniaud Leon 13) (Spangheroo Walter 12) (Bidochon Paul 11) (Johannes Jules 10))*.

Nous venons d'avoir un aperçu assez complet sur la classe *Collection* et sa descendance. Nous allons maintenant étudier en détail la manière dont Smalltalk V travaille pour manipuler leurs instances.

4.7 - La classe Bag et ses méthodes

4.7.1 Redéfinition des méthodes inadéquates pour la classe Bag

Nous avons vu, dans la classe *IndexedCollection*, une méthode d'accès aux objets par leur position dans la collection. Cette méthode, *at:*, est aussi présente dans la classe *Bag*. Or la classe *Bag*, par définition, ne permet pas d'accéder aux objets contenus dans ses instances par leur position.

Le contenu de la méthode *at:* examiné dans le *Class Hierarchy Browser* est repris figure 4.13.

```
at: anInteger
  "Answer the element of the receiver at index position anInteger.
   Report an error since bags are not indexable."
  self errorNotIndexable
```

Figure 4.13 : la méthode *at:* définie dans *Bag*

Le commentaire, placé entre guillemets, nous explique que les objets de la classe *Bag* ne sont pas indexables, et qu'en conséquence, un message d'erreur est renvoyé. Nous pouvons tester ce message en reprenant notre exemple du chapitre 4.1 pour évaluer :

```
panierDeCourse at: 4.
```

Le message d'erreur qui est alors affiché est envoyé par la méthode *errorNotIndexable* qui est appliquée à notre instance de la classe *Bag*. Mais cette méthode n'existe pas au niveau de la classe *Bag*. Il faut donc la chercher en remontant la hiérarchie des classes. Nous la trouverons très rapidement, au niveau de la classe *Collection*.

Quel intérêt peut-on trouver à définir dans la classe *Bag* deux méthodes (*at:* et *at:put:*) pour expliquer aussitôt qu'elles ne sont pas applicables dans notre cas ?

Les objets de la classe *Bag* n'étant pas indexables, si les méthodes *at:* et *at:put:* existent déjà dans une classe mère de *Bag*, il est nécessaire de les redéfinir dans *Bag* pour éviter des erreurs. Effectivement, lorsque nous remontons de nouveau la hiérarchie des classes pour les rechercher, nous en trouvons une

implémentation dans la classe *Object*, classe mère de toutes les classes du système.

Grâce à la redéfinition de ces deux méthodes, si un utilisateur envoie malgré tout le message *at:* à une instance de la classe *Bag*, un message d'erreur explique que les instances de *Bag* ne sont pas indexables.

Cette construction est nécessaire lorsque des sous-classes voisines ne peuvent utiliser toutes les méthodes de leur classe mère. Par analogie avec une sous-classe voisine, il est fréquent d'envoyer un message inadéquat. L'instance réceptrice tentera quand même de traiter ce message, puisque la méthode correspondante est définie dans une des classes mères (c'est le cas de notre méthode *at:* pour *Bag*). Il est donc conseillé de redéfinir toutes les méthodes inappropriées pour une classe donnée, en affichant un message d'erreur. Si plusieurs sous-classes font appel au même message d'erreur (c'est le cas de *Bag* et de *Set* pour les erreurs d'accès par position), il est plus pratique de placer la méthode affichant l'erreur dans la classe mère (ici, *errorNotIndexable* est dans *Collection*).

4.7.2 - Initialisation de la classe Bag

Lorsqu'une instance est créée dans une classe, il est fréquent que la méthode *new* (méthode définie dans la classe *Behavior* et généralement utilisée pour créer une instance) soit redéfinie localement et fasse appel à une méthode d'instance supplémentaire.

Dans le cas de la classe *Bag*, telle qu'elle est implémentée sous Smalltalk V, cette méthode supplémentaire est *initialize*. Son contenu (figure 4.14) n'est pas très explicite. Le commentaire "*Private*" en tête de méthode signifie qu'elle n'a pas de raison d'être employée telle quelle par l'utilisateur, mais est appelée en complément d'une autre méthode.

Cette méthode est privée, en ce sens qu'elle n'a aucune raison d'être exécutée directement par le programmeur. Seule la méthode de classe *new* y fait appel.

```
initialize
  "Private - initialize the receiver to be empty."
  elements := Dictionary new
```

Figure 4.14 : méthode d'initialisation des instances de *Bag*

La seule activité du message *initialize* est de créer un dictionnaire (classe *Dictionary*) appelé *elements*. L'objet *elements* n'a pas été défini comme variable locale, il n'est pas non plus une variable globale (pas de majuscule en première lettre). Que peut-il être ?

En cliquant sur *Bag* dans la fenêtre des classes, nous voyons s'afficher dans la sous fenêtre inférieure du *Class Hierarchy Browser* la définition de la classe *Bag* (figure 4.15).

```
Collection subclass: #Bag
  instanceVariableNames:
    'elements '
  classVariableNames: ''
  poolDictionaries: ''
```

Figure 4.15 : définition de la classe *Bag*

L'objet *elements* est défini comme une variable d'instance. Autrement dit, chaque instance de la classe Bag a une variable personnelle, *elements*, dont le contenu lui est propre.

4.7.3 - La variable d'instance elements

Toutes les méthodes définies dans la classe *Bag* font appel aux méthodes de *Dictionary*, par l'intermédiaire de la variable d'instance *elements*, définie comme une instance de *Dictionary*. La figure 4.16 montre précisément l'usage qui est fait de ce dictionnaire variable d'instance, en prenant comme exemple la méthode *add:* telle qu'elle est définie dans la classe *Bag*.

```
add: anObject
 "Answer anObject. Add anObject to the elements of the receiver."
 elements
   at: anObject
   put: (self occurrencesOf: anObject) + 1.
 ^anObject
```

Figure 4.16 : méthode *add:* de la classe *Bag*.

A la clé *anObject*, Smalltalk V associe son nombre d'occurrences dans l'instance de *Bag*. Cette valeur augmente d'une unité lorsqu'on ajoute *anObject* dans l'instance de *Bag* propriétaire du dictionnaire *elements*.

Plutôt que d'entasser les objets en vrac dans les instances de *Bag*, Smalltalk V utilise un dictionnaire dont les clés sont les objets eux-mêmes et les valeurs associées le nombre de fois où ces objets apparaissent dans l'instance de *Bag*. Cette manière de construire la classe *Bag* simplifie considérablement les méthodes, comme nous pouvons le vérifier avec *occurrencesOf:*, qui rend le nombre d'occurrences d'un objet dans le receveur. Smalltalk V renvoie simplement la valeur associée dans le dictionnaire *elements* à la clé de nom *anObject*.

La méthode *size* (figure 4.17) fait appel à la méthode *associationsDo:* de *Dictionary* pour totaliser le nombre d'objets de l'instance réceptrice. Il lui suffit de totaliser la valeur (*element value*) de chacune des associations composant le dictionnaire *elements* pour cumuler le nombre d'occurrences de chaque objet composant l'instance de *Bag* et obtenir ainsi la taille de cette instance.

```
size
  "Answer the number of elements in the receiver collection."
  | answer |
  answer := 0.
  elements associationsDo: [ :element |
     answer := answer + element value].
  ^answer
```

Figure 4.17 : méthode *size* de la classe *Bag*.

Dans une classe donnée, Smalltalk V emploie souvent comme variables d'instances des objets appartenant à une classe voisine, parfois même à une classe fille. Il n'est pas rare de trouver de telles imbrications de classes dans Smalltalk V. Nous en verrons de nouveau en étudiant la classe *Stream*.

4.8 - Méthodes directement définies dans la classe Collection

4.8.1 - Ajouter un objet dans une collection

La méthode *addAll:* est définie au niveau de la classe *Collection*. Comme son nom le laisse supposer, elle ajoute en bloc le contenu d'une collection dans une autre (figure 4.18).

```
addAll: aCollection
  "Answer aCollection. Add each element of aCollection to the elements of the receiver."
  aCollection do: [ :element | self add: element].
  ^aCollection  "renvoie l'argument aCollection et non le récepteur modifié"
```

Figure 4.18 : méthode *addAll* définie dans *Collection*

En réalité, elle ne fait qu'appliquer la méthode *add:* à tous les éléments contenus dans la collection à transférer. Nous avons déjà utilisé cette méthode *add:* pour charger *panierDeCourse*. L'examen du *Class Hierarchy Browser* nous montre que *add:* est définie pour la classe *Collection*. Mais cette définition reste assez vague (figure 4.19).

```
add: anObject
  "Answer anObject. Add anObject to the receiver collection."
  ^self implementedBySubclass
```

Figure 4.19 : méthode *Add:* définie dans *Collection*

L'exécution de la méthode *add:* telle qu'elle est définie dans la classe *Collection* envoie un message d'erreur (méthode *implementedBySubclass*, que l'on trouvera dans *Object*). Si le programmeur crée une sous-classe de *Collection* en

oubliant de reconstruire la méthode *add:*, un message d'erreur s'affiche à l'écran, expliquant que *add:* aurait du être redéfini dans cette nouvelle sous-classe.

La variable *panierDeCourse* supportait sans broncher le message *add:*. On a déjà constaté (Cf figure 4.16) qu'il est redéfini dans la sous-classe *Bag*. Plus largement, toute sous-classe de *Collection* doit redéfinir le message *add:*. Sans cela, un message d'erreur s'affichera.

Cette manière d'implémenter les méthodes permet de standardiser la forme des messages qui ont un effet comparable (cas d'un ajout d'objet dans une collection, avec la méthode *add:*). Les sous-classes étant malgré tout assez différentes, la construction pratique de la méthode est laissée à la charge de chacune des sous-classes[6].

Ce point est à garder en mémoire, lors de la création de nouvelles sous-classes : il faut redéfinir toutes les méthodes incomplètes de la classe mère.

4.8.2 - Passer d'une sous-classe de Collection à une autre

La figure 4.20 détaille le contenu de la méthode *asSet*, dont le rôle est de renvoyer la traduction de l'objet récepteur en instance de *Set*.

```
asSet
 "Answer a new Set containing all the elements of the receiver."
  ^(Set new: self size) "crée une instance de Set de la taille du récepteur"
        addAll: self; "ajoute tous les objets du récepteur à cette nouvelle
            instance de Set"
        yourself "renvoie cette nouvelle instance de Set comprenant tous les
        éléments du récepteur"
```

Figure 4.20 : méthode de traduction d'une collection en instance de Set

La méthode *yourself* est définie dans la classe *Object* et renvoie l'objet récepteur (du message *yourself*)[7]. Ici, nous récupérons donc après évaluation de *asSet* l'équivalent de *(Set new:self size) addAll:self.* Une nouvelle instance de *Set* est créée, de taille équivalente à celle du récepteur. Tous les objets du récepteur y sont ensuite chargés par la méthode *addAll:*. C'est cette nouvelle instance contenant les éléments du récepteur qui est renvoyée en fin d'évaluation par la méthode *yourself.*

L'utilisation de *yourself* permet d'éviter l'emploi d'une variable locale. En effet, l'instruction *^(Set new:self size) addAll:self* renverra le contenu de *self*, puisque le dernier message évalué dans la séquence (*addAll:self*) renvoie *self* (Cf figure 4.18). En complétant cette instruction par *;yourself*, le dernier objet renvoyé devient le récepteur de *yourself*, soit la nouvelle instance de *Set* (qui reçoit une cascade de deux messages séparés par le signe ; (point virgule)). Une variable locale aurait pu être employée pour un résultat identique, comme dans la séquence suivante :

6 Principe du polymorphisme, abordé en 1.6.

7 Le corps de sa définition est : ^self.

```
|aSet|
aSet := (Set new size:self) addAll:self.
^aSet
```

Comme *asSet*, les autres méthodes de transfert sont basées sur le principe suivant : créer une instance de la classe cible, et y placer tous les éléments du récepteur.

4.9 - Quelques méthodes de Set

4.9.1 - Ajouter un élément dans une instance de Set

Nous pouvons examiner le contenu de la méthode *add:* telle qu'elle est redéfinie au niveau de *Set* (figure 4.21). Nous allons détailler cette méthode pour illustrer la manière dont Smalltalk gère les éléments d'une collection en mémoire.

L'en-tête nous signale que la méthode ajoute un élément à une instance de *Set* après avoir vérifié qu'il n'existe pas déjà. En effet, une instance de *Set* ne peut contenir plus d'un exemplaire du même élément, contrairement aux instances de *Bag*.

En consultant la définition de la classe *Set*, nous pouvons vérifier que l'objet *elementCount* (utilisé à la treizième ligne) est une variable d'instance. Son nom, et la façon dont elle est utilisée montre qu'il s'agit d'un compteur, qui mémorise en permanence le nombre d'éléments contenus dans l'instance de *Set* à laquelle elle appartient.

```
add: anObject
"Répond anObject. Ajoute anObject au récepteur si le récepteur ne le contient pas  déjà."
 | index |
"index repère la position de anObject ou la première case libre si anObject n'existe pas"
 anObject isNil ifTrue: [^anObject].
"si l'argument est nil, la  méthode s'arrête en renvoyant nil"
 self adjustSize.
"On vérifie qu'il est possible d'accueillir le nouvel objet dans le récepteur"
 (self basicAt:
    (index := self findElementIndex: anObject)) isNil
"si l'objet désigné par index est nil, alors anObject n'est pas encore dans le récepteur et il
 faut l'ajouter"
     ifTrue: [elementCount := elementCount + 1.
              "il y a un élément de plus dans le récepteur"
              ^self basicAt: index put: anObject].
              "on place le nouvel objet à la place vide désignée par index"
 ^anObject
```

Figure 4.21 : méthode *add:* de la classe *Set*

L'évaluation de

```
|unEnsemble|
unEnsemble := Set new. "crée unEnsemble, nouvelle instance de Set"
unEnsemble add:1. "met le chiffre 1 dans unEnsemble"
unEnsemble add:#lapin. "ajoute le symbole lapin dans unEnsemble"
unEnsemble add:'chasseur'. "ajoute la chaîne de caractères 'chasseur' dans
                                       unEnsemble"
unEnsemble size "renvoie le cardinal de unEnsemble"
```

donne 3. Chacun des messages reçus par *unEnsemble* a utilisé sa variable d'instance *elementCount* :

- la méthode *new* fait appel à la méthode *new:*, qui elle-même déclenche l'exécution de *initialize* (mise à 0 de *elementCount*). Comme pour la classe *Bag*, la méthode *initialize* est une méthode privée, appelée par *new:*, et dont le seul but est d'initialiser la variable d'instance.
- La méthode *add:* recalcule *elementCount* (voir fig. 4.21).
- La méthode *size* renvoie tout simplement le contenu de *elementCount*.

4.9.2 - Stockage des objets dans une instance de Set : notion de hachage

Remarquons qu'une instance de *Set* peut enregistrer comme élément n'importe quel objet (nombre, symbole, chaîne, tableau, collection etc.). Comme dans la classe *Bag*, les instances de *Set* répondent aux messages *includes:*, *occurrencesOf:*, *remove: ifAbsent:*, *do:*, et signalent une erreur (instance non indexable) lors de l'exécution de *at:* et *at: put:*.

Le fait qu'un objet ne soit présent qu'en un seul exemplaire autorise des méthodes supplémentaires :

findElementIndex: anObject (à laquelle fait appel *add:*)
rend la position de *anObject*

find: anObject ifAbsent: aBlock
rend la position de *anObject* s'il existe, exécute *aBlock* sinon.

Nous venons pourtant de noter que les méthodes *at:* et *at:put:* (qui font référence à un index ou position de l'objet dans la collection) bloquent sur le message d'erreur « Instance non indexable ». Regardons plus précisément l'effet de ces méthodes.

Dans notre cas, l'évaluation à la suite de la séquence précédente de :

```
unEnsemble findElementIndex: unObjet
```

donnera *1* pour *#lapin*, *5* pour *'chasseur'* et *2* pour *1*.

Or, nous n'avons entré que trois éléments dans notre instance de *Set*. Pour vérifier la taille réelle de *unEnsemble*, complétons les messages précédents par

```
unEnsemble inspect.
```

Le message *inspect* (Cf 3.1.3) a pour effet d'ouvrir une fenêtre d'inspection sur l'instance qui le reçoit. Dans notre cas, nous observons sur la moitié gauche de cette fenêtre la liste des variables nommées liées à notre instance (*self*,

elementCount, et une suite de numéros[8] allant de 1 à 15). En cliquant successivement sur chacune de ces variables, nous observons leur contenu dans la moitié droite de la fenêtre.

La variable *elementCount* affiche bien entendu 3, puisque nous n'avons chargé que trois éléments. Quant aux numéros suivants, il s'agit des quinze cases disponibles que comporte *unEnsemble*. La case 1 contient bien le symbole *#lapin*, la case 5 la chaîne *'chasseur'* et la case 2 le chiffre *1*. Toutes les autres désignent l'objet *nil*. La figure 4.22 reprend cette fenêtre dans laquelle la variable indexée de rang 5 a été sélectionnée.

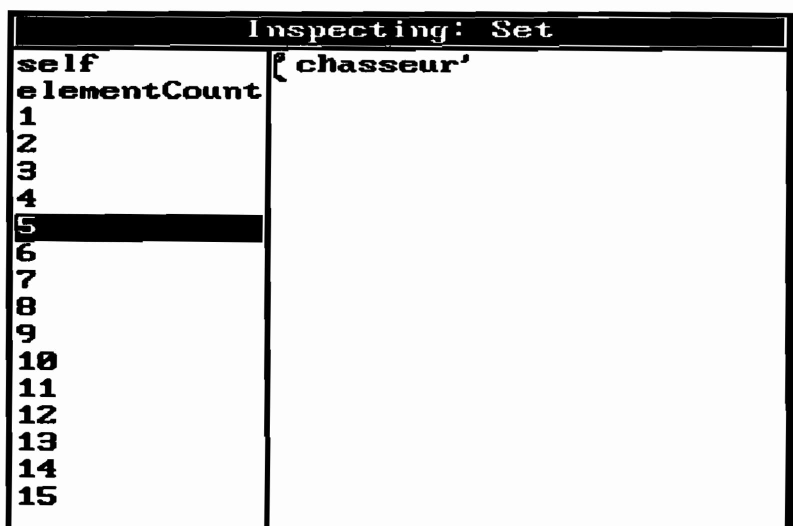


Figure 4.22 : Fenêtre d'inspection d'une instance de *Set*

Smalltalk V utilise le principe du hachage pour placer les objets d'une collection en mémoire. Lorsqu'un utilisateur crée une nouvelle collection, il a l'intention d'y placer une suite d'objets. Nous pourrions mettre ces objets en mémoire dans l'ordre de leur apparition. Ce n'est pourtant pas la meilleure façon de procéder, pour deux raisons.

La première est qu'il est impossible pour Smalltalk de « deviner » à l'avance quelle sera la taille des objets à placer dans la collection, et donc encore moins la taille à réserver en mémoire pour stocker la longue suite d'objets qui pourraient bien faire partie de l'énumération.

La deuxième est que les accès à des objets précis deviennent très longs : le seul moyen de retrouver un objet est de parcourir du début jusqu'à la fin l'espace mémoire alloué à la collection. Pire : à chaque chargement d'un nouvel objet, Smalltalk doit vérifier si cet objet n'existe pas déjà dans l'instance de *Set*, puisque celle-ci n'est pas censée accueillir plus d'une fois le même objet. L'opération conduite par *add:* tiendrait alors du marathon.

---

8 Ce sont les variables d'instance indexées de unEnsemble.

Pour résoudre la première difficulté, Smalltalk ne charge pas les objets eux-mêmes dans l'espace-mémoire alloué à la collection, mais leurs adresses. Les objets sont chargés à un autre endroit. Les langages structurés (C, Pascal) appellent ces adresses des « pointeurs »[9]. La particularité d'une adresse est d'être codée sur un nombre constant de cases mémoires. Si la collection doit accueillir dix objets (qu'il s'agisse d'une chaîne de vingt caractères, d'un tableau de mille éléments ou d'un chiffre), Smalltalk réserve une place égale à la taille de dix pointeurs consécutifs.

Toute création d'instance de la classe *Set* réserve donc une certaine place en mémoire, correspondant au nombre maximum de pointeurs que l'on pourra y stocker. Par défaut, Smalltalk V crée un emplacement mémoire pouvant recevoir quatre pointeurs (message *new: 4* appelé dans la méthode *new*). Le programmeur peut, s'il le désire, réserver plus d'espace (*new: n*).

Nous verrons plus loin comment l'espace réservé en mémoire peut s'accroître, si le programmeur charge plus d'objets qu'il n'en a spécifié avec *new: n*.

Si nous avons réglé la première difficulté, la seconde (perte de temps induite par un chargement linéaire des objets) n'est absolument pas réduite par le stockage de pointeurs sur ces objets. Au contraire, le parcours de la collection est deux fois plus long puisque le système lit pour chaque objet son adresse dans le pointeur, se déplace à cette adresse et y lit l'objet.

C'est à ce niveau qu'intervient la notion de hachage. Son principe est assez simple. Chaque objet étant unique, nous pourrions le « traduire » en un nombre, unique lui aussi, grâce à une fonction adaptée (« fonction de hachage »). Il suffirait ensuite d'allouer un espace mémoire suffisamment grand à la collection, et de placer le pointeur sur l'objet à la position indiquée par le nombre obtenu par hachage.

Un exemple de « traduction » d'un objet de la classe *Character* en nombre serait de le remplacer par son rang dans l'alphabet (entre 1 et 26) si c'est une lettre, ou bien par sa position dans la table des caractères au standard ASCII. Pour une chaîne de caractères (objet de la classe *String*), une fonction de hachage pourrait consister en la concaténation des rangs de chacun des caractères qui la composent (par exemple, '*bac*' se « traduirait » *213*). Nous remarquons que cette fonction n'est pas une bijection, puisque la chaîne '*uc*' donnerait elle aussi le nombre *213*.

Tel quel, ce hachage présente un avantage et deux inconvénients :

- nous avons éliminé les parcours inutiles de l'espace mémoire pour retrouver un objet. La fonction de hachage appliquée à l'objet donne la position dans l'espace mémoire du pointeur qui le désigne. Un seul accès suffit pour aller lire le pointeur, un deuxième pour l'objet lui-même. Quant au chargement d'un nouvel objet, il se fait de la même manière : si l'emplacement indiqué par la fonction de hachage contient un pointeur, c'est que l'objet existe déjà dans la collection ;

9 Les pointeurs sont des types de données au même titre que des entiers, des caractères etc. L'espace mémoire qui leur est alloué dépend de la taille d'une adresse-mémoire et donc du type de machine.

- mais il faut certainement un espace mémoire énorme pour assurer la correspondance entre tous les numéros qui pourraient être obtenus par la fonction de hachage et la position en mémoire ;
- de plus, la fonction de hachage devrait nous donner un nombre unique pour chaque objet. Il est impossible de trouver une telle fonction, sans que le temps de calcul devienne à son tour prohibitif.

En réalité, la mise en oeuvre du hachage est un compromis :

- l'espace mémoire alloué dépasse assez largement le nombre d'objets que l'on prévoit de stocker dans la collection, sans prendre pour autant des proportions anormales (dans notre exemple, nous avions quinze emplacements pour trois objets effectivement stockés). Du coup, le nombre obtenu par la fonction de hachage a toutes les chances de dépasser la dernière position en mémoire de la collection. La fonction de hachage divise donc ce nombre par la taille de la collection et rend le reste de cette division. C'est ce reste (compris entre 0 et la taille de la collection) qui sera utilisé pour placer l'adresse de l'objet considéré.

- D'autre part, il est admis que les fonctions de hachage puissent donner le même nombre pour deux objets différents. Il faut alors gérer ce qu'on appelle une « collision ». Dans ce cas, le programme parcourt les cases mémoires suivantes jusqu'à trouver une case disponible. Pour vérifier la présence d'un objet précis, la méthode est la même : la fonction de hachage donne une position en mémoire, à partir de laquelle le programme recherche systématiquement si l'objet est présent en explorant les adresses successives. Soit l'objet existe, et dans ce cas il a été stocké à la première case disponible, soit il n'existe pas, et dans ce cas, le système finira par tomber sur une case vide.

Ce compromis entre un accès direct à l'objet et l'économie de mémoire optimise les temps d'accès en mémoire, en calcul, et l'espace total réservé à une collection.

Les méthodes chargées d'explorer les objets désignés par les pointeurs sont *basicAt:*, *basicAt:put:*, *basicSize.* Elles sont définies dans la classe *Object*, mère de toutes les classes et font appel à des primitives écrites en langage machine. Leur effet est comparable à celui des méthodes *at;*, *at:put:* et *size* que nous avons vues pour la classe *IndexedCollection.* L'index qui est donné en argument correspond à la position du pointeur dans l'espace alloué à la collection, et non à la position de l'objet dans la collection. Quant à la taille renvoyée par *basicSize*, il s'agit du nombre maximum de pointeurs que peut stocker la collection.

La méthode *findElementIndex:anObject* renvoie la position du pointeur sur *anObject* dans l'espace-mémoire alloué à la collection réceptrice. Si *anObject* n'existe pas dans la collection, cette méthode renvoie la première position inoccupée (nous allons voir dans le paragraphe suivant qu'il y a toujours des positions inoccupées). Cette position vide est en fait occupée par un pointeur sur *nil*[10]. C'est pour cela que la méthode *add:* (figure 4.21) vérifie, avant de charger un nouvel objet, si le pointeur renvoyé par *findElementIndex:* désigne *nil* (dans le cas

10 A l'initialisation, tous les pointeurs désignent l'objet nil.

contraire, il est inutile d'ajouter l'objet puis qu'il est déjà présent dans la collection).

Pour trouver la place mémoire correspondant à la méthode *anObject*, *findElementIndex: anObject* fait appel à la fonction de hachage de *anObject* (dans Smalltalk, cette méthode s'appelle *hash*, quelle que soit la classe considérée).

La méthode *hash* fait appel, selon les classes, à des primitives en langage machine (classe *String*), à une addition bit à bit d'une constante et de la représentation binaire de l'objet (classe *Symbol*) ou récupère le nombre obtenu par le hachage d'un des éléments qui compose l'instance réceptrice (classe *IndexedCollection*, qui renvoie dans certains cas le nombre obtenu par hachage de son premier élément).

Dans tous les cas, un objet ayant toujours une représentation binaire plus ou moins longue dans la mémoire de la machine, la fonction de hachage récupère les derniers bits de cette représentation (les trente-deux derniers dans le cas de la classe *Symbol* sous Smalltalk V) et les interprète comme un entier.

4.9.3 - Croissance d'une instance de Set

A chaque ajout d'objet (méthode *add:*), le système vérifie qu'il reste encore suffisamment de cases libres pour accueillir un nouveau pointeur. C'est le rôle de la méthode *adjustSize* appelée dans *add:*. De manière à ne pas dégrader les performances du hachage, un algorithme vérifie si la proportion de cases vides est toujours acceptable. Si ce n'est pas le cas, un appel à la méthode *grow* génère une nouvelle instance de *Set*, plus grande, dans laquelle vont être recopiés tous les éléments de l'instance initiale, augmentée du nouvel objet à placer.

La figure 4.23 nous permet d'étudier en détail le fonctionnement des méthodes *adjustSize* et *grow*.

```
adjustSize
  "Answer the receiver. If the receiver set is getting full, expand it to accomodate more
   objects."
  (elementCount * 10) >= (self basicSize - 2 * 9)
     ifTrue: [^self grow]


grow
  "Answer the receiver expanded to accomodate more elements."
  | aSet |
  aSet := self species new: self basicSize * 4 // 3 + 10.
  self do: [ :element | aSet add: element].
  ^self become: aSet
```

Figure 4.23 : méthodes *adjustSize* et *grow* de la classe *Set*

Pour savoir si l'instance atteint un taux de remplissage excessif, la méthode *adjustSize* compare le nombre d'objets du receveur (*elementCount*) à sa capacité maximale (nombre fourni par la méthode *basicSize*). La relation numérique

proposée ici correspond à la recherche d'une efficacité optimale de Smalltalk V. Nous pouvons vérifier que l'instance de *Set* que nous avons créée avec quatre cases (*new:4* appelé par *new*) a changé de taille dès le deuxième objet que nous y avons mis. Sa nouvelle taille est donnée par l'algorithme de la méthode *grow* (figure 4.23).

Dans notre cas, la taille est passée à quinze cases. Nous pourrions calculer qu'elle ne changera plus avant le chargement du douzième objet dans la collection.

Nous remarquons trois choses :

- la méthode *grow*, définie dans la classe *Set* pourrait être utilisée dans toutes ses sous-classes. En effet, cette méthode crée une nouvelle instance de la classe à laquelle appartient le receveur, mais de taille un peu plus grande, puis transfère les objets du receveur dans la nouvelle instance avant de renvoyer cette nouvelle instance. C'est le message *species* qui rend cette méthode utilisable par toutes les sous-classes. Son rôle consiste à renvoyer la classe à laquelle appartient le receveur[11]. C'est ensuite à cette classe (et non à la classe mère *Set*) qu'est adressé le message *new:* pour créer une instance.
- Le transfert d'objet est exécuté élément par élément (méthode *do:*) avec *add:*. Or nous avons vu que *add*: teste systématiquement la taille du récepteur avant de lui rajouter l'élément. Récapitulons : pour ajouter un élément dans *unEnsemble*, le système vérifie si *unEnsemble* n'est pas trop rempli. S'il l'est, il va ajouter le contenu d'origine dans une nouvelle instance, plus grande, mais va vérifier pour chaque objet si cette instance est suffisamment vaste. Cette vérification est inutile, puisque la taille de la nouvelle instance est plus grande que celle de *unEnsemble*. Une amélioration possible serait de faire appel à une autre méthode que *add:*, qui éviterait cette vérification de taille.
- *grow* fait appel à la méthode *become:* pour renvoyer la nouvelle collection. En effet, une instruction comme *^aSet* est incorrecte, puisqu'elle ne fait que renvoyer une nouvelle collection sans modifier celle de départ (un peu comme la méthode *asSet* qui renvoie une instance de *Set* à partir du récepteur mais ne transforme en aucun cas le récepteur en instance de *Set*). Le récepteur doit changer de taille, il va certainement changer de place en mémoire. Mais pour le moment, c'est la variable locale *aSet* qui a la bonne taille, contient les bons éléments et se situe à un nouvel emplacement mémoire. La méthode *become:*, définie dans *Object*, inverse correctement le récepteur et l'argument dans la mémoire, y compris toutes les références que des pointeurs extérieurs pourraient faire à l'un ou l'autre.

---

11 Le lecteur pourrait s'étonner de voir species faire le même travail que la méthode class. En fait, class renvoie systématiquement la classe propriétaire de l'instance, alors que species est parfois redéfinie, comme dans FileStream (Cf chapitre 5) pour renvoyer une autre classe. Cette redéfinition s'impose lorsque la méthode new (qui, ici, est appliquée juste après species) est insuffisante pour définir une instance (variables d'instances non définies etc.).

4.10 - La classe Dictionary et ses sous-classes

4.10.1 - Ajouter des éléments à un dictionnaire

Comme pour toute autre sous-classe issue de *Collection*, la méthode *add:* doit être redéfinie pour la classe *Dictionary*.

L'examen de la méthode *add:* redéfinie dans la classe *Dictionary* montre que chaque objet est placé en fonction de sa clé (*findKeyIndex:anAssociation key*) et non de l'association clé/valeur (figure 4.24). Ce n'est pas gênant dans la mesure où une instance de *Dictionary* rassemble des associations dont la clé est unique.

```
add: anAssociation
  "Answer anAssociation. Add anAssociation to the receiver dictionary (hash is by the
   key of anAssociation)."
  | index element |
  index := self findKeyIndex: anAssociation key.
  (element := self basicAt: index) == nil
     ifTrue: [elementCount := elementCount + 1.
              self basicAt: index put: anAssociation]
     ifFalse: [element value: anAssociation value].
  self adjustSize.
  ^anAssociation
```

Figure 4.24 : méthode *add:* de *Dictionary*

Si la clé n'existe pas déjà, la variable héritée de la classe *Set* (*elementCount*) est incrémentée et l'association est chargée dans le dictionnaire. Dans le cas contraire, la nouvelle association donnée en argument de la méthode *add:* remplace l'ancienne dans le dictionnaire récepteur.

Contrairement à la classe *Set*, la vérification de la taille du dictionnaire n'est faite qu'à la fin de la méthode *add:*. De plus, la méthode *grow* est redéfinie (figure 4.25).

```
grow
  "Answer the receiver doubled in size to accomodate more key/value pairs."
  | aDictionary |
  aDictionary := self class new: self basicSize * 2.
  self associationsDo:[:anAssociation |
                      aDictionary add: anAssociation].
  ^self become: aDictionary
```

Figure 4.25 : méthode *grow* de *Dictionary*

La taille du dictionnaire est multipliée par deux, quand elle n'était multipliée que par 4/3 puis augmentée de 10 dans la classe *Set*. Là aussi, il s'agit d'une implémentation destinée à optimiser l'exécution des programmes.

4.10.2 - Limiter les redéfinitions de méthodes dans une sous-classe : l'exemple de select et reject

La conception particulière de la classe *Dictionary* impose de redéfinir un certain nombre de méthodes. Nous allons voir comment limiter le nombre de méthodes à redéfinir.

La classe *Collection* propose deux méthodes pour isoler des objets répondant à une condition (*select:*) ou à son contraire (*reject:*).

La méthode *select:* de *Dictionary* travaille sur la valeur de l'association (figure 4.26) et non sur l'association entière comme le ferait la méthode *select:* telle qu'elle est définie dans *Collection.* Elle est donc redéfinie au niveau de *Dictionary.*

```
select: aBlock
 "For each key/value pair in the receiver, evaluate aBlock with the value part of the pair as
 the argument. Answer a new object containing those key/value pairs for which aBlock
 evaluates to true."
 | answer |
 answer := self species new.
 self associationsDo: [ :each | (aBlock value: each value)
                              ifTrue: [answer add: each]].
 ^answer
```

Figure 4.26 : méthode *select* dans *Dictionary*

La méthode *reject:* travaillant elle aussi sur la valeur de l'association aurait due être redéfinie dans *Dictionary.* Ce n'est pas le cas. La méthode *reject:* n'est définie qu'une seule fois, dans la classe *Collection* (figure 4.27).

```
reject: aBlock
 "For each element in the receiver, evaluate aBlock with that element as the argument.
 Answer a new collection containing those elements of the receiver for which aBlock
 evaluates to false."
 ^self select: [ :element | (aBlock value: element) not]
```

Figure 4.27 : méthode *reject* de la classe *Collection*

Nous voyons qu'elle fait appel à *select:*. De cette façon, il est inutile de redéfinir *reject:* dans chaque sous-classe, puisque l'expression *self select:[...]* est toujours cohérente avec la classe du récepteur.

De nombreuses méthodes font ainsi appel à une méthode principale (*add:* en est un exemple pour la classe *Collection*). Plutôt que de redéfinir chaque méthode dans les sous-classes, il suffit de redéfinir la méthode principale.

4.10.3 - La classe IdentityDictionary : un dictionnaire un peu particulier

IdentityDictionary est une sous-classe de la précédente qui mérite d'être signalée. Toutes les opérations de recherche de clés sont basées sur l'égalité (=) dans le cas de *Dictionary*, de l'identité (==) dans *IdentityDictionary*. Pour mieux saisir cette différence, évaluons la séquence :

```
|Dico cle1 cle2|
Dico := Dictionary new.
cle1 := 'index'.
cle2 := 'index'.
Dico at:cle1 put:'index cle1'.
Dico at:cle2
```

puis cette nouvelle séquence :

```
|Dico cle1 cle2|
Dico := IdentityDictionary new.
cle1 := 'index'.
cle2 := 'index'.
Dico at:cle1 put:'index cle1'.
Dico at:cle2.
```

Dans le premier cas, l'évaluation renvoie *'index cle1'*, dans le deuxième elle signale une erreur *'key is missing'*. En effet, *cle1* et *cle2* ne représentent pas le même objet, même si leur contenu est identique.

L'évaluation de *cle1=cle2* renverra *true* alors que celle de *cle1==cle2* renverra *false*. La méthode = se contente de comparer le contenu de deux objets, alors que la méthode == vérifie que le récepteur et l'argument sont vraiment le même objet.

Les instances de la classe *IdentityDictionary* sont utiles pour manipuler des dictionnaires de symboles, lorsqu'on veut s'assurer que les clés sont uniques dans le système.

La conception des deux classes est légèrement différente. Seule la recherche de l'index d'une clé (méthode *findIndexKey: aKey*) nécessiterait une modification (== au lieu de =). En fait, *IdentityDictionary* dispose d'une variable d'instance supplémentaire, *valueArray*, qui stocke les valeurs correspondant aux clés du dictionnaire.

Le principe est le suivant : dans un espace alloué en mémoire selon les formules des méthodes *grow*, *adjustSize* et *new*, sont stockés les pointeurs sur les clés du dictionnaire. Chaque clé est unique en mémoire. Smalltalk y accède grâce au pointeur qui la désigne. Ce pointeur est stocké dans l'espace alloué au dictionnaire à une position qui dépend de la valeur de hachage de la clé. Le tableau *valueArray* a exactement la même taille que l'espace alloué au dictionnaire. Il peut donc contenir autant de pointeurs que l'espace mémoire du dictionnaire. Mais les pointeurs de *valueArray* désignent les valeurs (et non les clés) des associations du dictionnaire. Pour chaque association du dictionnaire, le pointeur repérant sa clé est mis dans l'espace alloué au dictionnaire, et le pointeur désignant sa valeur est mis dans le tableau *valueArray*. Les deux pointeurs sont placés à la même position dans leur espace mémoire respectif. C'est ce que nous

avons schématiquement représenté sur la figure 4.28, avec *Dico*, instance de *IdentityDictionary*, contenant les associations (cleA ⇨ valA), (cleB ⇨ valB), (cleC ⇨ valC) et (cleD ⇨ valD).

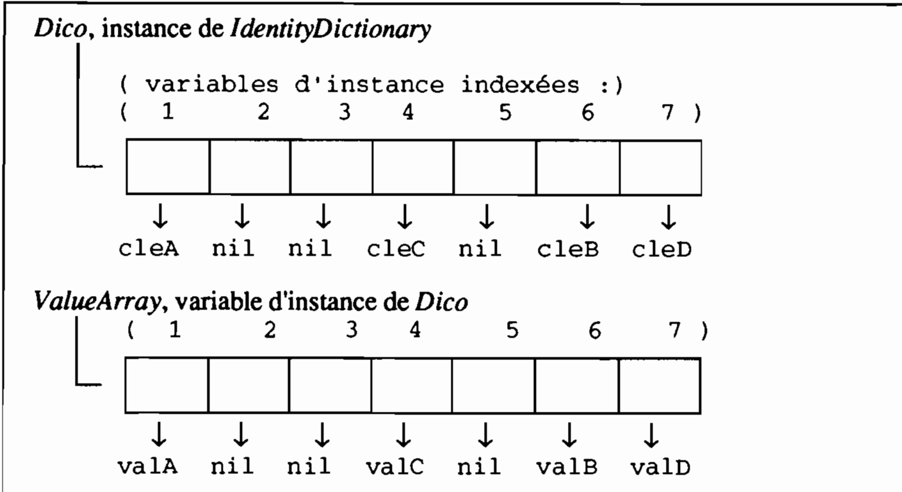


Figure 4.28 : représentation schématique d'une instance de *IdentityDictionary* en mémoire

Pour accéder à une valeur à partir de sa clé, Smalltalk recherche la position du pointeur sur cette clé dans l'espace alloué au dictionnaire (par les méthodes *findKeyIndex* etc.), et regarde à la position correspondante dans le tableau *valueArray* : le pointeur sur la valeur recherchée y est stocké.

Dans la classe *Dictionary*, l'espace alloué au dictionnaire contient les pointeurs sur les associations (et non sur leurs clés, comme dans *IdentityDictionary*).

De nombreuses méthodes d'accès et de manipulation des paires clé/valeur doivent être redéfinies dans *IdentityDictionary*. Nous remarquerons par exemple que la méthode *at:put:* qui faisait appel à *add:* dans la classe *Dictionary* est entièrement reconstruite dans *IdentityDictionary*.

La figure 4.29 illustre la différence d'implémentation pour la méthode *keys*, qui renvoie une instance de *Set* rassemblant les clés.

Dans le premier cas, il explore directement la suite d'associations, conservant uniquement la clé. Dans le second cas, Smalltalk parcourt l'espace alloué en mémoire au dictionnaire (contenant les pointeurs sur les clés), et sélectionne les objets pointés différents de *nil*.

```
keys  "Méthode de Dictionary"
  "Answer a Set containing all the keys of the key/value pair in the receiver dictionary."
  | answer |
  answer := Set new: self size * 2.
  self associationsDo: [:assoc | answer add: assoc key].
  ^answer

keys  "Méthode de IdentityDictionary"
  "Answer a Set containing all the keys of the key/value pairs of the receiver."
  | answer |
  answer := Set new: self size * 2.
  1 to: self basicSize do:[:index |
           (self basicAt:index) == nil
           ifFalse: [answer add: (self basicAt: index)]].
  ^answer
```

Figure 4.29 : comparaison des méthodes *keys* des classes *Dictionary* et *IdentityDictionary*

4.11 - La classe IndexedCollection

IndexedCollection est une classe abstraite. On ne peut donc définir aucune instance directement à son niveau. Les objets pourront être définis dans les sous-classes *FixedSizeCollection*, *orderedCollection* ou *SortedCollection*

4.11.1 - Méthodes héritées de la classe IndexedCollection

Si l'on désire utiliser un index pour accéder à une énumération d'objets, il est indispensable de disposer de quelques opérations de base :

checkIndex: index vérifie que index est cohérent avec l'énumération (index doit être numérique, et compris entre les bornes de l'énumération)

errorInBounds signale une erreur si l'index sort des limites.

Ces deux méthodes sont « privées » (elles n'ont donc pas l'occasion d'être directement appelées par l'utilisateur), mais nous pouvons aisément tester leur efficacité en évaluant pour une chaîne de caractères (*String* est une sous-classe de *IndexedCollection*) l'expression *'9 lettres' at: 14*. Le message *'index 14 is outside collection bounds'* s'affiche en titre d'une fenêtre d'erreur.

Et comme pour d'autres sous-classes de *Collection*, les méthodes de croissance de l'énumération sont redéfinies (*grow* etc.) de manière à optimiser le fonctionnement du système.

4.11.2 - La classe FixedSizeCollection

Comme son nom l'indique, toutes les instances de ses sous-classes ont une taille prédéfinie dès leur création. L'examen des méthodes *with: with:* et *add:* est particulièrement démonstratif (figure 4.30).

```
with: firstObject with: secondObject
 "Answer a new object of two elements, firstObject and secondObject."
  | answer |
  answer := self new: 2.
  answer at: 1 put: firstObject.
  answer at: 2 put: secondObject.
  ^answer
```

Figure 4.30 : méthode *with:with:*

Contrairement à la méthode du même nom définie dans la classe mère *Collection* qui crée une instance contenant plus de places que d'éléments à mettre (4 pour *with:with:*), cette méthode crée exactement le nombre de places demandées dans l'instance (deux). On comprend aisément le commentaire renvoyé par le message *add:object* : "*Inappropriate message for this object*" étant donné que la taille de l'instance ne peut augmenter.

4.11.3 - La classe OrderedCollection

Cette classe permet de créer des listes ordonnées. Comme dans toutes les autres classes issues de *Collection*, à chaque instance de *OrderedCollection* correspond un espace mémoire contenant les pointeurs sur les objets qu'elle contient. Mais contrairement aux autres classes, il n'est plus nécessaire d'optimiser la recherche des objets en mémoire. Au contraire, les instances de *OrderedCollection* correspondent à une liste ordonnée d'objets. Les opérations les plus courantes sont le parcours de la liste dans les deux sens, objet par objet.

Les pointeurs sont donc placés dans l'espace alloué à la liste dans leur ordre de parcours (pointeur sur le premier objet à la première position, sur le deuxième objet à la deuxième position, etc.).

Pour améliorer les performances, il est utile de connaître la position du dernier pointeur stocké dans l'espace alloué à la liste. Ajouter un objet en fin de liste sera ainsi plus rapide. C'est le rôle de la variable *endPosition.*

Une autre variable, *startPosition* est utilisée pour mémoriser l'emplacement du premier pointeur.

La classe *OrderedCollection* permettant d'ajouter un objet en milieu ou début de liste, il est parfois nécessaire de décaler toutes les adresses concernées par cette insertion. Toutes les méthodes d'insertion d'objet font donc appel à des méthodes particulières, au nom explicite (*putSpaceAtEnd, putSpaceAtStart* etc.). Ces méthodes ont des algorithmes assez longs, dont le but est de décaler les adresses dans l'espace alloué à la liste afin d'ajouter l'adresse du nouvel objet à insérer.

4.11.4 - La classe SortedCollection

La classe *SortedCollection* redéfinit pratiquement toutes les méthodes qu'elle partage avec ses classes-mères, en y insérant l'évaluation du bloc de classement, afin de stocker à chaque instant les adresses des objets dans l'ordre imposé par ce bloc de tri.

La classe *Stream* est le complément naturel des instances de *Collection*. Nous avons d'ailleurs souvent éludé l'explication de certaines méthodes (comme *printOn:* ou *storeOn:*) qui faisaient explicitement référence à des instances de *Stream*. Nous étudierons cette classe dans le prochain chapitre.

5 - La classe Stream et sa descendance

5.1 - Premiers pas dans la classe Stream

La figure 5.1 illustre la hiérarchie des sous-classes de *Stream*. La classe *Stream* est elle-même issue de *Object*.

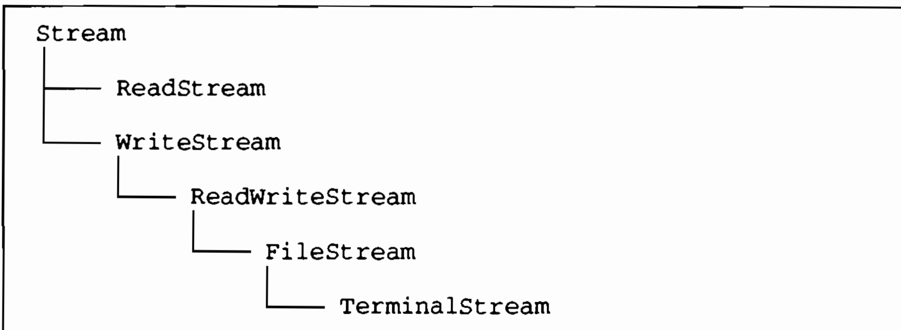


Figure 5.1 : classe *Stream* et ses descendants

Les instances de la classe *Stream* sont utilisées pour manipuler des flux de données (d'objets) issus de collections ou de fichiers[1]. Elles font appel à de nombreuses méthodes de la classe *Collection* et une de leurs variables d'instance est précisément l'instance de la classe *Collection* sur laquelle elles travaillent. La lecture du chapitre précédent est donc nécessaire pour une bonne compréhension de ce qui va suivre.

Nous étudierons, à partir d'exemples très simples, les principales méthodes utilisables dans les classes *ReadStream*, *WriteStream* et *ReadWriteStream* avant de travailler sur un fichier en utilisant une instance de la classe *FileStream*.

---

1 Les fichiers peuvent être assimilés à une collection de caractères.

5.1.1 - Qu'est-ce qu'un flux ?

Une collection d'éléments est sans conteste un objet. Elle a une réalité tangible, dans la mémoire de l'ordinateur (sous forme d'octets) et dans l'esprit de l'utilisateur (sous forme d'une représentation mentale des objets qu'elle contient). Un flux est une notion plus difficile à cerner. En fait, un flux est associé à une action, et en ce sens reste un peu abstrait.

Lorsque nous lisons un livre, notre oeil enregistre les caractères (ou les mots) les uns à la suite des autres. Quand un technicien recopie une série de mesures, il le fait nombre après nombre. Si après cinq cents écritures, il s'aperçoit qu'une mesure a été oubliée, il remonte sa colonne de chiffres jusqu'à trouver l'erreur, la corrige et continue son travail.

Le flux est en quelque sorte la réalisation matérielle de ces actions de lecture et d'écriture. Il n'a pas réellement d'existence physique comme on l'entendrait pour un tableau ou une page de fichier, mais grâce à lui, le parcours d'une collection est possible en lecture, en écriture, par accès séquentiel ou direct.

On ouvre donc un flux sur une énumération d'objets. Cette énumération doit impérativement être ordonnée (un flux parcourt une collection élément par élément : il faut donc que ces éléments aient une position permanente et définie dans la collection). Lorsque le programmeur ouvre un flux sur une collection, Smalltalk crée un objet de la classe *Stream* auquel il associe immédiatement une variable d'instance (*collection*). Cette variable d'instance désigne la collection parcourue par le flux. Comme il s'agit de parcourir la collection, une autre variable d'instance (*position*) mémorise la position courante (celle du dernier objet lu ou écrit). D'autres variables d'instance permettent de connaître la taille maximale de la collection à parcourir (*readLimit*, *writeLimit*).

5.1.2 - Pourquoi une classe réservée aux flux ?

Toutes les opérations offertes par la classe *Stream* et dont nous venons de donner un aperçu restreint sont réalisables dans la classe *Collection* : le message *at:* permet d'accéder directement à un objet de la collection, pourvu qu'on en connaisse sa position ; son alter ego *at:put:* permet d'écrire dans la collection.

Effectivement, toutes les opérations de la classe *Stream* travaillent sur la variable d'instance *collection*. De ce fait, elles utilisent des méthodes définies dans la classe *Collection*. Mais il est infiniment plus simple (et plus conforme à la philosophie orientée « objets ») de créer une classe dans laquelle nous pouvons construire des méthodes communes à des flux s'ouvrant sur des objets aussi divers que des tableaux, des chaînes de caractères, des fichiers etc.

Deux exemples illustreront mieux la nécessité de la classe *Stream*.

Travailler sur un fichier

L'informatique dispose de matériels de plus en plus puissants. La taille des disques durs s'accroît sans cesse. Des fichiers dépassant largement la taille de la mémoire vive de l'ordinateur peuvent être conservés sur ces disques durs.

Supposons que nous voulions parcourir un fichier comportant plus d'un million de mesures stockées les unes à la suite des autres. Si nous assimilons

notre fichier à une instance de la classe *Collection*, il nous faudra manipuler un objet monstrueux (en supposant que nous puissions le charger en mémoire), et qui a toutes chances de saturer les capacités de l'ordinateur.

La classe *FileStream*, descendante de *Stream* permet d'accéder à des fichiers de n'importe quelle taille. Pour cela, elle utilise une petite « fenêtre » qu'elle va déplacer le long du fichier, permettant ainsi à l'utilisateur de le parcourir en toute sécurité, page après page (tampon après tampon). Des méthodes permettent la mise à jour temporaire, la lecture caractère par caractère, ligne par ligne ou mot à mot.

Imprimer ou afficher des nombres

Lorsqu'un comptable signe un reçu, il écrit (ou tape à la machine) « reçu de Monsieur X. la somme de 23440 francs ». Dans cette phrase, tout est caractère, y compris le montant reçu. Pourtant, le caissier n'a pas dicté à son collègue « ... la somme de chiffre deux chiffre trois chiffre quatre... », mais « ... la somme de vingt trois mille quatre cent... ».

De même, la représentation d'un nombre (instance de la classe *SmallInteger* ou autre) dans l'ordinateur est totalement différente de cette écriture en caractères.

Si nous voulons nous-mêmes extraire un nombre d'une base de données pour l'enregistrer dans un fichier texte, ou simplement pour l'afficher à l'écran dans une fenêtre d'édition de texte, nous devrons réaliser une gymnastique numérique importante : repérer le signe, la position de la virgule, diviser successivement par dix, garder les restes, transformer chaque entier en caractère le représentant, avant d'imprimer tous ces caractères dans l'ordre correct.

Un nombre s'écrit comme une suite de chiffres et de signes dans un ordre précis. Partant de ce principe, les concepteurs de Smalltalk ont implémenté dans la classe *Integer* une méthode permettant de transférer dans un flux une représentation imprimable ou affichable des nombres. Cette méthode s'appelle *printOn: aStream.*

Plus largement, tous les objets de Smalltalk disposent de ce message. Dans la classe *Association*, la copie d'une instance dans un flux débute par l'envoi de la représentation en caractères de sa clé, puis des signes ==> suivis de la représentation en caractères de sa valeur.

5.1.3 - Opérations de transfert

Les deux exemples que nous venons de donner font implicitement référence aux transferts : la classe *Stream* est totalement adaptée à ces actions. Les opérations suivantes sont typiquement des manipulations de flux :

- lire le contenu d'un fichier tout en le recopiant octet par octet dans un autre ;
- récupérer la représentation imprimable d'un objet pour le recopier dans un fichier ou une fenêtre d'éditeur de texte ;
- lire les objets d'une instance de *SortedCollection* et les recopier dans une instance de *Array* ;
- lire les caractères d'une instance de *String* et les transférer dans une instance de *MotFrancais* (Cf 3.3).

En ce qui concerne ce dernier exemple, il aurait été beaucoup plus simple de construire la méthode *aPartirDe:* en faisant appel à *Stream* plutôt qu'en recopiant caractère par caractère le contenu de la chaîne paramètre.

5.1.4 - Ouvrir un flux

Avant de créer une instance de la classe *Stream*, il est nécessaire de savoir si nous voulons lire les objets de notre collection (sous-classe *ReadStream*), y écrire de nouveaux objets (sous-classe *WriteStream*) ou bien lire et écrire à la fois (sous-classe *ReadWriteStream*). Les méthodes de parcours ont le même nom dans toutes les sous-classes, mais elles sont parfois redéfinies pour interdire ou permettre l'écriture.

Stream est donc une classe abstraite, et n'autorise pas la création d'instances à son niveau. Elle rassemble pourtant un certain nombre de méthodes communes à toutes ses sous-classes. Deux de ces méthodes sont des méthodes de classe permettant de créer des instances. Elles sont bien entendues inutilisables sous *Stream* (car elles font appel à des méthodes d'instance qui ne sont définies que dans ses sous-classes).

Dans certains cas, il n'est pas utile de parcourir la totalité d'une énumération. La classe *Stream* permet de restreindre la collection parcourue au seul segment intéressant. Pour éviter au programmeur de gérer lui-même les bornes de ce segment (et de courir ainsi le risque d'écrire ou de lire par mégarde hors des limites), Smalltalk copie ce segment dans une nouvelle collection. C'est cette nouvelle collection qui sera associée au flux. Cette manière d'ouvrir un flux offre l'avantage de protéger totalement la collection originale puisque les opérations de lecture et d'écriture se font sur une *copie* limitée au segment. Par contre, les opérations de mise à jour de l'original ne peuvent avoir lieu de cette façon.

A ce stade, nous pourrions comparer les instances de *Stream* à des fenêtres placées sur des collections :

- une fenêtre peut avoir la taille de la collection toute entière ;
- une fenêtre peut être « vitrée » et n'autoriser que l'observation (lecture) ;
- une fenêtre peut être « ouverte » et autoriser le remplacement ou la pose de nouveaux objets (écriture) ;
- plusieurs fenêtres peuvent donner en même temps sur la même collection, côte à côte, ou se chevauchant totalement ou partiellement ;
- une fenêtre en écriture peut modifier une collection observée au même instant par une fenêtre en lecture[2].

2 Il faut que les fenêtres soient ouvertes sur toute la collection, puisque nous avons vu qu'une fenêtre limitée à un segment de collection travaille sur une copie qui lui est personnelle et non sur l'original.

5.2 - Utiliser une instance de ReadStream

5.2.1 - Comment créer une instance de ReadStream ?

Nous pouvons créer une instance de *ReadStream* en utilisant une des deux méthodes de classe héritées de *Stream* :

on: anIndexedCollection
ouvre un flux sur la collection toute entière

on: anIndexedCollection from: firstIndex to: lastIndex
ouvre un flux sur une copie de la collection comprise entre les positions *firstIndex* et *lastIndex*.

Nous avons vu précédemment que les éléments de la collection sur laquelle s'ouvre un flux doivent avoir une position bien identifiée. Nous remarquons dans le libellé des deux méthodes *on:* et *on:from:to:* qu'elles sont limitées aux classes dépendant de *IndexedCollection*, seules classes pour lesquelles chaque objet a une position connue. L'évaluation de la séquence :

```
|unFlux|
unFlux := ReadStream on:(Bag with:'patate'
                             with:'carotte'
                             with:'tomate')
```

signale une erreur précisant que les instances de *Bag* ne sont pas des collections indicées.

Prenons donc le tableau *galerie* comme collection indicée sur laquelle nous appliquerons *visite*, instance de *ReadStream*. La variable *galerie* rassemblera toutes les salles d'une exposition consacrée à la peinture, chaque salle étant réservée à un peintre. L'ordre des salles correspond à l'ordre logique de parcours du bâtiment. Les peintres en vogue pourront avoir plusieurs salles[3], qui ne seront pas toutes consécutives (figure 5.2).

```
|galerie visite|
 galerie := #(Picasso Vinci Dali Picasso David David
             Vinci Monnet).
 visite := ReadStream on:galerie.
```

Figure 5.2 : initialisation d'un flux sur un tableau

Nous sommes prêts pour la visite. Si le temps nous est compté, nous pourrions la réduire en remplaçant par exemple *visite* par la variable *visiteCourte* définie comme suit :

```
visiteCourte := ReadStream on: galerie from: 2 to: 4.
```

La méthode *on:* ouvre un flux sur la collection *galerie* en entier, alors que *on:from:to:* restreint la collection parcourue par *visiteCourte* aux salles 2, 3 et 4 (*Vinci*, *Dali* et *Picasso*).

3 Nous avons choisi de représenter les salles par un symbole ayant le nom du peintre.

Pour vérifier le contenu de la collection que nous parcourons avec *visiteCourte* nous utiliserons la méthode *contents* :

```
visiteCourte contents
```

qui affichera, avec l'option *show it* :

```
(Vinci Dali Picasso)
```

Pour connaître la taille de la collection parcourue, nous pouvons utiliser la méthode *size*, définie dans la classe *Stream*. Remarquons que cette méthode existe aussi dans la classe *Collection* : *visite size* et *visite contents size* donnent donc strictement le même résultat (*8* dans notre exemple).

Pour connaître à tout instant notre position, c'est à dire l'indice du dernier objet que nous venons de lire, nous pourrons évaluer : *visite position* qui nous donnera *0* au départ de la visite (l'objet que nous nous préparons à lire étant en position 1, la position de départ est 0 par convention).

5.2.2 - Parcourir une collection avec un flux

Il reste donc à parcourir la collection. Le ministre de la culture, qui la connaît par coeur, la traverse au pas de charge (figure 5.3).

```
|galerie visiteMinistre|
 galerie := #(Picasso Vinci Dali Picasso David David
             Vinci Monnet).
 visiteMinistre := ReadStream on:galerie.
 [visiteMinistre atEnd] whileFalse:[
                visiteMinistre next printOn:Transcript.
                Transcript cr]
```

Figure 5.3 : visite du ministre de la culture

La séquence de la figure 5.3, évaluée avec l'option *do it*, provoque l'affichage des lignes suivantes dans la fenêtre *Transcript* :

```
Picasso
Vinci
Dali
Picasso
David
David
Vinci
Monnet
```

Nous avons utilisé deux nouvelles méthodes :

atEnd demande au flux récepteur de renvoyer *true* si sa position courante est en fin de collection, *false* sinon.

next le flux récepteur augmente sa variable *position* d'une unité et renvoie l'objet situé à cette position dans la collection qu'il parcourt.
Il faut noter que la méthode *next* déplace la position courante avant de

lire l'objet. La séquence d'expression *galerie position:2. galerie next* renvoie donc le troisième objet et non le deuxième[4].

Sans le savoir, nous avons aussi utilisé deux autres méthodes reliées à la classe *Stream* : ce sont *printOn: aStream* et *cr*. Nous aborderons ces deux méthodes au paragraphe 5.4.2.

Une promotion d'étudiants des Beaux-Arts entre dans la galerie. Le premier se sent mal et se précipite vers la sortie (figure 5.4).

```
|galerie visiteEcourtee|
 galerie := #(Picasso Vinci Dali Picasso David David
              Vinci Monnet).
 visiteEcourtee := ReadStream on:galerie.
 visiteEcourtee setToEnd.
 visiteEcourtee atEnd.
```

Figure 5.4 : placer un flux en fin de collection

La méthode *setToEnd* déplace la position jusqu'à la fin de la collection, ce que nous pouvons vérifier en envoyant le message *atEnd* à *visiteEcourtee* (figure 5.4), qui renverra ici la valeur *true*.

5.2.3 - Découper une collection en sous-collections avec un flux

Les trois étudiants suivants décident de se partager le travail : l'un visitera les deux premières salles, un autre prendra la suite jusqu'au peintre David non compris et le dernier partira de David pour aller jusqu'à la fin (figure 5.5).

```
|galerie visiteProposee visite1 visite2 visite3|
 galerie := #(Picasso Vinci Dali Picasso David David
              Vinci Monnet).
 visiteProposee := ReadStream on:galerie.
 visite1 := ReadStream on:(visiteProposee next:2).
 visite2 := ReadStream on:(visiteProposee upTo:#David).
 visite3 := ReadStream on:(
      visiteProposee copyFrom:visiteProposee position
                     to:visiteProposee readLimit).
 visite1 contents printOn:Transcript. Transcript cr.
 visite2 contents printOn:Transcript. Transcript cr.
 visite3 contents printOn:Transcript. Transcript cr.
```

Figure 5.5 : ouvrir des flux sur les sous-collections obtenues à partir d'un autre flux

4 D'où le nom de la méthode, qui renvoie le prochain objet du flux.

Les trois dernières lignes de la figure 5.5 permettent d'afficher dans la fenêtre *Transcript* le contenu des flux ouverts sur chacune des collections de salles visitées par les étudiants, soit :

```
(Picasso Vinci)
(Dali Picasso)
(David David Vinci Monnet)
```

La méthode *next: anInteger* appliquée à une instance de *ReadStream* renvoie une collection composée des *anInteger* objets extraits de la collection parcourue par le receveur à partir de la position courante (les deux premiers, dans notre exemple). La position courante du receveur est, bien entendu, augmentée de la valeur de *anInteger.*

La méthode *upTo: anObject* renvoie une collection extraite de celle que parcourt le receveur depuis la position courante de ce dernier jusqu'à *anObject* si celui-ci existe ou jusqu'à la fin dans le cas contraire.

La méthode *copyFrom: firstIndex to: lastIndex* renvoie une collection extraite de celle que parcourt le receveur entre les positions *firstIndex* et *lastIndex* comprises. Dans notre exemple, nous avons obtenu la position courante avec la méthode *position* et la dernière position avec *readLimit*[5].

Les flux *visite1*, *visite2* et *visite3* sont totalement indépendants, à la fois de *visiteProposee* et de la collection désignée par la variable *galerie* : le déplacement de la position courante dans *visiteProposee* ne les affectera pas, de même que la modification d'une salle dans *galerie*, puisque ces trois flux travaillent sur des copies partielles de cette collection (et non sur l'original).

5.2.4 - Tester la présence d'un objet sans déplacer la position courante

Deux étudiants se sont trompés : ils sont rentrés par la sortie et cherchent en vain une plaquette de l'exposition. Croisant ceux de leurs camarades qui viennent de terminer la visite, ils leur empruntent leur guide et repartent au début de l'exposition. Celui qui lit la plaquette annonce à l'avance quelle est la première salle à visiter :

```
visiteProposee reset  "remet la position courante de visiteProposee à zéro"
visiteProposee peek  "renvoie le prochain objet à lire, sans modifier la position
                      courante"
```

Evaluées par *show it* à la suite de la séquence de la figure 5.5, ces lignes renverront la salle *Picasso.*

Nos deux étudiants entament la visite, et celui qui n'a pas le catalogue essaie de deviner à l'avance quelle sera la salle suivante (figure 5.6).

Comme on le voit dans la figure 5.6, la seule différence entre les méthodes *nextMatchFor: anObject* et *peekFor: anObject* est que la première avance quoi

5 Dernière position en lecture. La dernière position en écriture sera donnée dans la classe WriteStream par la méthode writeLimit. Ces deux méthodes (readLimit et writeLimit) sont des méthodes « privées » mais leur intérêt dépasse celui d'une stricte gestion interne des instances.

qu'il arrive alors que la seconde n'avance que si l'objet suivant est bien *anObject*. Les deux répondent *true* si l'objet à lire est *anObject*, *false* sinon.

```
|galerie visiteProposee|
 galerie := #(Picasso Vinci Dali Picasso David David
              Vinci Monnet).
 visiteProposee := ReadStream on:galerie.
 visiteProposee nextMatchFor:#Picasso
 "avance d'une salle et répond vrai car l'objet lu est Picasso" .
 visiteProposee nextMatchFor:#Dali
 "avance d'une salle et répond faux car l'objet lu n'est pas Dali, mais Vinci" .
 visiteProposee nextMatchFor:#Dali
 "avance d'une salle et répond vrai car l'objet  lu est Dali" .
 visiteProposee peekFor:#Monnet
 "n'avance pas et répond faux car l'objet suivant n'est pas Monnet mais Picasso"
 visiteProposee peekFor:#Picasso
 "avance d'une salle et répond vrai car l'objet suivant est Picasso"
```

Figure 5.6 : tester la présence d'objets dans une collection parcourue par un flux

5.2.5 - Parcourir une collection de repère en repère

Le dernier étudiant est un fanatique de Vinci. Seule la Joconde l'intéresse. Il va donc se déplacer directement dans les salles Vinci sans s'arrêter aux autres (figure 5.7).

```
|galerie visiteJoconde nombreDeSalles|
 galerie := #(Picasso Vinci Dali Picasso David David
              Vinci Monnet).
 visiteJoconde := ReadStream on:galerie.
 nombreDeSalles := 0
 [visiteJoconde atEnd] whileFalse:[
       (visiteJoconde skipTo:#Vinci)
       ifTrue: [nombreDeSalles := nombreDeSalles +1 ]]
```

Figure 5.7 : parcourir une collection de repère en repère

La méthode *skipTo: anObject* déplace la position courante du receveur sur *anObject* et répond vrai si *anObject* existe, ou bien déplace la position courante jusqu'à la fin de la collection et répond faux si *anObject* est absent de la collection[6]. La variable *nombreDeSalles* totalise le nombre de salles exposant *Vinci* (elle est incrémentée selon le résultat de *skipTo:*).

---

6 Plus précisément de la position courante jusqu'à la fin de la collection.

5.2.6 - Accès direct dans une collection

Un couple croise quelques étudiants dans la salle exposant Monnet, dernière de la galerie, et surprend un commentaire passionné sur la deuxième salle. Le mari comprend qu'il s'agit de la deuxième salle de la galerie et s'y précipite (*visiteMari* de la figure 5.8).

Sa femme comprend qu'il s'agit de la deuxième salle en revenant sur ses pas à partir de Monnet (*visiteEpouse*).

La méthode *skip: anInteger* permet de se déplacer de *anInteger* objets à partir de la position courante, dans un sens ou dans l'autre.

Dernières méthodes pour lesquelles nous ne proposerons pas d'exemples :

isEmpty le receveur répond *true* si la collection qu'il parcourt est vide, *false* sinon ;

do:aBlock permet d'évaluer *aBlock* pour chacun des objets composant la collection parcourue par le receveur, et ce depuis la position courante jusqu'à la fin.

```
|galerie visiteMari visiteEpouse|
 galerie := #(Picasso Vinci Dali Picasso David David
             Vinci Monnet).
 visiteMari := ReadStream on:galerie.
 visiteEpouse := ReadStream on:galerie.
 visiteMari skipTo:#Monnet. "déplace la position courante sur Monnet"
 visiteEpouse skipTo:#Monnet.
 "Le mari et son épouse sont maintenant tous deux dans la salle Monnet"
 visiteMari position:2. "déplace la position de lecture jusqu'au deuxième
                         objet de la collection"
 visiteMari peek.  "renvoie la salle suivante, soit Dali"

 visiteEpouse skip:-2.  "revient en arrière de deux salles"
 visiteEpouse peek.  "renvoie la salle suivante, soit Vinci"
```

Figure 5.8 : accès direct et relatif dans une collection

5.3 - Les sous-classes WriteStream et ReadWriteStream

5.3.1 - Différences entre les sous-classes de Stream

Les classes *WriteStream* et *ReadWriteStream* permettent les opérations d'écriture sur les collections parcourues par leurs instances. La seconde reprend également un certain nombre de méthodes que nous avons étudiées dans la classe *ReadStream* et qui permettent la lecture de la collection parcourue.

Les opérations de lecture et d'écriture découlent en fait d'un nombre réduit de primitives. Seules les méthodes *next* (lecture), *upTo:*(lecture), *nextPut:* (écriture)

et *atEnd* (position) y font référence. Tous les autres messages aboutissent tôt ou tard à l'une de ces quatre méthodes.

Chaque sous-classe redéfinit ainsi un nombre limité de méthodes. L'orientation en lecture de *ReadStream* est due à la définition dans cette classe de la méthode *next*. L'orientation en écriture de *WriteStream* vient de la méthode *nextPut:*. Toute tentative d'écriture sur une instance de *ReadStream* sera sanctionnée par un message d'erreur (*message not understood : nextPut:*). A l'opposé, tout essai de lecture sur une instance de *WriteStream* provoque le message d'erreur *message not understood : next*. Pour pouvoir lire, la classe *ReadWriteStream* redéfinit la méthode *next*.

A ce stade, l'emploi de la classe *WriteStream* peut paraître assez contraignant : en réalité, cette classe est particulièrement adaptée aux opérations de transferts. Les instances de *ReadStream* permettent de lire une collection objet par objet. Le cas échéant, un traitement peut être fait (éliminer certains objets par exemple) avant de recharger les objets sélectionnés dans une instance de *WriteStream*. Le transfert a lieu, et l'emploi de *WriteStream* pour écrire dans une nouvelle collection limite les erreurs de manipulations puisqu'un message de lecture envoyé par mégarde serait immédiatement détecté et refusé.

5.3.2 - Ouvrir un flux en écriture

Comme les instances de *ReadStream*, un flux en écriture s'ouvre sur une collection. La méthode *on:*, héritée de *Stream* peut ne pas convenir dans certains cas, puisqu'elle place le flux en début de collection. La moindre écriture détruirait les objets initiaux un par un. La classe *WriteStream* s'enrichit donc d'une nouvelle méthode de classe, qui permet de placer le flux en fin de collection afin d'en préserver le contenu initial. C'est la méthode *with: aCollection* détaillée à la figure 5.9.

```
with: aCollection
  "Answer an instance of the receiver streaming on aCollection and positioned to the
   end."
  ^(self on: aCollection) setToEnd
```

Figure 5.9 : méthode *with:* de création d'instance dans *WriteStream*

Cette méthode fait appel à *on: aCollection* et place ensuite le flux en fin de collection (*setToEnd*). La méthode de classe *with: aCollection from: firstIndex to: lastIndex* restreint la collection parcourue par le flux à une copie comprise entre les positions *firstIndex* et *lastIndex*.

Reprenons notre exemple de galerie artistique, et supposons que le conservateur décide d'ouvrir de nouvelles salles à la suite de la galerie existante. Il parcourt ces salles en dictant à un de ses collaborateurs les noms des peintres à exposer. Nous utiliserons la méthode *with:* pour créer l'instance de *WriteStream* (*expo*) qui représentera cette action (figure 5.10).

Pour ajouter une nouvelle salle, il suffit d'évaluer :

```
expo nextPut:#VanGogh
```

ou bien, pour en ajouter plusieurs :

```
expo nextPutAll:#(VanGogh VanGogh FraAngelico Raphael)
```

La séquence de la figure 5.10 complète la collection initiale avec cette méthode *nextPutAll:*. Evaluée par *show it*, elle renverra la collection *(Picasso Vinci Dali Picasso David David Vinci Monnet VanGogh VanGogh FraAngelico Raphael)*.

Nous avons vu que les instances de la classe *Array* ont une taille fixe. Pourtant, la collection parcourue et complétée par le flux *expo* est une instance de la classe *Array*. La classe *WriteStream* gère les augmentations de taille des collections parcourues par ses instances. C'est un des avantages qu'elle offre. Pour agrandir un tableau dans un programme, il est pratiquement plus facile d'utiliser un flux que de calculer soi-même les modifications de taille nécessaires en fonction du nombre d'objets insérés dans le tableau. Si nous complétons la séquence de la figure 5.10 par l'expression « *galerie* », nous verrons s'afficher, après évaluation avec l'option *show it*, le contenu de cette variable : tous les objets y sont, mais ils sont suivis par huit objets *nil*. La méthode *nextPut:* (comme *nextPutAll:*) vérifie si la variable d'instance *collection* peut absorber un nouvel objet. Si ce n'est pas le cas, elle lui envoie le message *grow* (Cf 4.9.3). A la réception de ce message, *collection* s'accroit, mais par paliers. Ceci explique l'apparition des nombreux objets *nil* à la suite de nos salles d'exposition.

```
|galerie expo|
 galerie := #(Picasso Vinci Dali Picasso David David
              Vinci Monnet).
 expo := WriteStream with:galerie.
 expo nextPutAll:#(VanGogh VanGogh FraAngelico Raphael).
 expo contents "renvoie la nouvelle collection"
```

Figure 5.10 : modification de la collection après un accès direct

Au cours de cette visite technique, le conservateur décide de modifier la salle *Vinci* pour y placer quelques Toulouse-Lautrec. Il dicte donc dans la foulée cette modification. Pour vérifier l'aspect général de sa galerie, il demande aussitôt à son collaborateur un état actuel de l'exposition (figure 5.11).

L'évaluation de la séquence de la figure 5.11 ne donne pas la totalité des salles de la galerie, mais *(Picasso Toulouse-Lautrec)*. La variable *galerie* n'est pas perdue et contient toujours l'exposition au grand complet, y compris la modification de salle *Toulouse-Lautrec* et une suite d'objets *nil* en fin d'énumération. Nous pouvons nous en assurer en examinant la variable *galerie* à l'issue de la séquence de la figure 5.11. Pour comprendre ce comportement particulier, il sera nécessaire de regarder plus en détail la manière dont est gérée la variable d'instance *collection*. Auparavant, citons deux dernières méthodes utilisables sur les instances de la classe *WriteStream* (et ses sous-classes).

```
|galerie expo|
galerie := #(Picasso Vinci Dali Picasso David David
            Vinci Monnet).
expo := WriteStream with:galerie.
expo nextPutAll:#(VanGogh VanGogh FraAngelico Raphael).
expo position:1. "se place juste avant Vinci pour le remplacer"
expo nextPut:#Toulouse-Lautrec.
expo contents "devrait renvoyer la nouvelle collection"
```

Figure 5.11 : modification d'un élément au milieu de la collection

show: aCollection

comme *nextPutAll:*, rajoute les éléments de *aCollection* à la suite de ceux parcourus par le récepteur (la méthode *show* diffère de *nextPutAll* lorsqu'elle est appliquée à un éditeur de texte : elle provoque alors l'affichage immédiat de *aCollection* dans la fenêtre, contrairement à *nextPutAll*).

next: integer put: object

écrit *integer* fois *object* dans la collection parcourue par le flux receveur, juste après la position courante définie par la variable d'instance *position*.

5.3.3 - La variable d'instance collection et son emploi selon les sous-classes de Stream.

La variable *collection* est définie dans la classe *Stream*. Elle est donc héritée par toutes ses sous-classes. Nous venons de voir qu'il est possible de la modifier en ouvrant sur elle un flux en écriture. Que le flux soit une instance des classes *WriteStream* ou *ReadWriteStream*, il faut retenir quelques points :

- si le flux est créé avec le message *with:* (ou *with:from:to:*), alors l'écriture commence en fin de collection ;
- si le flux est créé avec le message *on:* (ou *on:from:to:*), alors l'écriture commence en début de collection ;
- on peut positionner le flux sur n'importe quel objet de la collection par la méthode *position:* ;
- l'écriture d'objets dans la collection se fait par les méthodes *nextPut:*, *nextPutAll:* ou *show*. Ces écritures ont lieu juste après l'endroit défini par la variable d'instance *position*, et peuvent donc remplacer un objet déjà existant (ou plusieurs) à partir de cette position ;
- les méthodes *on:from:to:* et *with:from:to:* parcourent des *copies* de la collection argument. Celle-ci n'est donc *jamais* modifiée par les opérations d'écriture qui pourront avoir lieu ensuite.

Smalltalk utilise fréquemment les flux en écriture pour la construction de ses méthodes, dès qu'il s'agit de transférer des objets. La figure 5.12 reprend la méthode *next: anInteger* qui retourne les *anInteger* objets suivants la position courante d'un flux en lecture (méthode définie dans la classe *Stream*).

Le message *next: anInteger* a pour effet d'ouvrir un nouveau flux en écriture sur une collection de taille *anInteger*. Les objets sont ensuite lus dans le flux récepteur (message *self next*) pour être aussitôt transférés dans le nouveau flux en écriture (*aStream nextPut:*). Le résultat est obtenu sous la forme d'une collection renvoyée par la méthode *contents*.

```
next: anInteger
  "Answer the next anInteger number of items from the receiver, returned in a collection
  of the same species as the collection being streamed over."
  | aStream |
  aStream := WriteStream on:
              (collection species new: anInteger).
  anInteger timesRepeat: [ aStream nextPut: self next].
  ^aStream contents
```

Figure 5.12 : méthode *next:anInteger* dans *Stream*

A part la possibilité de parcourir en lecture sa variable d'instance *collection*, un flux de *ReadWriteStream* présente des différences de comportement importantes avec une instance de *WriteStream* :

- un flux en lecture/écriture est supposé pouvoir parcourir en tous sens sa variable d'instance *collection*, pour remplacer ou lire un élément. Quelle que soit la position du flux à un instant donné, c'est la collection toute entière qui intéresse l'utilisateur. La méthode *contents* renvoie donc le contenu de la collection parcourue par le flux, du premier élément au dernier (non compris les derniers espaces vides occupés par *nil* si la collection a dû s'accroître pendant une opération d'écriture).
- un flux en écriture est normalement utilisé séquentiellement : n'ayant pas la possibilité de lire les objets, il n'est pas très facile à utiliser pour des remplacements ponctuels d'objets. On s'intéresse essentiellement aux objets qui viennent d'être écrits jusqu'à présent : la méthode *contents* renvoie donc le contenu de la collection parcourue depuis le premier objet jusqu'à la position courante du flux (enregistrée dans la variable d'instance *position*). Si le programmeur a forcé le déplacement du flux vers une position *i* (par le message *position:i*), alors le message *contents* renverra un extrait de la collection parcourue allant des objets de rang *1* à *i*. La figure 5.13 illustre cette différence de comportement des classes *ReadWriteStream* et *WriteStream*.

Il est possible, dans la classe *ReadWriteStream* de limiter la collection renvoyée par la méthode *contents* aux objets compris entre le début de la collection et la position courante. Le message *truncate* signale au flux récepteur que sa position courante devient aussi sa limite de lecture. C'est une manière détournée de simuler le comportement de la classe *WriteStream* chez une instance de *ReadWriteStream* (figure 5.13).

```
|expoFixe galerie expoModifie |
galerie := #(Vinci Raphael MichelAnge FraAngelico)
expoFixe := WriteStream with:galerie.
"expoFixe est déclarée en écriture seule"
expoModifie := ReadWriteStream with:galerie.
"expoModifie est déclarée en  lecture/écriture"
expoFixe position:2.
expoFixe contents printOn:Transcript.
"affiche (Vinci Rafael)"

expoModifie position:2.
expoModifie contents printOn:Transcript.
"affiche (Vinci Rafael MichelAnge FraAngelico)"
expoModifie truncate.
"la position courante de expoModifie devient aussi sa limite de lecture"
expoModifie contents printOn:Transcript.
"affiche (Vinci Rafael)"
galerie printOn:Transcript.
"affiche (Vinci Rafael MichelAnge FraAngelico)"
"la variable galerie n'a pas été modifiée"
```

Figure 5.13 : différences entre *ReadWriteStream* et *WriteStream*

Toute cette gestion des déplacements dans un flux en écriture ou en lecture/écriture utilise trois variables d'instance. Nous allons les aborder plus en détail dans le paragraphe suivant.

5.3.4 - Les variables d'instance de position

Les variables d'instance de position dans les classes *WriteStream* et *ReadWriteStream* sont au nombre de trois :

- *position* enregistre la position courante du flux (c'est à dire la position du dernier objet venant d'être lu ou écrit dans la collection parcourue) ;
- *readLimit* enregistre la position du dernier objet pouvant être lu dans la collection. Nous avons entrevu son rôle avec la méthode *truncate* ;
- *writeLimit* enregistre la position du dernier objet qui peut être écrit dans la collection (en fonction de sa taille actuelle).

Ces trois variables d'instance sont initialisées lors de la création du flux par un appel à une méthode privée, *setLimits*. La figure 5.14 montre les différences d'implémentation de cette méthode selon les sous-classes de *Stream*.

Cas de la classe *ReadStream* :

la seule opération possible est la lecture. La variable d'instance *writeLimit* n'existe pas (elle n'apparaît qu'au niveau de *WriteStream*). La variable *readLimit* prend la taille de la collection parcourue. Le flux est placé en début de collection (*position := 0*). Il est ainsi prêt à lire la collection.

```
setLimits "Méthode de la classe ReadStream"
 position := 0.
 readLimit := collection size

setLimits "Méthode de la classe WriteStream"
 readLimit := position := 0.
 writeLimit := collection size

setLimits " Méthode de la classe ReadWriteStream"
 position := 0.
 readLimit := writeLimit := collection size
```

Figure 5.14 : la méthode *setLimits* pour les classes *ReadStream*, *WriteStream* et *ReadWriteStream*

Cas de la classe *WriteStream* :

la seule opération possible est l'écriture. La variable d'instance *readLimit*, héritée de *Stream*, est initialisée à 0. Par contre, la limite d'écriture (*writeLimit*) prend comme valeur la taille de la collection parcourue. Si l'écriture d'éléments dans la collection nécessite d'augmenter sa taille, alors le message *grow* est envoyé à la variable d'instance *collection* et *writeLimit* prend ensuite la nouvelle taille de la collection (figure 5.15). Sauf appel à la méthode *setToEnd* qui déplacerait la position du flux en fin de collection (par le message *position:readLimit*), la position du flux est initialisée en début de collection. Rappelons que l'appel à la méthode *setToEnd* n'a lieu qu'avec la méthode de classe *with:* (ou *with:from:to:*).

Cas de la classe *ReadWriteStream* :

l'écriture et la lecture sont possibles. Les variables *readLimit* et *writeLimit* prennent donc comme valeur la taille de la collection. La variable *writeLimit* enregistre les augmentations successives de la taille de *collection* si elles ont lieu. Quant à *readLimit*, elle suit la position courante en cas d'écriture de nouveaux objets en fin de collection (on doit évidemment pouvoir lire ce qui vient d'être écrit). Elle ne bouge pas lors des opérations de lecture (elle sert au contraire de borne pour certaines méthodes, comme *contents* ou *setToEnd*) sauf en cas d'appel à *truncate* : dans ce cas, elle prend la valeur de la position courante.

La méthode *nextPut:* décrite figure 5.15 permet d'illustrer le rapport parfois complexe qui existe entre les primitives en langages machine et les instructions Smalltalk. L'écriture d'un objet dans la collection est une action de bas niveau : il faut enregistrer physiquement un nouvel objet en mémoire. La primitive 66 est chargée de ce travail. Mais elle peut échouer dans un cas bien précis : si la position du flux est arrivée en fin de collection, il n'y a plus de place prévue en mémoire pour placer ce nouvel objet. Les instructions Smalltalk qui suivent traitent alors ce cas particulier. Après avoir vérifié (avec la variable *index*) que le flux est en bout de collection, le message *grow* permet d'agrandir celle-ci, et la variable *writeLimit* est réinitialisée. Ce n'est plus une primitive en langage

machine qui écrit l'objet dans la collection, mais une méthode de la classe *Collection* (*at:put:*). Remarquons que seul l'échec de la primitive 66 dû à un dépassement de capacité de la collection est traité. Si la primitive échoue pour une autre raison, Smalltalk laisse le soin de gérer cette erreur à la méthode *at:put:*, puisque l'expression *collection at: index put: anObject* est systématiquement évaluée en cas d'échec de la primitive 66.

```
nextPut: anObject
 "Write anObject to the receiver stream. Answer anObject."
 | index |
 <primitive: 66>
 index := position + 1.
 index > writeLimit
     ifTrue:[collection grow.
             writeLimit := collection size].
 position := index.
 collection at: index put: anObject.
 ^anObject
```

Figure 5.15 : méthode *nextPut:* redéfinie dans *WriteStream*

La méthode *nextPut:* est redéfinie dans la classe *ReadWriteStream* pour mettre à jour *readLimit* : cette variable d'instance prend en compte l'ajout d'un objet et sa valeur augmente d'une unité si nécessaire (si l'objet est ajouté en fin de collection).

5.4 - Travailler sur un fichier avec FileStream

5.4.1 - Comment accéder à un fichier sous Smalltalk ?

Créer un fichier texte à partir de l'environnement Smalltalk

La lecture et l'écriture de fichiers passent aussi par des instances issues de la classe *Stream*. La sous-classe *FileStream* est spécialisée dans ces opérations. Fille de *ReadWriteStream*, ses instances peuvent donc écrire et lire dans les collections qu'elles parcourent.

Le codage des fichiers ayant pour base l'octet, ou le caractère, nombre de méthodes utilisables dans *FileStream* sont spécialisées dans la manipulation d'octets, de caractères et de mots-machine. Nous aborderons, en plus des méthodes de *FileStream*, toutes celles des classes *ReadStream* et *WriteStream* qui concernent plus particulièrement les flux de caractères.

Auparavant, nous allons créer un fichier texte qui servira de base à tous nos exemples. Smalltalk V offre, dans son environnement de travail, une fenêtre spécialisée dans la gestion des fichiers. Cette fenêtre est accessible par le menu principal (cliquer sur l'écran avec le bouton droit de la souris, dans une zone libre de toute autre fenêtre), en choisissant l'option *Browse Disk*. Selon les versions, un

dialogue plus ou moins évolué permet ensuite de choisir le disque sur lequel la fenêtre doit travailler. Une fois le disque choisi et les dimensions de la fenêtre validées avec le bouton gauche de la souris, un espace de travail divisé en trois sous-fenêtres principales s'affiche à l'écran (figure 5.16).

- Le quart en haut et à gauche est réservé à l'affichage et au défilement des répertoires. Un menu particulier permet de créer ou détruire un répertoire et de remettre à jour la fenêtre (si l'on a changé de disquette entre-temps, ou modifié les répertoires par des commandes extérieures à Smalltalk).
- Le quart en haut et à droite contient tous les fichiers du répertoire sélectionné. Un menu permet de créer, renommer, supprimer un fichier, ou de mettre à jour la fenêtre.
- La moitié inférieure de la fenêtre affiche la liste détaillée des fichiers du répertoire étudié tant qu'aucun fichier n'a été sélectionné dans la fenêtre précédente. Si un fichier a été sélectionné (bouton gauche de la souris), cette fenêtre affiche alors le contenu du fichier (en entier si elle le peut, ou bien les premiers et derniers caractères si le fichier est trop important pour tenir en entier dans une fenêtre d'édition de texte). Il est possible de modifier le contenu du fichier. Le menu propre à cette fenêtre permet de sauvegarder, d'effacer, de copier le texte ou de s'y déplacer (cas des gros fichiers).

L'exemple que nous avons choisi figure 5.16 est le fichier *stream.ex1*, appartenant au répertoire *luc* du disque *E:*. Nous remarquons sur la barre de titre que le nombre d'octets disponibles dans le disque est affiché à côté du nom du répertoire de travail (*8273920*). Le fichier étant sélectionné, son contenu est affiché dans la sous-fenêtre inférieure.

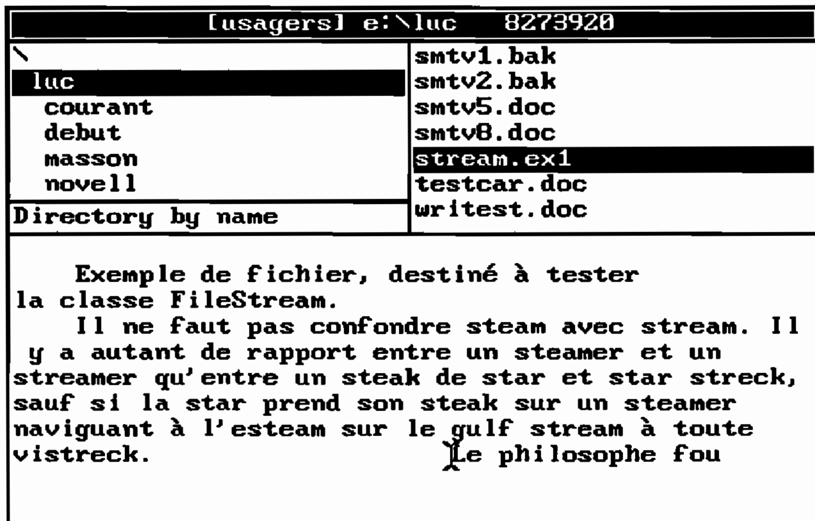


Figure 5.16 : contenu du fichier *stream.ex1*

Représentation « objets » d'un fichier dans Smalltalk

L'accès aux fichiers est une opération de bas niveau. Pour la réaliser, Smalltalk doit échanger des informations avec les modules du système d'exploitation qui gèrent les entrées/sorties. Pour cela, plusieurs classes ont été créées dans l'environnement de base de Smalltalk. Connaître le contenu de ces classes comme le fonctionnement exact de leurs méthodes n'est pas indispensable pour travailler sur des fichiers. Il est quand même utile d'avoir quelques notions de base.

Quatre classes sont associées à la gestion de fichiers :

Directory rassemble les objets répertoires. Les méthodes de classe sont utilisées pour gérer le répertoire courant, créer ou supprimer un répertoire donné etc. Les méthodes d'instance permettent de créer, supprimer, traiter un fichier etc.

File rassemble les objets fichiers. Ces objets ont une représentation en mémoire particulière (il ne s'agit pas du contenu du fichier lui-même mais d'un descriptif permettant à Smalltalk d'accéder au fichier). Plusieurs méthodes permettent de créer, effacer, ouvrir, fermer, sauvegarder ou renommer un fichier.

FileControlBlock
L'accès physique au contenu du fichier passe par les instances de cette classe, structurées en tableaux d'octets (cette classe hérite de *ByteArray*). Cette classe est réservée aux fichiers de format défini par MS-DOS. L'écriture (ou la lecture) d'un fichier se fait bloc par bloc et non caractère par caractère. Les instances de la classe *FileControlBlock* assurent les opérations de lecture et d'écriture sur disque, selon le format requis.

FileStream Les trois classes précédentes sont le support des instances de *FileStream*, qui sont les seuls objets permettant de consulter et de modifier le contenu d'un fichier. Les méthodes de création ou d'ouverture de fichier des classes *Directory* et *File* renvoient d'ailleurs un objet de la classe *FileStream*.

Pour travailler sur un fichier existant, comme nous l'avons fait sur des collections (par exemple *galerie*), la première chose à faire est d'ouvrir une instance de *FileStream* sur ce fichier. Pour cela, nous pouvons envoyer une séquence de messages à la classe *Directory* :

```
(Directory pathName:'e:\luc') file:'stream.exl'
```

qui ouvre un fichier de nom *stream.exl* (*file:'stream.exl'*) dans le répertoire représenté par l'instance de la classe *Directory* que nous avons créée sur le répertoire *e:\luc* (message *pathName:'e:\luc'*). Le même résultat serait obtenu en utilisant la classe *File* :

```
(File pathName:'e:\luc\stream.exl')
```

où le message *pathName:'e:\luc\stream.exl'* ouvre le fichier dont le nom est donné en argument.

Ces deux expressions renverront une instance de *FileStream*. Il n'est pas possible de créer directement une instance par une méthode de classe dans *FileStream*. La méthode *on:* héritée de *Stream* est d'ailleurs redéfinie pour signaler une erreur.

Pour créer un nouveau fichier (et s'assurer qu'un éventuel fichier de même nom sera détruit et remplacé par le nouveau), nous utiliserons la classe *Directory* :

```
(Directory pathName:'e:\luc') newFile:'stream.ex1'
```

ou la classe *File*, avec précaution :

```
(File pathName:'e:\luc\stream.ex1) reset
```

car cette séquence crée le fichier *e:\luc\stream.ex1* s'il n'existe pas, mais l'ouvre sans le détruire s'il existe. Le message *reset* place ensuite le flux en début de fichier. Comme pour les flux sur collections, l'instance de *FileStream* se contente de remplacer les caractères au fur et à mesure de leur écriture. Supposons que le fichier *stream.ex1* existe et contienne la ligne *Première version du fichier.* Rouvert avec la séquence précédente, nous voulons remplacer son contenu par *Nouveau fichier.* Après la sauvegarde, le fichier contient *Nouveau fichiern du fichier.* Les derniers caractères de la première version sont toujours là.

5.4.2 - Copier un fichier existant

Dupliquer un fichier

Nous allons ouvrir deux flux, l'un en lecture sur le fichier *stream.ex1*, l'autre en écriture sur un fichier cible (*stream.bak*). Evaluons maintenant la séquence reprise figure 5.17.

```
|sortie entree|
 entree := File pathName:'e:\luc\stream.ex1'.
 sortie := (Directory pathName:'e:\luc')
             newFile:'stream.bak'.
 entree reset. "positionne le flux en début de fichier"
 [entree atEnd]
          whileFalse:[sortie nextPut:(entree next)].
 sortie close. "ferme le fichier stream.bak"
 entree close. "ferme le fichier stream.ex1"
```

Figure 5.17 : copier un fichier en utilisant deux instances de *FileStream*

L'examen du fichier *stream.bak* dans le *Disk Browser* nous montrera que nous avons créé une copie conforme de notre fichier source.

La séquence que nous venons d'évaluer ne présente pas de méthodes nouvelles, si ce n'est *close*, dont le rôle est de sauvegarder et fermer le fichier correspondant au flux recevant ce message : le flux en lui-même ne se ferme pas, mais il envoie un message de fermeture à l'instance de la classe *File* qui lui correspond. A la réception de ce message, cette instance force l'écriture sur disque des quelques caractères pouvant rester en suspens dans la mémoire (nous avons vu que les

écritures sur disques se font par blocs de caractères : un bloc n'est écrit sur le disque que lorsqu'il est plein, sauf si un message force cette écriture). Le bloc de contrôle correspondant au fichier enregistre ensuite la fermeture (le fichier n'est plus signalé en ouverture et, en cas d'écriture sur le fichier, les informations comme la date, l'heure et la taille sont mises à jour).

Modifier le contenu d'un fichier avant de le copier

Découvrons maintenant quelques méthodes particulières à la classe *FileStream*.

La méthode *nextLine* renvoie la suite de caractères compris entre la position courante et la prochaine marque de fin de ligne. Cette marque de fin de ligne est un caractère spécial et varie selon les systèmes d'exploitation (sous MS-DOS, ce caractère a pour code ASCII 13). Lorsque nous écrivons du texte, chaque retour à la ligne (frappe de la touche *Entrée*) est donc chargé dans le fichier correspondant sous forme d'un caractère de fin de ligne.

Pour ne pas redéfinir à chaque changement de système toutes les méthodes utilisant les marques de fin de ligne, Smalltalk utilise une abréviation pour ce caractère (*Lf*). Cette abréviation est répertoriée dans un dictionnaire partagé auquel ont accès les instances de *FileStream*. Il s'appelle *CharacterConstants*. La correspondance entre l'abréviation *Lf* et le caractère qui sera réellement utilisé est enregistrée dans ce dictionnaire. Il est quand même possible de préparer un fichier pour un autre système d'exploitation que celui sur lequel on travaille en imposant un nouveau caractère comme marque de fin de ligne. C'est le rôle du message *lineDelimiter: aCharacter*. Pour retrouver le caractère de fin de ligne en cours, on emploiera la méthode *lineDelimiter*.

Nous allons maintenant copier le fichier *stream.ex1* dans la fenêtre *System Transcript*. Nous rajouterons, lors de l'affichage, le rang de chaque ligne. La figure 5.18 montre une façon de réaliser cet affichage.

```
|numLigne  entree|
 entree := File pathName:'e:\luc\stream.exl'.
 entree reset. "positionne le flux en début de fichier"
 numLigne := 1. "initialise la variable des numéros de ligne"
 [entree atEnd] whileFalse:
       ['numLigne ' printOn:Transcript.
        numLigne printOn:Transcript. "imprime le numéro de ligne"
        Transcript tab. "ajoute une tabulation"
        entree nextLine printOn:Transcript.
       "imprime la ligne suivante du fichier"
        Transcript cr. "saute à la ligne"
        numLigne := numLigne + 1 "incrémente le numéro de ligne"].
 entree close. "referme le fichier"
```

Figure 5.18 : afficher un fichier ligne par ligne

Nous voyons successivement s'afficher sur la fenêtre *System Transcript* les lignes numérotées du fichier source. Ce texte annoté n'est pas conservé dans un fichier. Modifions légèrement nos instructions pour le récupérer dans un fichier *stream.num* (figure 5.19).

```
|numLigne  entree sortie|
 entree := File pathName:'e:\luc\stream.exl'.
 sortie := (Directory pathName:'e:\luc')
                newFile:'e:\luc\stream.num'
 entree reset.
 numLigne := 1.
 [entree atEnd] whileFalse:
     [numLigne  printOn:sortie.
     "le numéro de ligne est écrit dans le fichier"
      sortie tab.  "une tabulation est écrite dans le fichier"
     "écrit la ligne suivante du fichier entree dans le fichier sortie : "
      sortie nextPutAll:(entree nextLine).
      sortie cr.  "ajoute une marque de fin de ligne au fichier sortie"
      numLigne := numLigne + 1].
 entree close.sortie close.
```

Figure 5.19 : lire un fichier tout en écrivant une version modifiée

L'évaluation par l'option *do it* crée le fichier *stream.num*, qui pourra être consulté dans la fenêtre du *Disk Browser*. Dans ces derniers exemples, nous avons fait appel à trois nouvelles méthodes :

printOn: cette méthode s'applique à pratiquement tous les objets, et imprime leur représentation en caractères dans un flux. Nous l'appliquons ici sur un entier (*numLigne*). Cette méthode est d'ailleurs le moyen le plus simple de récupérer la représentation d'un nombre (entier, réel, ou autre) sous forme de caractères.

tab envoie une marque de tabulation dans le flux récepteur.

cr envoie une marque de fin de ligne dans le flux récepteur. Remarquons que la chaîne de caractères obtenue par le message *nextLine* n'intègre pas les marques de fin de ligne. Il faut rajouter ce code de fin de ligne avec le message *cr*.

Récupérant le fichier source ligne après ligne, nous l'avons copié dans le fichier destination par la méthode *nextPutAll: aCollection*.

Illustrons maintenant l'importance des caractères séparateurs en lisant mot à mot (méthode *nextWord*) le fichier *stream.exl* avant de le copier, toujours mot à mot, dans le fichier *stream.mot* (figure 5.20).

Après évaluation par l'option *do it* de la séquence de la figure 5.20 nous examinerons le contenu de *stream.mot* dans la fenêtre du *Disk Browser*.

`ExempledefichierdestintesterlasousclasseFileStreamIlnefau`
`tpasconfondresteamavecstreamIlyaautantd...` etc.

Les espaces et tous les caractères accentués ont disparus. La méthode *nextWord* lit une chaîne de caractères mot à mot, mais elle est construite pour un environnement anglo-saxon. Elle interprète comme caractères séparateurs tous les caractères accentués français (entre autres), en plus des espaces ou marques de tabulation. Pour aérer un peu notre ligne, nous aurions pu ajouter le message *sortie space*. Nous aurions récupéré les blancs entre chaque mot, mais perdu définitivement les caractères accentués, comme les marques de fin de ligne (à moins de les ajouter avec *sortie cr*).

```
| entree sortie|
 entree := File pathName:'e:\luc\stream.exl'.
 sortie := (Directory pathName:'e:\luc')
            newFile:'stream.mot'.
 entree reset.
 [entree atEnd] whileFalse:[
 "nous ajoutons au fichier sortie chaque mot lu dans le fichier entree :"
      sortie nextPutAll:(entree nextWord)].
 entree close.sortie close.
```

Figure 5.20 : importance des caractères séparateurs

Nous pourrions aussi compter nos blancs avec la méthode *countBlanks*, qui rendra un total correspondant au nombre de blancs et tabulations présents dans le flux (1 par blanc et 4 par tabulation).

Jouer sur les codes de retour de ligne, espace et tabulation offre parfois des possibilités insoupçonnées. Témoin, cet essai de traduction d'un texte en auvergnat (figure 5.21).

```
|entree sortie|
 entree := File pathName:'e:\luc\stream.exl'.
 sortie := (Directory pathName:'e:\luc')
            newFile:'stream.sch'.
 entree reset.
 entree lineDelimiter:$s. "définit le caractère s comme signal de fin de
                                    ligne"
 [entree atEnd] whileFalse:[
       sortie nextPutAll:(entree nextLine ).
      "lit une ligne (de s en s). Le caractère de fin de ligne est maintenant interprété
       comme un caractère normal"
       sortie nextPutAll:'ch'].
      "ajoute les caractères ch en fin de ligne dans le fichier destinataire"
 entree close. sortie close.
```

Figure 5.21 : jouer sur les codes de fin de ligne

La séquence de la figure 5.21 nous donne, évaluée avec *do it* le contenu du fichier *stream.sch* suivant :

```
    Exemple de fichier, dechtiné à techter
la clachche FileStream.
    Il ne faut pach confondre chteam avec chtream. Il y a
autant de rapport entre un chteamer et un
chtreamer qu''entre un chteak de chtar et chtar
chtreck,...
```

Nous avons perturbé la méthode *nextLine* appliquée à *entree*, en choisissant le caractère *$s* comme marque de fin de ligne. Les chaînes de caractères stockées dans *sortie* allaient donc de s en s, considérant l'ancien code de retour de ligne (*Lf*) comme un caractère normal. Le fichier *stream.sch* a donc gardé tous les codes de retour de ligne *Lf* de *stream.exl*. Il suffisait d'avoir auparavant remplacé le s par le ou les caractères désirés (*sortie nextPutAll:'ch'*) pour obtenir notre nouveau texte.

5.4.3 - Manipuler des octets

Les fichiers sont écrits sur les disques comme une suite d'octets ou de mots-machine. Disposer de méthodes ne permettant de traiter que des flux de caractères est insuffisant. Smalltalk V propose deux méthodes (héritées de *WriteStream*), réservées à l'écriture d'octets dans un flux :

nextBytePut: anInteger

charge l'octet de valeur *anInteger* à la position courante dans le flux et déplace cette position d'un octet (un caractère). La valeur de *anInteger* doit être comprise entre 0 et 255. Il peut être sous forme binaire, octale, décimale ou hexadécimale.

nextTwoBytesPut: anInteger

charge le mot de deux octets de valeur *anInteger* à la position courante et déplace celle-ci de deux octets (deux caractères). Conformément aux règles courantes du système MS-DOS et des processeurs INTEL, l'octet de poids faible est chargé en première position, suivi de l'octet de poids fort. La valeur de *anInteger* doit être comprise entre 0 et 65535. Si *anInteger* dépasse ces limites, seuls les deux derniers octets sont pris en considération.

Illustrons l'utilisation de ces deux méthodes par l'exemple suivant, appliqué à une instance de *WriteStream* (figure 5.22).

```
|flux|
 flux := WriteStream on:(String new).
 "On ouvre un flux en écriture sur une chaîne de caractères vide"
 flux nextBytePut:49. "le premier octet écrit correspond à 49"
 flux nextTwoBytesPut:50.
 "les deux octets suivants correspondent à 50 (en base  décimale)"
 flux nextBytePut:60.
 "l'octet suivant à 60 (en base décimale)"
 flux nextTwoBytesPut:12594.
 "les deux octets suivants à 12594 (en base décimale)"
 flux contents.
```

Figure 5.22 : charger des octets et des mots machines dans un flux

Une fois évaluée par *show it* la séquence de la figure 5.22 nous renvoie : *'12 <21'.*

Le code correspondant à *49* est le caractère *1*. Celui de *50* est le caractère *2*. La méthode *nextTwoBytesPut:* s'attend à recevoir un mot de deux octets en paramètre. Comme la valeur décimale *50* se code sur un seul octet, la méthode considère que l'octet de poids fort est égal à 0. L'octet de poids faible représentera *50*. Les deux caractères suivants de notre chaîne sont donc *2* et ' ' (l'octet *0* apparaît comme un espace). Le code de *60* est <.

Reste la représentation en octets du nombre *12594*. En base hexadécimale, ce nombre est égal à 3132h. L'octet de poids faible (31h ou 49 décimal) est le caractère *1* et l'octet de poids fort (32h ou 50 décimal) est le caractère 2.

De cette façon, Smalltalk permet de stocker des nombres codés sous forme binaire dans un fichier. Il reste ensuite à les lire, pour réutiliser ce fichier.

L'exemple de la figure 5.23 montre comment créer une table des codes ASCII dans un fichier *ascii.cod*. Le numéro du code est imprimé dans le fichier par le message *code printOn:sortie*. L'octet correspondant est chargé par l'instruction *sortie nextBytePut:code*. Pour améliorer la présentation de notre table, nous avons inséré des retours à la ligne tous les dix codes ASCII.

```
|sortie |
 sortie := (Directory pathName:'e:\luc')
            newFile:'ascii.cod'.
 0 to:255 do:[:code| code printOn:sortie.
                     sortie nextPutAll:': '.
                     sortie nextBytePut:code.
                     sortie space.
                     sortie nextBytePut:179.
                     (code \\10=0)ifTrue:[sortie cr]].
 sortie contents."affiche le contenu du fichier sur la fenêtre de travail"
 sortie close."ferme le fichier"
```

Figure 5.23 : création d'une table des codes ASCII dans un fichier

La figure 5.24 montre le résultat après évaluation par *show it*.

Figure 5.24 : affichage de la table des codes ASCII

5.4.4 - Comment fonctionne le traitement des flux sur des fichiers

Nous aborderons dans ce paragraphe les méthodes concernant le traitement des flux de caractères, dont une bonne partie est implémentée dans la classe *WriteStream*, puis les méthodes spécifiques aux traitements des fichiers.

Recours à des abréviations

Nous avons utilisé pour modifier nos textes plusieurs méthodes améliorant la présentation (ajouter une tabulation, un espace, une marque de fin de ligne).

La méthode *cr* n'envoie pas un caractère de fin de ligne mais la valeur correspondant à l'abréviation *Lf* :

```
cr
  "Write the line terminating character (line-feed) to the receiver stream."
  self nextPut: Lf
```

Cette abréviation est définie dans le dictionnaire partagé *CharacterConstants* (Cf 2.9.2), auquel les instances de *WriteStream* ont accès.

Dans la définition de la classe, le nom du dictionnaire partagé doit être mentionné après le mot-clé *poolDictionary:* (figure 5.25). Toute instance de *WriteStream* peut ainsi retrouver les caractères les plus courants par un équivalent mnémonique (*Lf*, *Space*, *Tab*).

Il est nécessaire de reprendre cette déclaration dans toutes les sous-classes qui peuvent faire appel aux abréviations qu'elle autorise : les dictionnaires partagés ne sont jamais hérités des classes mères. Nous verrons plus loin que le dictionnaire *CharacterConstants* est de nouveau mentionné dans la définition de la classe *FileStream.*

```
Stream subclass: #WriteStream
  instanceVariableNames:
    'writeLimit '
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants '
```

Figure 5.25 : définition de la classe *WriteStream*

Particularité du retour à la ligne dans la classe FileStream

La méthode *cr* est redéfinie dans la classe *FileStream* pour tenir compte des codages particuliers de la marque de fin de ligne dans les systèmes UNIX et MS-DOS (figure 5.26).

Le changement de ligne correspond à deux opérations mécaniques :

- ramener le curseur en début de ligne (*Cr*) ;
- déplacer le curseur sur la ligne suivante (*Lf*).

Certains systèmes d'exploitation interprètent automatiquement le caractère *Lf* (retour ligne) comme une paire *Cr-Lf.* D'autres exigent au contraire la paire *Cr-Lf* pour passer à la ligne suivante. Il est important de savoir quel est le standard sur lequel on travaille pour gérer correctement les fins de ligne. La classe *FileStream* propose donc une variable d'instance, *lineDelimiter*, dans lequel est mémorisé le caractère reconnu comme marque de fin de ligne par le système d'exploitation. Les méthodes de même nom permettent d'obtenir ou de modifier la valeur de cette variable.

```
cr  "méthode de la classe FileStream"
  "Write the line terminating character (carriage-return line-feed pair or line-feed) to the
   receiver stream."
  self lineDelimiter == Cr ifTrue: [self nextPut: Cr].
  self nextPut: Lf
```

Figure 5.26 : définition de la méthode *cr* dans la classe *FileStream*

La méthode *cr* vérifie si le caractère de fin de ligne est *Cr* (message *lineDelimiter*). Si c'est le cas, alors le système d'exploitation impose la paire *Cr-Lf* comme marque de fin de ligne. La méthode *Cr* envoie alors le caractère *Cr.* Dans tous les cas, elle envoie ensuite le caractère *Lf.*

Les variables d'instance de la classe FileStream

La variable *LineDelimiter* n'est pas la seule qui soit définie au niveau de *FileStream* (figure 5.27). Les instances de cette classe manipulant des fichiers, elles doivent s'adapter à la manière dont sont gérés ces fichiers en informatique.

```
ReadWriteStream subclass: #FileStream
  instanceVariableNames:
    'file pageStart writtenOn lastByte lineDelimiter '
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants '
```

Figure 5.27 : définition de la classe *FileStream*

Une imprimante reçoit les textes à imprimer caractère par caractère. Par contre, un disque communique avec le processeur de l'ordinateur en échangeant des blocs de caractères (granule, secteur, tampon). De cette façon, les pertes de temps induites par les accès mécaniques au disque sont réduites. En contrepartie, cela implique quelques contraintes pour le système d'exploitation :

- la gestion du disque doit se faire en tenant compte de ce partage en secteurs ;
- les opérations de lecture d'un fichier doivent être capables de trouver son premier secteur et d'enchaîner les suivants ;
- lors d'une écriture, il faut attendre que la zone tampon correspondant en mémoire à un secteur soit pleine avant de pouvoir l'écrire sur le disque ;
- si la lecture d'un seul octet est demandée, il faut pouvoir retrouver le secteur dans lequel se situe cet octet, le charger en mémoire dans une zone tampon (« buffer »), puis calculer sa position relative ;
- il faut faire de même pour modifier un caractère, sans oublier de réécrire ensuite la totalité du secteur sur le disque.

La plupart du temps, ces opérations sont transparentes et réalisées par le système d'exploitation. Ce n'est pas le cas pour Smalltalk V sous MS-DOS qui gère lui-même la zone tampon correspondant au secteur lu sur le disque (512 octets).

Pour repérer sa position dans le fichier, la classe *FileStream* dispose de quatre variables d'instance héritées ou définies à son niveau :

lastByte correspond à la position du dernier octet du fichier ;

writeLimit contrairement à la signification qu'elle avait dans les autres classes, cette variable d'instance représente la position du dernier caractère accessible *dans le secteur* en cours de lecture (ou d'écriture) ;

pageStart repère la position dans le fichier du premier caractère du secteur en cours de lecture (ou d'écriture) ;

position représente la position relative du flux dans le secteur en cours de lecture ou d'écriture.

La sauvegarde sur disque du fichier

Cette opération n'est pas automatiquement faite par Smalltalk V : si l'utilisateur reste dans le même secteur (au sens MS-DOS du terme), les modifications qu'il peut écrire ne seront prises en compte qu'à réception de deux instructions :

close qui écrit le secteur courant et ferme le fichier

flush qui écrit le secteur courant sans fermer le fichier.

Les opérations d'écriture sur disque étant relativement longues, les concepteurs de Smalltalk V ont rajouté une nouvelle variable d'instance, *writtenOn*, qui signale si le secteur courant a été modifié ou non depuis la dernière écriture sur disque. Ce booléen est systématiquement testé et remis à jour lors de la sauvegarde sur disque du secteur courant.

La dernière variable d'instance qu'il nous reste à voir est *file*, qui désigne l'instance de la classe *File* sur laquelle le flux se déplace (c'est en quelque sorte la carte d'identité du fichier pour Smalltalk). L'envoi du message *close* (comme *flush*) à une instance de *FileStream* provoque une instruction *file close* (ou *file flush*) qui referme le fichier désigné par cette variable d'instance.

6 - Réaliser une application avec IBM Smalltalk

Nous allons maintenant, à travers un exemple, montrer comment on peut, avec IBM Smalltalk, construire une application complète. Notre objectif est de montrer la puissance et le confort apportés par l'environnement de la programmation orientée objets. Il est aussi d'illustrer la nouvelle approche de cet environnement qui favorise une programmation incrémentale.

6.1 - Quelle application ?

Nous souhaitons ici réaliser une application qui ne se limite pas à un exercice pédagogique et qui correspondra à un logiciel qui pourrait réellement être utilisé, même si, dans le cadre de ce chapitre, nous n'en construirons qu'une maquette. Nous allons reprendre l'étude du dictionnaire analogique, que nous avions abordée en 3.5.4, et construire une application qui permette à un utilisateur de consulter et de mettre à jour un tel dictionnaire. Cette application pourrait par exemple être couplée à l'utilisation d'un traitement de texte, permettant ainsi de modifier la rédaction des textes traités, grâce aux indications fournies par le dictionnaire.

IBM Smalltalk est un logiciel qui fonctionne sous plusieurs environnements : Windows, OS/2, AIX. Dans la suite de ce chapitre, nous avons choisi de présenter une application sous Windows, l'un des environnements les plus courants. Le lecteur pourra ainsi expérimenter facilement cette application.

6.1.1 - Le cahier des charges

Précisons ici les objectifs de l'application. La première contrainte sera de permettre à l'utilisateur, entre deux sessions de travail, de conserver son dictionnaire. Il faut donc que celui-ci soit enregistré dans un fichier et l'application pourra charger, dans un objet, le contenu d'un tel fichier et, bien entendu, effectuer l'opération inverse. Ainsi, on sera en mesure de charger le dictionnaire en début

de session et le sauvegarder en fin de travail. On pourra aussi, en cours de session, changer de dictionnaire, si besoin est[1].

Une autre contrainte sera que l'application puisse être utilisée à travers une interface graphique. On peut ainsi imaginer que l'application, une fois lancée avec l'indication d'un fichier-dictionnaire de départ, charge ce fichier dans un objet et ouvre ensuite une fenêtre à l'écran. Dans cette fenêtre, l'utilisateur pourra consulter la liste des analogies d'un mot donné, passer d'une analogie à la suivante[2], modifier le dictionnaire, changer de dictionnaire, etc. Mais nous ne réaliserons ici qu'une maquette de l'application, et dans la suite, nous laisserons volontairement de côté les détails indispensables à l'obtention d'un produit fini, comme par exemple la vérification d'existence des fichiers. Attirons cependant l'attention du lecteur sur le fait que ces opérations sont avec IBM Smalltalk, moins fastidieuses qu'avec un langage classique : nous en donnerons un exemple dans le paragraphe 6.2.3.

6.1.2 - La structure des fichiers-dictionnaires

Un fichier-dictionnaire sera un fichier dans lequel on conservera de manière permanente les informations d'un dictionnaire analogique. Nous choisirons d'y enregistrer les informations en mode « texte ». Ainsi, ce fichier pourra éventuellement être modifié directement par un éditeur de texte, à condition de respecter les conventions de présentation des informations.

Nous choisirons ces conventions pour que le fichier soit lisible. Une analogie étant représentée par un mot-clé *m* et une liste de mots considérés comme proches de *m*, on la représentera par deux lignes dans le fichier, la première donnant le mot *m* précédé par le libellé *analogies pour*, la seconde, commençant par un retrait de deux espaces pour améliorer la lisibilité, donnera la liste des mots proches de *m*.

```
analogies pour amour
  passion tendresse
analogies pour tendresse
  amour
analogies pour rage
  colère
analogies pour colère
  fureur rage
analogies pour fureur
  colère
analogies pour passion
  amour
```

Figure 6.1 : un exemple de fichier-dictionnaire

1 On peut en effet imaginer qu'un utilisateur se serve de plusieurs dictionnaires, en regroupant des analogies par thèmes, ou encore utilise alternativement, avec le logiciel, un dictionnaire analogique et un dictionnaire des synonymes.

2 Si « rage » est une analogie pour « colère », on peut souhaiter consulter les analogies de « rage » et passer ainsi de proche en proche, etc.

La figure 6.1 donne un exemple de fichier. Une telle structure permet, bien évidemment, de créer le fichier avec un éditeur quelconque. Nous n'imposerons pas à un tel fichier de présenter les informations dans l'ordre alphabétique des mots-clés. En effet, le chargement du fichier dans un objet de la classe *DicoAnalogies* s'effectuera par hachage des mots-clés (cf. 4.9.2) et ne serait donc pas facilité par le fait que le fichier soit ordonné. Cela ne nous empêchera pas, au niveau de l'interface graphique, de présenter la liste ordonnée des mots-clés.

6.2 - Une classe, comme support de l'application

En Smalltalk, une application est toujours représentée par un objet qui est l'instance d'une classe qui intervient dans les traitements réalisés par l'application. Cet objet sera considéré comme le « propriétaire » de l'application, *model* ou *application model* dans la terminologie anglaise de Smalltalk. Ici, cet objet sera une instance de la classe *DicoAnalogies* et il représentera le dictionnaire manipulé. Nous avons déjà défini cette classe en 3.5.4. Reprenons cette définition, en ajoutant la variable d'instance *nomDuFichier* dont le rôle sera de désigner le fichier permanent associé au dictionnaire[3] :

```
Dictionary subclass: #DicoAnalogies
  instanceVariableNames: 'nomDuFichier'
  classVariableNames: ''
  poolDictionaries: ''
```

Un objet de la classe *DicoAnalogies* est donc désormais un dictionnaire (par héritage de *Dictionary*) disposant d'une variable d'instance nommée. Examinons de plus près l'impact de l'adjonction d'une nouvelle variable d'instance.

6.2.1 - La nouvelle variable d'instance et l'héritage

Définissons tout d'abord une méthode pour attribuer une valeur à la variable d'instance *nomDuFichier* :

```
nomDuFichier: uneChaine
  "affecte uneChaine à la variable d'instance nomDuFichier"
 nomDuFichier := uneChaine
```

puis évaluons, avec *Execute*, la séquence de la figure 6.2 :

```
|d| d := DicoAnalogies new.
    d nomDuFichier: 'E:\chap6\dico1.txt'.
    d ajouteAnalogieEntre: 'fragile' et: 'frêle'.
    d inspectionStandard
```

Figure 6.2 : une séquence pour analyser la création d'un dictionnaire d'analogies

3 Une première définition de cette classe a été donnée en 3.5.4. Pour la modifier, on utilisera une fenêtre d'examen des classes (Classes Browser en IBM Smalltalk) dans laquelle on aura sélectionné la classe DicoAnalogies. En cliquant sur le nom de cette classe, on fera apparaître, dans le panneau du bas la définition de la classe. Il suffira alors, de modifier cette définition puis de sélectionner l'option « save » de ce panneau.

On pourra se reporter au paragraphe 3.5.5 pour la description de la méthode *inspectionStandard* qui permet d'examiner complètement[4] l'objet *d*. La figure 6.3 montre deux fenêtres d'inspection obtenues par l'évaluation de la séquence précédente.

Dans la figure 6.3, la première fenêtre est la fenêtre d'inspection sur l'objet *d*, ouverte par l'évaluation de la séquence précédente. La deuxième fenêtre est la fenêtre d'inspection de la variable *associationSet*, obtenue en double cliquant sur la variable *associationSet* dans le panneau d'inspection précédent.

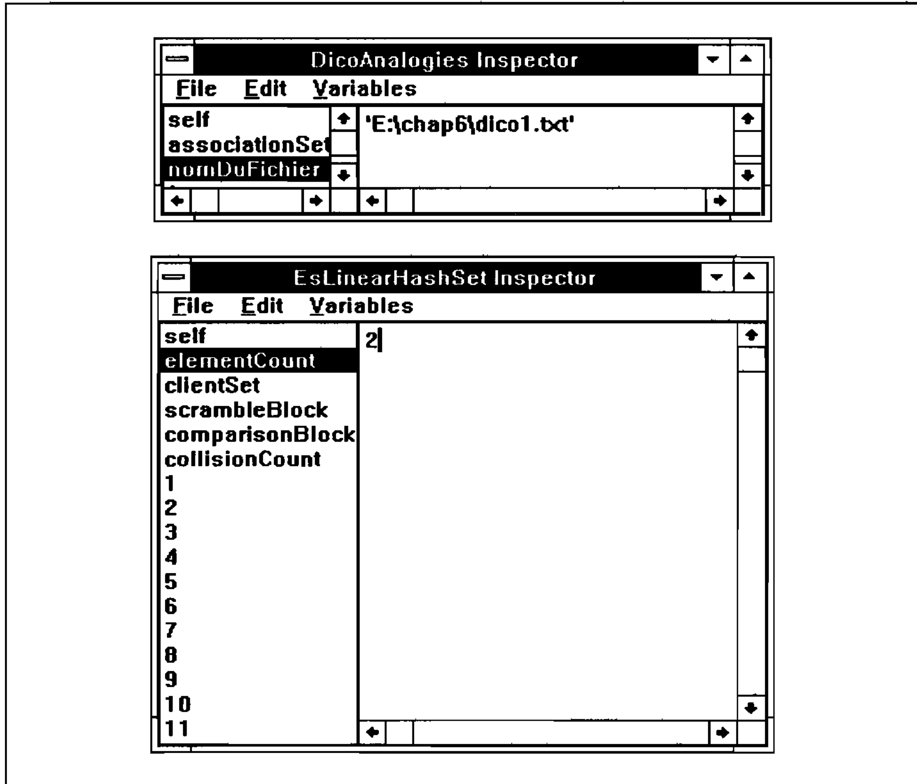


Figure 6.3 : les variables *associationSet* et *nomDuFichier* de l'objet *d*

La première fenêtre indique que la valeur de la variable d'instance *nomDuFichier* est *'E:\chap6\dico1.txt'*. C'est bien ce que nous pouvions attendre puisque la séquence évaluée affecte la chaîne *'E:\chap6\dico1.txt'* à cette variable.

Dans la deuxième fenêtre, la variable d'instance *elementCount* est sélectionnée. Sa valeur indique que le dictionnaire a enregistré deux analogies. Ce sont les analogies *'frêle'* ⇨ *'fragile'* et *'fragile'* ⇨ *'frêle'*. On pourrait vérifier qu'elles sont

4 L'utilisation de inspect, héritée de Dictionary, ne montrerait que les associations du dictionnaire et ne ferait pas apparaître ses variables d'instances nommées. En particulier, on ne pourrait examiner la variable nomDuFichier.

enregistrées respectivement dans les variables d'instance indexées 6 et 9 : il suffirait, pour cela, de cliquer successivement sur les numéros de ces deux variables dans le panneau de gauche.

On peut être surpris que cette fenêtre d'inspection fasse apparaître onze variables d'instance indexées alors que la méthode *new* utilisée pour créer le dictionnaire, crée un objet sans variables d'instance indexées. Pour comprendre l'origine du problème, il faut étudier de plus près les classes *Dictionary* et *EsLinearHashSet.*

Pour conserver au hachage une efficacité suffisante, chaque ajout d'une association à une instance de la classe *Dictionary* par la méthode *add:* (respectivement par la méthode *at:put:*) déclenche, l'envoi du message *dictAdd:* (respectivement du message *dictAt:put:*) à la collection des clés du dictionnaire, instance de la classe *EsLinearHashSet.* Ce message appelle, si nécessaire, la méthode *growEmptyBy:* de la classe *EsLinearHashSet,* qui provoque un agrandissement de la capacité du dictionnaire. Or, dans la séquence de la figure 6.2, le message :

```
d ajouteAnalogieEntre: 'fragile' et: 'frêle'.
```

fait deux ajouts successifs au dictionnaire. Lors du premier ajout, la taille du dictionnaire est fixée à onze éléments. Puis lors des ajouts suivants, la condition qui, dans la méthode *dictAdd:*, déclenche l'envoi du message *growEmptyBy:* est :

```
(elementCount := elementCount + 1) > (elementsSize // 2)
```

on peut donc affirmer que :

- après le premier ajout et le deuxième ajout, le nombre d'associations dans le dictionnaire (2 respectivement 4 associations) sera inférieur à 11//2 (c'est à dire 5), la taille de d restera de onze éléments,
- après le troisième ajout, le nombre d'associations dans le dictionnaire (6 associations) sera supérieur à 5 et le dictionnaire s'agrandira.

6.2.2 - L'interface avec les fichiers

Nous avons expliqué, en 6.1.2, comment nous souhaitions pouvoir conserver un dictionnaire dans un fichier permanent et nous avons précisé la structure d'un tel fichier. Ajoutons maintenant à la classe *DicoAnalogies* les deux méthodes d'instance permettant de charger et conserver un dictionnaire (*chargeDuDisque* et *sauveSurDisque*). Elles sont présentées aux figures 6.4 et 6.5.

Comme son nom l'indique, la première de ces méthodes ouvre un fichier-dictionnaire dont le nom est obtenu en interrogeant l'utilisateur (dialogue via une fenêtre *CwTextPrompter*). Ce fichier est ensuite lu, ligne par ligne, et, conformément à la structure choisie (cf. l'exemple de la figure 6.1), les informations contenues dans chaque groupe de deux lignes sont enregistrées sous la forme d'une association dans le dictionnaire-récepteur. On remarquera que :

- le nom du fichier est conservé dans la variable *nomDuFichier* (il sera réutilisé pour la sauvegarde),

- un flux sur le fichier est ouvert en envoyant le message *pathName:* à la classe *EstdCodeStream*. Cette classe possède des fonctionnalités similaires à la classe *Stream* et ses descendantes de *Smalltalk/V* (cf. chapitre 5).
- des contrôles élémentaires sont effectués sur la structure du fichier : y-a-t-il bien trois mots sur la première ligne de chaque groupe (car la clé de l'association est le troisième mot) ? La deuxième ligne existe-t-elle (si le fichier n'a pas un nombre pair de lignes, il est incorrect) ? En cas d'échec de ces contrôles, le chargement est interrompu et la méthode renvoie *false*, indiquant ainsi que l'opération n'a pas pu être correctement effectuée.

```
chargeDuDisque
  "détermine le nom du fichier de chargement et garnit le dictionnaire récepteur;
   renvoie true si tout s'est bien passé, false si le fichier était de format incorrect"
 |fichier ligne1 ligne2|
 nomDuFichier := CwTextPrompter
                     prompt: 'nom du fichier de chargement ?'
                     answer: 'd:\chap6\analogie.txt'.
 fichier := EstdCodeStream pathName: nomDuFichier.
 "Chaque association est décrite par deux lignes dans le fichier"
 [fichier atEnd] whileFalse:
     [ "première ligne : la clé de l'association est le troisième mot"
       "la méthode subStrings de la classe String en IBM Smalltalk est analogue
       à la méthode asArrayOfSubstrings de la classe String en Smalltalk/V"
       ligne1 := fichier fileStream nextLine subStrings.
       (ligne1 size < 3) ifTrue: [^false].
       fichier atEnd ifTrue: [^false]. "car il manque la deuxième ligne"
       "deuxième ligne : elle contient la chaîne des analogies"
       ligne2 := fichier fileStream nextLine trimBlanks.
       self at: (ligne1 at: 3) put: ligne2].
 ^ true
```

Figure 6.4 : interface avec les fichiers (chargement)

La méthode présentée à la figure 6.5 est la méthode *sauveSurDisque*. Le fichier de sauvegarde est celui indiqué par la variable d'instance *nomDuFichier*. Si celle-ci est encore indéterminée, la méthode l'initialise en demandant à l'utilisateur d'indiquer un nom de fichier[5]. Chaque association est sauvegardée dans ce fichier, sous la forme de deux lignes, conformément à la structure choisie. Le lecteur qui aurait déjà une large pratique de l'utilisation de la méthode *pathName:* de la classe *EstdCodeStream* pourra objecter que, si la taille du fichier diminue entre deux sauvegardes, la réécriture du fichier sera incorrecte car elle laissera subsister, en fin de fichier, des informations de la version précédente. Nous remédierons à cet inconvénient en améliorant la gestion des fichiers de sauvegarde, au paragraphe suivant.

5 On peut en effet imaginer qu'un dictionnaire soit sauvegardé sans avoir été chargé lorsqu'il s'agit d'une première création par l'utilisateur.

```
sauveSurDisque
  "recopie le dictionnaire récepteur dans le fichier désigné par nomDuFichier"
  |sauvegarde "représentera le fichier de sauvegarde" |
  "on vérifie tout d'abord que l'utilisateur a bien choisi un fichier"
 nomDuFichier isNil
   ifTrue:
     [nomDuFichier := CwTextPrompter
                       prompt: 'nom du fichier de sauvegarde ?'
                       answer: 'd:\chap6\analogie.txt'].
 sauvegarde := EstdCodeStream pathName: nomDuFichier.
 self associationsDo:
   [:couple "désignera chaque analogie enregistrée" |
    (sauvegarde fileStream)
        nextPutAll: 'analogies pour ', couple key; cr;
        nextPutAll: '   ', couple value; cr].
 sauvegarde close.
```

Figure 6.5 : interface avec les fichiers (sauvegarde)

Pour l'instant, si l'on fait abstraction de la remarque précédente, on peut vérifier le fonctionnement de ces deux méthodes. Supposons que le fichier traité est le fichier *dico1.txt*, dont l'état initial est représenté sur la partie gauche de la figure 6.6 et évaluons la séquence S :

```
"Séquence S"
 |d| d:= DicoAnalogies new.
     d chargeDuDisque.
     d ajouteAnalogieEntre: 'ire' et: 'colère'.
     d sauveSurDisque
```

A l'issue de l'évaluation, le contenu du fichier *dico1.txt* sera celui qui est représenté sur la partie droite de la figure 6.6.

fichier dico1.txt (état initial)	fichier dico1.txt (état final)

analogies pour amour
  passion tendresse
analogies pour passion
  amour
analogies pour tendresse
  amour
	analogies pour colère
	  ire
	analogies pour amour
	  passion tendresse
	analogies pour tendresse
	  amour
	analogies pour passion
	  amour
	analogies pour ire
	  colère

Figure 6.6 : le fichier *dico1.txt* avant et après l'évaluation de la séquence S

6.2.3 - L'environnement objets facilite la programmation incrémentale

Dans les deux méthodes construites au paragraphe précédent, nous n'avons pas pris en considération toutes les précautions de manipulation possibles lors du chargement ou de la sauvegarde du fichier-dictionnaire. Sans vouloir les traiter ici de manière exhaustive, nous allons montrer sur un exemple comment la programmation orientée objets facilite un enrichissement progressif du développement.

Intéressons-nous à la méthode *sauveSurDisque* et supposons que nous désirions, par mesure de précaution, si le fichier de sauvegarde existe déjà, conserver sa version antérieure dans un fichier de même désignation, mais dont le suffixe sera *bak*. Si nous supposons que le nom du fichier de sauvegarde (indiqué par la variable d'instance *nomDuFichier*) n'a pas, lui-même, le suffixe *bak*, les opérations à effectuer sont les suivantes :

- s'il existe déjà un fichier de même désignation et de suffixe *bak*, détruire ce fichier,
- renommer le fichier de sauvegarde en lui donnant le suffixe *bak*.

On pourra ensuite recopier le contenu du dictionnaire dans le fichier désigné par la variable *nomDuFichier*.

Nous allons supposer que la variable *nomDuFichier* donne toujours le nom complet du fichier c'est-à-dire qu'elle a pour valeur une chaîne de caractères qui indique le chemin d'accès au fichier, la désignation du fichier et son suffixe[6]. Par exemple la valeur *d:\chap6\analogie.txt* indiquera, par le chemin *d:\chap6*, que le fichier est sur le répertoire *chap6* du disque *d:*, qu'il a la désignation *analogie* et que son suffixe est *txt*.

Pour effectuer la modification souhaitée de la méthode *sauveSurDisque*, il suffit d'ajouter, dans cette méthode, après l'établissement de la valeur correcte de la variable d'instance *nomDuFichier*, l'évaluation de :

```
nomDuFichier nomDeFichierChangeEnBak
```

Dans cette expression, la méthode *nomDeFichierChangeEnBak* est une méthode d'instance de la classe *String* qui, adressée à la chaîne représentant le nom du fichier, effectue le changement de suffixe. Bien entendu, cette méthode n'est pas prédéfinie dans l'environnement de Smalltalk mais il est simple de la construire. Nous la présentons à la figure 6.7.

---

6 Ceci suppose que la saisie du nom du fichier, lors de l'évaluation de chargeDuDisque ou sauveSurDisque, assure que cette condition est réalisée. Cela constitue encore une amélioration possible de ces deux méthodes, que nous n'étudierons pas ici.

```
nomDeFichierChangeEnBak   "méthode d'instance de la classe String"
    " suppose que le receveur est un nom de fichier existant décrit de manière complète :
     chemin d'accès, désignation, suffixe
     cette méthode change le suffixe de ce fichier en BAK"
    |acces chemin designation|
    acces := EtFileNamePrompter splitPath:(self asLowercase).
    "acces est un tableau de deux éléments : 1- chemin, 2- nom du fichier"
    chemin := acces at:1.
    designation := (acces at:2) nomDeFichier trimBlanks.
    "si le fichier du suffixe bak existe, il faut le détruire"
    (CfsDirectoryDescriptor
      opendir: chemin
      pattern: (designation,'.bak')
      mode:  FREG) noMatch
       ifFalse:[ CfsFileDescriptor
                   remove:(chemin,'\',designation,'.bak')].
    "il ne reste plus qu'à changer le suffixe en renommant le fichier"
    CfsFileDescriptor
      rename:(chemin,'\',designation,'.txt')
      new: (chemin,'\',designation,'.bak')
```

Figure 6.7 : une nouvelle méthode d'instance de la classe *String*

Commentons maintenant les opérations effectuées par cette méthode.

- La première expression qui sera évaluée enverra à la classe *EtFileNamePrompter* un message *splitPath:* avec, comme argument, la chaîne réceptrice (c'est-à-dire un nom complet de fichier). En réponse, la classe *EtFileNamePrompter* décomposera ce nom en séparant le chemin d'accès et le nom du fichier qu'elle placera respectivement dans les deux éléments d'un tableau.
- Ensuite, les variables *chemin* et *désignation* sont initialisées à partir de ce tableau pour désigner respectivement le nom complet du répertoire et la désignation du fichier (la méthode *nomDeFichier* est décrite dans la suite de ce chapitre).
- L'opération suivante examine le répertoire désigné par la variable *chemin* et supprime, s'il existe sous ce répertoire, le fichier de même désignation que le fichier de départ, mais dont le suffixe est *bak*. Cet examen s'effectue en deux temps. Tout d'abord, en adressant le message *opendir:pattern:mode:* à la classe *CfsDirectoryDescriptor* : le premier paramètre correspond au répertoire de recherche, le deuxième paramètre est le nom du fichier ou du répertoire recherché et le troisième paramètre (*FREG*) indique que l'on recherche un fichier (si l'on avait recherché un repertoire, on aurait passé la constante *FDIR* en troisième paramètre). Le message précédent renvoie un objet de type *CfsDirectoryDescriptor* qui reçoit le message *noMatch* ; ce message renvoie *true* si aucun fichier n'a été trouvé sous le répertoire spécifié.

- La dernière opération envoie enfin un message *rename:new:* la classe *CfsFileDescriptor*. Ce message correspond une méthode de classe qui permet de changer le nom d'un fichier. Ici, seul le suffixe est modifié : il devient *bak*.

Il nous reste indiquer cependant que la méthode *nomDeFichier*, permet partir du nom complet d'un fichier (désignation + suffixe), de récupérer la désignation du fichier. Cette nouvelle méthode de la classe *String* est présentée ci-après :

```
nomDeFichier      "méthode d'instance de String"
  "renvoie la désignation "
   | chaine |
   chaine := self trimBlanks.
   (chaine = (String with: $.) or:
      [chaine = (String with: $. with: $.)])
         ifFalse: [
             chaine := (ReadStream on: chaine)
                 upTo: $..
             chaine size = 0
                 ifTrue: [self error: 'file name missing'].
             chaine size > 8
                 ifTrue: [self error: 'file name too long']].
   ^ chaine
```

Figure 6.8 : la méthode d'instance *nomDeFichier* pour la classe *String*

Les exemples que nous venons de traiter montrent qu'il est facile, avec la programmation orientée objets, d'enrichir un logiciel en programmant par étapes. Le lecteur peut par exemple imaginer, que dans la recherche d'une gestion de fichiers toute épreuve, la prochaine étape serait de s'assurer, dans la méthode *chargeDuDisque*, que le fichier de chargement indiqué par l'utilisateur est bien un fichier existant[7]. Avec ce que nous venons de voir, on peut aisément concevoir comment définir une méthode de classe de la classe *String*, que l'on pourrait appeler *nomDeFichierExistant* et qui, travers un dialogue éventuellement répétitif avec l'utilisateur, renverrait une chaîne représentant le nom d'un fichier existant.

En conclusion de ce paragraphe, nous souhaitons dégager quelques particularités du développement d'applications dans un environnement orienté objets :

- la programmation orientée objets apporte une nouvelle approche incrémentale de la programmation, qu'il ne faut pas confondre avec l'analyse descendante de la programmation classique.

7 En effet, si l'utilisateur frappe par erreur un nom de fichier qui ne correspond pas à un fichier existant, l'évaluation, dans la méthode chargeDuDisque, de :
fichier := EstdCodeStream pathName: nomDuFichier.
renverra l'objet nil.

- Sans exclure une conception descendante, la programmation orientée objetspermet aussi une démarche ascendante qui va *habiller* progressivement une application.
- Le très haut degré de réutilisation des composants (méthodes ou classes) facilite cette programmation incrémentale. Chaque nouveau composant construit devient partie intégrante de l'environnement de développement, ce qui augmente encore les possibilités de réutilisation.
- Les méthodes sont, de par la puissance d'expression des mécanismes de la programmation orientée objets, toujours concises et très lisibles. Le code d'une méthode excède très rarement la vingtaine de lignes.

6.3 - L'interface graphique de l'application

Définissons maintenant les outils qui sont nécessaires à l'application pour permettre à l'utilisateur de consulter un dictionnaire analogique à travers une interface graphique.

6.3.1 - Faire apparaître une fenêtre dédiée à l'application

La plupart des applications sous Windows sont multi-fenêtres, c'est-à-dire qu'elles gèrent plusieurs fenêtres à la fois. Dans de telles applications, on peut distinguer trois types de fenêtres :
- la fenêtre principale, également appelée fenêtre de base ou fenêtre mère, elle représente le cadre de l'application ;
- les fenêtres filles ou fenêtres documents, elles sont toutes rattachées à la fenêtre mère ;
- la fenêtre client ou surface de travail, elle représente l'espace dans lequel sont confinées les fenêtres filles.

En Smalltalk, une fenêtre est un objet et IBM Smalltalk dispose de plusieurs classes dont les instances représentent des objets d'interface (*widgets* dans la terminologie IBM Smalltalk) tels que des fenêtres ou des boîtes-listes. Nous examinerons, à la fin de ce paragraphe, la hiérarchie de ces classes.

Pour l'instant, notons que, si nous souhaitons disposer d'une fenêtre pour l'application, il est nécessaire que cette fenêtre soit associée au dictionnaire : elle sera donc désignée par une variable d'instance de la classe *DicoAnalogies*. Nous sommes donc amenés à modifier la définition de la classe *DicoAnalogies* pour lui ajouter la variable d'instance *fenetre*. La définition de cette classe est la suivante :

```
Dictionary subclass: #DicoAnalogies
  instanceVariableNames:
    'nomDuFichier
     fenetre '
  classVariableNames: ''
  poolDictionaries: ''
```

Il convient également, comme nous l'avons fait remarquer en 6.2.1, de définir la méthode d'instance qui permet d'affecter une valeur cette variable d'instance :

```
fenetre: uneFenetre
  "affecte la valeur uneFenetre à la variable d'instance fenetre"
  fenetre := uneFenetre
```

Nous pouvons maintenant définir la méthode qui fera apparaître la fenêtre l'écran. Cette méthode correspondra au message envoyer au dictionnaire propriétaire (cf. début du paragraphe 6.2) de l'application. En réponse ce message, le dictionnaire créera une fenêtre (qui sera désignée par la variable d'instance *fenetre*) et la fera apparaître l'écran. Une première version, très rudimentaire, de la méthode nécessaire (*ouvrirFenetre*) est présentée la figure 6.9.

```
ouvrirFenetre
  "ouvre une fenêtre pour l'application (version 1)"
  | form |
  fenetre :=
  CwTopLevelShell createApplicationShell:'fenetrePrincipale'
        argBlock:[:w |
        w title:'Utiliser le dictionnaire ', nomDuFichier].
  form := fenetre createForm: 'fenetreTravail' argBlock:nil.
  form manageChild.
  fenetre realizeWidget
```

Figure 6.9 : faire apparaître une fenêtre

Cette méthode crée un objet de classe *CwTopLevelShell*, dans la variable *fenetre*. Un tel objet représente le cadre d'une fenêtre Windows. Celle-ci possède une barre de titre laquelle est associé un menu système, une icône de minimisation et une icône de restauration de taille. Elle ne peut contenir qu'une seule fenêtre fille.

Dans la méthode *ouvrirFenetre*, le cadre de la fenêtre est créé par l'évaluation de

```
fenetre := CwTopLevelShell
  createApplicationShell:'fenetrePrincipale'
     argBlock:[:w |
     w title:'Utiliser le dictionnaire ', nomDuFichier].
```

qui crée l'objet de classe *CwTopLevelShell* sans toutefois le faire apparaître l'écran. Cet objet reçoit ensuite le message *title:* qui fixe l'intitulé qui figurera dans la barre de titre de cette fenêtre. On définit également sous la barre de titre, une fenêtre fille unique, qui sera un objet de la classe *CwForm*, elle est obtenue en envoyant le message *createForm:argBlock:* la variable d'instance *fenetre*. Cette fenêtre est l'espace de travail de l'application, elle pourra elle-même être divisée en plusieurs sous-fenêtres comme nous le verrons dans les paragraphes suivants.

La méthode *manageChild* permet de spécifier que la géométrie de la fenêtre fille est gérée en fonction de la géométrie de la fenêtre mère : toute modification de taille ou de position de la fenêtre mère est répercutée sur la fenêtre fille.

Enfin, la dernière ligne de la méthode *ouvrirFenetre* affiche la fenêtre à l'écran. C'est le message *realizeWidget* qui est envoyé à la fenêtre mère qui déclenche l'affichage de la fenêtre complète. On peut vérifier maintenant le fonctionnement de la méthode en évaluant, avec *Execute*, la séquence :

```
|x| x:= DicoAnalogies new.
x chargeDuDisque.
x ouvrirFenetre
```

6.3.2 - Les différentes classes de fenêtres

Nous venons de voir comment ouvrir une fenêtre à l'écran. Cependant, nous ne pouvons rien faire dans cette fenêtre. Nous ne pouvons même pas introduire du texte sans rapport avec l'application. Avant d'améliorer, dans le paragraphe suivant, le couplage entre l'application et sa fenêtre, présentons brièvement les différentes classes de fenêtres proposées.

Tout composant d'interface ou *widget* est une instance de l'une des sous-classes des classes suivantes :

CwShell Les instances de cette classe ne possèdent qu'une seule fenêtre fille.

CwComposite Les instances de cette classe peuvent gérer plusieurs *Widgets* filles à la fois.

CwPrimitive Les instances de cette classe sont de « simples blocs de construction », elles ne possèdent pas de *Widgets* filles.

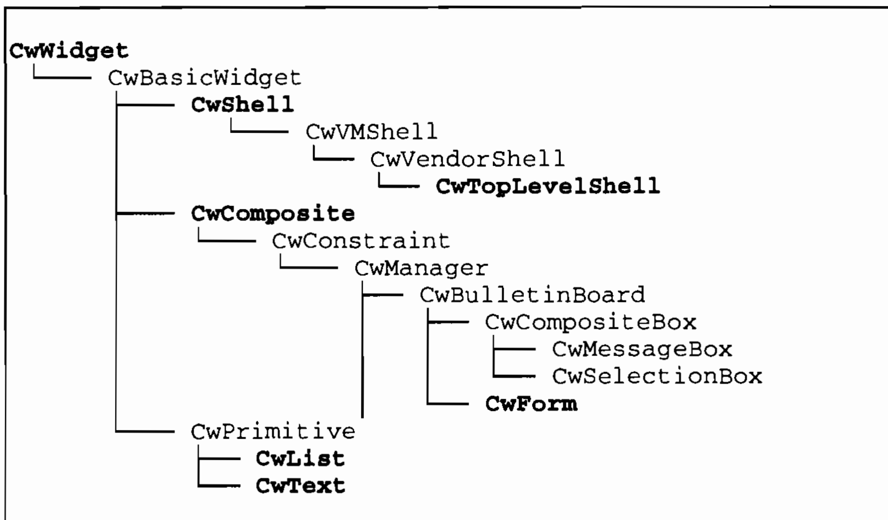


Figure 6.10 : la classe *CwWidget* et des classes descendantes

Avec IBM Smalltalk, implémenter une interface graphique revient élaborer un arbre de *widgets*. Chaque *widget*, sauf celui qui est placé au niveau le plus haut dans la hiérarchie, a un parent. La construction de cet arbre se fait de haut en bas. On met d'abord en place un objet de type *CwShell*, puis on crée un enfant de ce *widget* de type *CwComposite*, enfin on insère dans le *widget* composite un ou plusieurs objets de type *CwComposite* ou *CwPrimitive*.

Par exemple, la fenêtre de la figure 6.11 contient deux panneaux : une boîte-liste gauche et une fenêtre d'édition de texte droite.

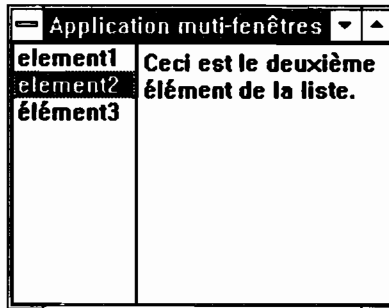


Figure 6.11 : Une application multi-fenêtres

Cette application a été élaborée partir de l'arbre de *widgets* représenté ci-après la figure 6.12 :

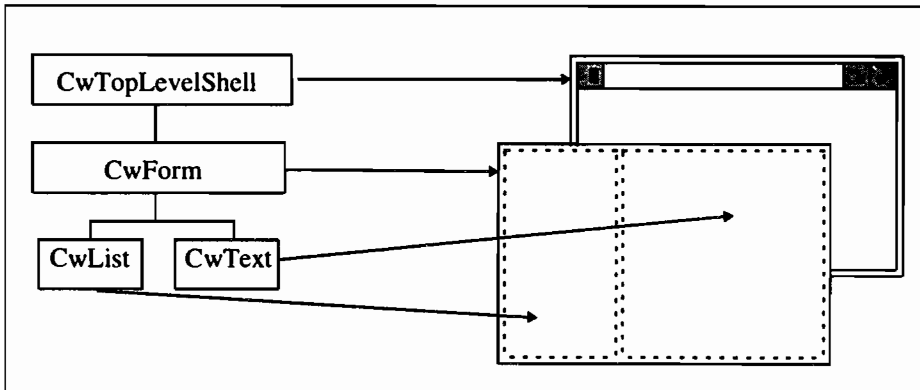


Figure 6.12 : l'arbre de widgets

6.3.3 - Faire apparaître des informations dans la fenêtre

Avec la méthode *ouvrirFenêtre* définie en 6.3.1 (version 1, figure 6.9), la fenêtre ouverte pour l'application ne présentait pas d'informations. Modifions-la pour y faire apparaître les informations du dictionnaire. Cette méthode devient alors :

```
ouvrirFenetre
  "ouvre une fenêtre pour l'application (version 2)"
  | form chaine |
  fenetre :=
  CwTopLevelShell createApplicationShell:'fenetrePrincipale'
         argBlock:[:w |
         w title:'Utiliser le dictionnaire ', nomDuFichier].
  form := fenetre createForm:'fenetreTravail' argBlock:nil.
  form manageChild.
  fenetreTexte := form createText:'panneauTexte'
                argBlock:[: w |
                           w editMode: XmMULTILINEEDIT;
                             width:300; height:120;
                             borderWidth: 1].
  fenetreTexte manageChild.
  chaine := self montreDico.
  fenetreTexte setString: chaine.
  fenetre realizeWidget
```

Par rapport à la version 1, on définit une fenêtre fille de l'objet *form* de type *CwText*, c'est-à-dire un panneau dans lequel on pourra éditer du texte. Pour cela, le message *createText:argBlock:* est envoyé à l'objet *form*. Le premier paramètre indique le nom de la fenêtre texte (*panneauTexte*) et le deuxième paramètre spécifie les caractéristiques de la fenêtre texte (le mode d'édition, la largeur et la hauteur de la fenêtre, la largeur de la bordure de la fenêtre). La fenêtre d'édition est désignée dans une nouvelle variable d'instance de la classe *DicoAnalogies* : *fenetreTexte*.

Puis, l'évaluation de

```
chaine := self montreDico.
```

permet de récupérer dans la variable temporaire *chaine* les informations contenues dans le dictionnaire. Pour afficher ces informations dans la fenêtre d'édition, il suffit d'envoyer le message *setString:chaine* de la classe *CwText* à l'objet *fenetreTexte*. Nous définissons la méthode *montreDico* de la manière suivante :

```
montreDico
  "message envoyé à l'application pour initialiser le contenu du panneau texte
  avec les analogies du dictionnaire "
  |chaine|
  chaine := String new.
  self associationsDo:[ :ass |
                     chaine := chaine, ass key,
                               LineDelimiter, ass value,
                               LineDelimiter].
  ^chaine
```

A présent, si l'on évalue la séquence :

```
|x| x:=DicoAnalogies new.
    x chargeDuDisque.
    x ouvrirFenetre
```

en indiquant, pour le chargement du fichier, le fichier *dico1.txt* dans son état initial, on verra apparaître la fenêtre présentée la figure 6.13.

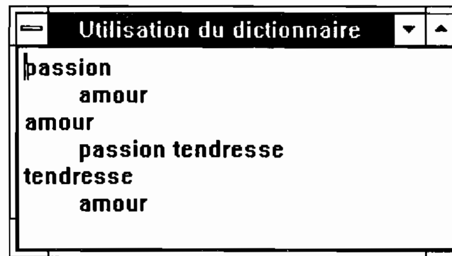


Figure 6.13 : la fenêtre présente le contenu du dictionnaire

6.3.4 - Gérer un menu pour la fenêtre

Pour pouvoir utiliser la fenêtre, il faut lui associer un menu. En l'état actuel de l'application, le panneau *fenetreTexte* ne dispose pas de menu. Etudions maintenant la manière de définir un menu adapté aux besoins de l'application.

L'utilisation d'un menu est réalisée dans une troisième version de la méthode *ouvrirFenetre*, qui est présentée la figure 6.14.

```
ouvrirFenetre
 "ouvre une fenêtre pour l'application (version 3)"
 | form |
 fenetre :=
 CwTopLevelShell createApplicationShell:'fenetrePrincipale'
       argBlock:[:w |
       w title:'Utiliser le dictionnaire ', nomDuFichier].
 form := fenetre createForm: 'fenetreTravail' argBlock:nil.
 form manageChild.
 fenetreText := form createText:'texte'
               argBlock:[: w |
                          w editMode: XmMULTILINEEDIT;
                            width:300; height:120;
                            borderWidth: 1].
 fenetreText manageChild.
 fenetreText setString: self montreDico.
 self openMenu.
 fenetre realizeWidget
```

Figure 6.14 : le panneau *fenetreText* gère un menu

La seule différence, par rapport à l'ancienne version, est l'envoi du message *openMenu* à l'instance de *DicoAnalogies*. Il nous faut donc définir cette nouvelle méthode d'instance de la classe *DicoAnalogies*. L'exécution de *openMenu* associera un objet menu à la fenêtre texte qui apparaîtra à l'écran et présentera une liste d'options parmi lesquelles l'utilisateur devra choisir. La figure 6.15 montre le menu affiché dans la fenêtre ; il se présente sous la forme d'un cadre comprenant une ligne par option.

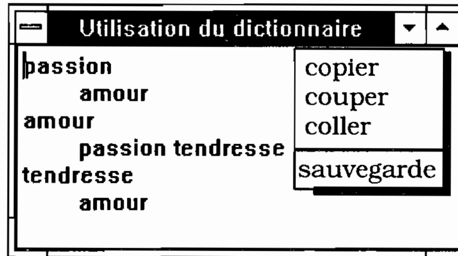


Figure 6.15 : Un menu pour la fenêtre

```
openMenu    "méthode d'instance de la classe DicoAnalogies"
  "associe un menu à la fenêtre fenetreTexte"
  |tableau|
"prise en compte de l'événement déclenché par le clic du bouton droit de la souris"
 fenetreTexte addEventHandler: ButtonMenuMask
              receiver: self
              selector:#button:clientData:event:
              clientData: nil.
 fenetreTexte manageChild.
 "initialisation d'un tableau de types"
 tableau := Array new:5.
 1 to: 3 do: [ :w | tableau at:w put:XmPUSHBUTTON].
 tableau at:4 put:XmSEPARATOR.
 tableau at:5 put:XmPUSHBUTTON.
 "création du menu de la fenêtre fenetreTexte"
 menu := fenetreTexte createSimplePopupMenu:'menu'
                      argBlock:[:w |  w buttons:
 #('copier' 'couper' 'coller' 'separator' 'sauvegarde');
 buttonType: tableau].
 "déclaration des méthodes correspondantes aux options du menu"
 menu addCallback: XmNsimpleCallback
      receiver: self
      selector:#menuItem:clientData:callData:
      clientData: nil.
```

Figure 6.16 : La méthode d'ouverture du menu

La méthode *addEventHandler:receiver:selector:clientData:* associe un gestionnaire d'événements la fenêtre texte. En particulier, l'événement *ButtonMenuMask* qui correspond l'ouverture d'un menu contextuel (clic du bouton droit de la souris), déclenchera l'envoi l'objet *self*, du message *button:clientData:event:* défini ci-après.

```
button:widget clientData:clientData event:event
  "déroule le menu contextuel"
  "teste si le bouton droit de la souris a été pressé"
  event button = 3 ifFalse:[^self].
  "positionne le menu en fonction de l'événement et l'affiche dans la fenêtre"
  menu menuPosition: event;
       manageChild
```

La création du menu contextuel est effectuée par l'envoi du message *createSimplePopupMenu:argBlock:* la fenêtre d'édition. Le premier paramètre indique le nom du menu et le deuxième paramètre spécifie :

- La liste des options du menu sous la forme d'un tableau de chaînes de caractères.
- Les types des options du menu sous la forme d'un tableau de constantes prédéfinies. La constante *XmPUSHBUTTON* définit une option de base du menu. La constante *XmSEPARATOR* définit un filet de séparation qui apparaîtra sous chacune des lignes qui sont indiquées par ce tableau (et sous celles-l seulement). Ces filets permettent ainsi de séparer visuellement des groupes d'options dans un menu. Par exemple, dans le menu affiché par la méthode *openMenu*, un filet est tracé sous la troisième ligne, séparant ainsi les options d'édition de texte de l'option de sauvegarde.

Le menu contextuel est désigné par une nouvelle variable d'instance de la classe *DicoAnalogies* : la variable d'instance *menu*.

Enfin, l'envoi du message *addCallback:receiver:selector:clientData:* la variable *menu* introduit le sélecteur de la méthode qui sera exécutée lors de la sélection d'une option dans le menu : l'événement *XmSimpleCallback* déclenchera l'appel de la méthode d'instance *menuItem:clientData:callData*.

```
menuItem:widget clientData:clientData callData:callData
  "associe des méthodes aux options du menu "
  clientData < 3
    ifTrue:[ fenetreText perform:
                 (#(#copySelection #cutSelection #paste )
                  at: clientData + 1)]
    ifFalse:[ self perform:(#(#sauvegarder) at: 1)].
```

Cette méthode associe des traitements aux options du menu. Le lecteur peut s'étonner de voir que l'on a distingué deux cas avec deux messages *perform:*. Le premier cas correspond aux traitements usuels d'un éditeur de texte ; les méthodes correspondantes (*copySelection*, *cutSelection*, *paste*) sont définies dans la classe *CwText*, elles seront donc adressées l'objet *fenetreTexte*. Le deuxième cas correspond un traitement spécifique de la classe *DicoAnalogies* ; la méthode utilisée (*sauvegarder*) est définie dans la classe *DicoAnalogies*, elle sera donc

appliquée à l'objet receveur *self.* Ici, ces messages ne sont pas envoyés directement aux objets, c'est le message *perform:* qui est utilisé. Le paramètre du message *perform:* est le sélecteur de la méthode à exécuter.

Pour le menu créé par la méthode *menuItem:clientData:callData:*, les options définies et les méthodes correspondantes sont répertoriées dans le tableau 6.17.

option du menu	méthode correspondante	origine et rôle
copier	*copySelection*	méthode prédéfinie de la classe *CwText*, elle correspond à la copie, dans le tampon de transfert, du texte sélectionné dans le panneau.
couper	*cutSelection*	méthode prédéfinie de *CwText* ; elle correspond à la copie de la sélection dans le tampon de transfert puis à la suppression de cette sélection, qui disparaît de l'écran.
coller	*paste*	méthode prédéfinie de *CwText* ; elle correspond au remplacement du curseur ou de la sélection par le texte présent dans le tampon de transfert.
sauvegarder	*sauvegarde*	méthode à définir dans la classe *DicoAnalogies* ; elle correspondra à la sauvegarde, dans un fichier, du texte présenté dans le panneau.

Tableau 6.17 : options et méthodes du menu créé par *menuItem:clientData:callData:*

Une seule méthode reste à définir : la méthode *sauvegarder* de la classe *DicoAnalogies*. Si l'on souhaite pouvoir rapidement expérimenter le menu, on peut par exemple réduire l'action de cette méthode à l'affichage d'une fenêtre qui informera l'utilisateur que l'option reste inopérante :

```
sauvegarder
  "méthode provisoire appelée par l'option 'sauvegarde' du menu
   pour l'instant, on se contente d'indiquer qu'on ne fait rien !"
  CwTextPrompter
     prompt:'Cette option n''est pas encore implementée'
     answer:'Valider pour abandonner'.
```

Nous n'irons pas plus loin dans la définition de cette méthode. En effet, telle qu'elle est présentée ici, l'application est trop rudimentaire :

- l'accès aux informations du dictionnaire n'est pas aisé,
- la grande liberté offerte par les options *copier*, *couper* et *coller* permet à l'utilisateur des opérations qui mettent cn danger la structure du fichier de sauvegarde.

Nous allons maintenant étudier les moyens de structurer la fenêtre en plusieurs panneaux entre lesquels nous définirons des relations de dépendance qui nous permettront de mieux répondre au cahier des charges défini en 6.1.1.

6.4 - Structurer la fenêtre de l'application

Nous pouvons maintenant aborder une nouvelle étape qui nous familiarisera avec la gestion, dans une même fenêtre, de plusieurs panneaux. Nous souhaitons que l'application puisse disposer d'une fenêtre telle que celle de la figure 6.18.

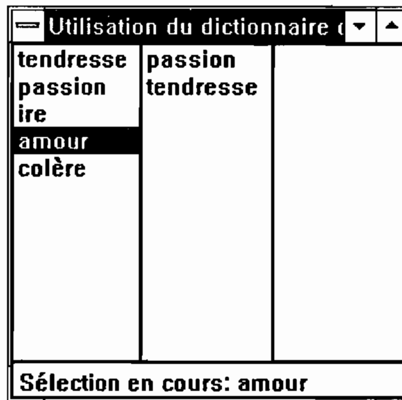


Figure 6.18 : la fenêtre souhaitée pour l'application

Cette fenêtre est formée de quatre panneaux. Trois d'entre eux sont verticaux et adjacents, et apparaissent au dessus du quatrième qui est disposé horizontalement sur toute la largeur de la fenêtre. Les trois panneaux verticaux sont des panneaux de la classe *CwList*, dans lesquels on peut passer en revue une liste d'informations. Une telle liste apparaît, sur la figure 6.18, dans les deux panneaux verticaux les plus à gauche. Le panneau horizontal est de type *CwText* : il est donc de même nature que l'unique panneau de la fenêtre gérée aux paragraphes précédents (6.3.3 et 6.3.4).

Le panneau vertical de gauche présente la liste des mots-clés du dictionnaire. En cliquant dans le panneau sur l'un de ces mots, on fait apparaître, dans le panneau vertical central, la liste des mots proches enregistrés, pour ce mot-clé, dans le dictionnaire. Sur l'exemple de la figure, on a sélectionné à gauche le mot-clé *amour* et on a fait apparaître, dans le panneau central, la liste des deux mots proches *passion* et *tendresse*.

Puisque ce panneau central est, lui aussi, de la classe *CwList*, on pourrait également y sélectionner un des deux mots *passion* et *tendresse* pour faire apparaître en regard, dans le panneau de droite, la liste des mots proches correspondants. L'utilisateur peut ainsi procéder par analogies successives, pour trouver le mot qui lui convient.

Le panneau horizontal, de la classe *CwText*, servira d'aide-mémoire à l'utilisateur qui procède par analogies successives : il indique le point de départ de la recherche et la succession des mots sélectionnés. Sur la figure 6.18, ce panneau indique que la recherche est partie du mot *amour* et qu'elle n'est pas encore allée plus loin. Si, dans le panneau vertical central, on sélectionnait le mot *tendresse*, alors on verrait apparaître à la fois la liste des mots proches de *tendresse* dans le troisième panneau vertical et l'enchaînement *amour tendresse* dans le panneau horizontal[8].

On constate bien que tous les panneaux sont interdépendants dans la mesure où la sélection d'une information dans l'un d'entre eux entraînera la modification de l'affichage d'au moins l'un des autres panneaux. Nous allons maintenant étudier les opérations à mettre en oeuvre pour structurer la fenêtre en plusieurs panneaux, exprimer les dépendances entre les panneaux et attribuer, à chaque panneau, un menu spécifique.

6.4.1 - Une nouvelle classe pour l'application

L'application qui s'appuyait sur la classe *DicoAnalogies* avait surtout un objectif pédagogique : il s'agissait de proposer au lecteur un premier contact avec la mise en oeuvre d'une interface graphique. Nous voulons maintenant construire une application plus réaliste qui va nécessiter la définition de plusieurs variables d'instances pour représenter, à chaque instant, l'état exact de l'application : nom du fichier, fenêtre, information sélectionnée ou présente dans chacun des panneaux. Plutôt que de modifier à nouveau la classe *DicoAnalogies*, en introduisant de nouvelles variables d'instances qui n'ont aucun rapport avec la classe-mère *Dictionary*, nous allons définir, comme support de l'application, la classe *ConsulteDico* dont le dictionnaire sera une variable d'instance. Nous pouvons définir cette classe comme le montre la figure 6.19.

```
Object subclass: #ConsulteDico
 instanceVariableNames:
   'dictionnaire        "désigne le dictionnaire examiné"
    nomDuFichier        "désigne le fichier qui contient le dictionnaire"
    fenetre'            "désigne la fenêtre ouverte pour l'utilisateur"
  classVariableNames: ''
  poolDictionaries: ''
```

Figure 6.19 : la classe *ConsulteDico*, propriétaire de l'application

Cette classe est une sous-classe de la classe *Object*. Est-ce à dire que nous renonçons au travail déjà fait avec la classe *DicoAnalogies* ? Pas du tout : on peut immédiatement ajouter à la nouvelle classe *ConsulteDico* toutes les méthodes de *DicoAnalogies* qui concernaient la mise à jour du dictionnaire, à la seule condition de remplacer, dans ces méthodes, toutes les apparitions du mot *self* (qui re-

8 Ce panneau horizontal n'apportera des informations utiles que si on enchaîne plus de trois analogies successives : alors que les panneaux supérieurs ne conservent que les trois dernières analogies, le panneau horizontal présentera toujours l'enchaînement complet.

présentait le dictionnaire dans *DicoAnalogies*) par le mot *dictionnaire* (qui, dans *ConsulteDico*, représente le même dictionnaire). On peut donc ainsi reporter très facilement, dans la nouvelle classe, les méthodes *ajouteRelationEntre:Et:*, *ajoute:Pour:*, *chargeDuDisque* et *sauveSurDisque*. Nous supposerons, dans la suite de ce chapitre, que ce report a été fait.

Avant d'étudier plus en détail la classe *ConsulteDico*, nous allons présenter l'un des concepts de base de la programmation d'interface graphique sous Smalltalk, le schéma *Model/View/Controller* ou schéma MVC.

6.4.2 - Le schéma Model/View/Controller

En 1970, lorsque démarre à Xerox PARC, en Californie, le projet de recherche pour la conception d'un nouveau mode de dialogue homme / machine, les chercheurs constatent non seulement que les langages et les outils disponibles sont insuffisants, mais aussi que les concepts sont inappropriés. En effet, avec une application classique, le dialogue est simple et l'application dirige les opérations. Elle peut appeler le système d'exploitation, mais inversement elle ne peut pas être appelée par le système d'exploitation. Par contre, avec une application sous interface graphique, la programmation devient événementielle car l'application doit réagir à des événements qu'elle n'a pas provoqués (des événements provenant de l'utilisateur par exemple).

Plutôt que d'utiliser un langage classique, qui nécessite pour dessiner un objet graphique plusieurs centaines d'instructions, les chercheurs imaginent une nouvelle approche de programmation et inventent le langage Smalltalk. De plus, pour gérer la complexité de la programmation événementielle, l'équipe de Xerox PARC organise celle-ci sous forme d'un scénario à trois acteurs que l'on appelle le schéma *Model / View / Controller*. Ce modèle, est encore aujourd'hui, à la base de toutes les interfaces graphiques :

- Le modèle est l'objet responsable de l'application, il dispose des traitements spécifiques de l'application.
- La vue est l'interface à travers laquelle le modèle va gérer le dialogue avec l'utilisateur.
- Le contrôleur est le gestionnaire d'événements (par exemple Windows). Il gère les actions de l'utilisateur (clic de souris, activation d'un menu, etc.), il réagit à ces actions en coordonnant les interactions entre le modèle et la vue.

Le schéma MVC (cf. figure 6.20) permet de séparer les problèmes applicatifs proprement dits des problèmes d'interface. Il est donc bien évident qu'un modèle unique peut être représenté par plusieurs vues, mais celui-ci n'a pas besoin de savoir quelles sont les vues qui lui sont associées. Par contre, les vues doivent connaître précisément le modèle auxquelles elles sont attachées. Il en va de même pour le contrôleur qui doit savoir sur quels modèles et sur quelles vues les actions de l'utilisateur doivent s'exécuter.

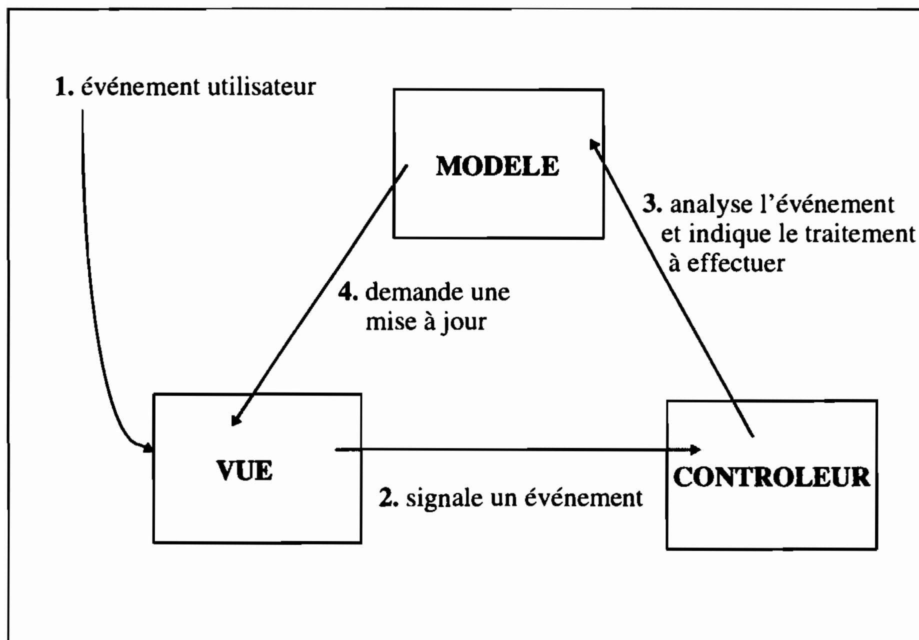


Figure 6.20 : Les liaisons entre Modèle, Vue et Contrôleur

Reprenons maintenant la figure 6.21 pour détailler les variables d'instance de la nouvelle classe. Comme dans la classe *DicoAnalogies*, *nomDuFichier* et *fenetre* désigneront respectivement le nom du fichier de sauvegarde et la fenêtre ouverte pour l'application.

6.4.3 - Créer une instance de la classe *ConsulteDico*

Si nous évaluons la séquence suivante :

|d| d:= ConsulteDico new inspect

nous constatons que, ayant fait appel à la méthode *new* (héritée de *Object*) pour créer une instance de *ConsulteDico*, les variables d'instance ont été créées et qu'elles désignent toutes l'objet *nil*. Pour *nomDuFichier* et *fenetre*, cette initialisation nous convient bien dans la mesure où, tant que l'on n'a pas entamé le dialogue avec l'utilisateur, ces variables doivent rester indéterminées. En revanche, il est souhaitable que, dès la création d'une instance de *ConsulteDico*, la variable d'instance *dictionnaire* prenne la structure d'un objet de la classe *Dictionary*, vide au départ. C'est ce que nous obtenons, en ajoutant à la classe *ConsulteDico* une redéfinition de la méthode de classe *new* qui fait appel à la méthode d'instance privée *initialise* pour donner une première valeur à la variable d'instance *dictionnaire*. Nous représentons ces méthodes à la figure 6.21.

```
new    "méthode de classe redéfinie dans ConsulteDico"
  "crée une instance de la classe ConsulteDico : la variable dictionnaire devient un
dictionnaire
    vide, les autres variables d'instance restent à nil"
  |d| d := super new.
  "la méthode new utilisée ici est celle de Object, pour éviter un appel récursif"
  d initialise.
  ^ d

initialise    "méthode d'instance privée de ConsulteDico appelée par new"
    initialise la variable d'instance dictionnaire comme un dictionnaire vide,
    laisse toutes les autres variables d'instance à nil"
  dictionnaire := Dictionary new
```

Figure 6.21 : les méthodes *new* et *initialise* de la classe *ConsulteDico*

6.4.4 - Partager la fenêtre en plusieurs panneaux

Utilisons la méthode *ouvrirFenetre*, telle qu'elle est définie à la figure 6.23, pour évaluer la séquence :

```
|d| d:= ConsulteDico new.
d chargeDuDisque.
d ouvrirFenetre
```

Nous obtenons la fenêtre représentée à droite du présent texte (figure 6.22). Etudions les nouveautés apportées par cette dernière version de la méthode *ouvrirFenetre*.

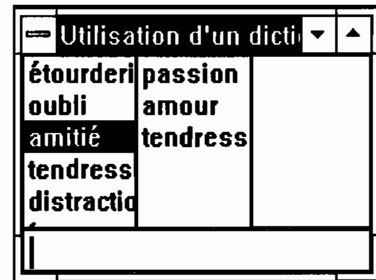


Figure 6.22 : les différents panneaux

Par rapport à la version 3 (cf. figure 6.13), nous définissons quatre panneaux, en envoyant à la fenêtre, représentée par la variable *form*, quatre messages *createList:argBlock:*. Ces panneaux sont des variables d'instance de la classe *ConsulteDico*, ils se nomment respectivement :

- *listeGauche* : panneau vertical de gauche,
- *listeCentrale* : panneau vertical central,
- *listeDroite* : panneau vertical de droite,
- *listeBas* : panneau horizontal du bas de la fenêtre.

```
ouvrirFenetre
  "ouvre une fenêtre pour l'application (version 4)"
  [...] "cf. figure 6.10 pour le début du code de la méthode"
  listeGauche := form createList:'liste1' argBlock:[ : w |
      w leftAttachment: XmATTACHFORM;
        rightAttachment: XmATTACHPOSITION;
        topAttachment: XmATTACHFORM ;
        bottomAttachment: XmATTACHPOSITION;
        rightPosition: 33; bottomPosition: 80;
        selectionPolicy: XmSINGLESELECT;
        items: self initialiseListeGauche].
  listeGauche manageChild.
  listeGauche addCallback: XmNsingleSelectionCallback
              receiver: self
              selector: #selectListe1:clientData:callData:
              clientData: nil.
  listeCentrale := form createList:'liste2' argBlock:[ : w |
       w leftAttachment: XmATTACHPOSITION;
         rightAttachment: XmATTACHPOSITION;
         leftPosition: 33;rightPosition: 66;
         topAttachment: XmATTACHFORM;
         bottomAttachment: XmATTACHPOSITION;
         bottomPosition: 80;
         selectionPolicy: XmSINGLESELECT].
  listeCentrale manageChild.
  listeCentrale addCallback: XmNsingleSelectionCallback
                receiver: self
                selector: #selectListe2:clientData:callData:
                clientData: nil.
  listeDroite := form createList:'list3' argBlock:[ : w |
       w leftAttachment: XmATTACHPOSITION;
         leftPosition: 66; bottomPosition: 80;
         rightAttachment: XmATTACHFORM;
         topAttachment: XmATTACHFORM ;
         bottomAttachment: XmATTACHPOSITION;
         selectionPolicy: XmSINGLESELECT].
  listeDroite manageChild.
  listeDroite addCallback: XmNsingleSelectionCallback
              receiver: self
              selector: #selectListe3:clientData:callData:
              clientData: nil.
  listeBas := form createText:'texte' argBlock:[ : w |
       w editMode: XmMULTILINEEDIT;
          width: 300; topPosition: 80;
          leftAttachment: XmATTACHFORM;
          rightAttachment:XmATTACHFORM;
          topAttachment: XmATTACHPOSITION;
                  bottomAttachment: XmATTACHFORM ].
  fenetre realizeWidget.
```

Figure 6.23 : la méthode *ouvrirFenetre* crée plusieurs panneaux

Pour répartir les panneaux dans la fenêtre, plusieurs méthodes héritées de la classe *CwWidget*, sont utilisées :

- `leftAttachment:XmATTACHFORM` indique que le côté gauche de la liste adhère à la bordure gauche de la fenetre mère
- `rightAttachment:XmATTACHPOSITION` indique que le côté droit de la liste adhère au coté gauche de la liste centrale
- `rightPosition:33` indique que la largeur de la liste correspond à 33% de la largeur de la fenêtre mère.
- `topAttachment:XmATTACHFORM` indique que le côté supérieur de la liste adhère à la bordure supérieure de la fenetre mère
- `bottomAttachment:XmATTACHPOSITION` indique que le côté inférieur de la liste adhère au coté supérieur du panneau du bas
- `bottomPosition:90` indique que la hauteur de la liste correspond à 90% de la hauteur de la fenêtre mère.
- `selectionPolicy: XmSINGLESELECT` indique que le panneau *listeGauche* est une boîte-liste à sélection simple, c'est à dire une boîte-liste dans laquelle on ne peut sélectionner qu'un élément.

Ainsi, le coin supérieur du panneau *listeGauche* coïncidera avec le coin supérieur gauche de la fenêtre, ce panneau occupera le tiers de la largeur et les neuf dixièmes de la hauteur de la fenêtre. Cela correspond bien aux proportions de ce panneau dans la figure 6.22.

Intéressons-nous maintenant aux contenus des panneaux. Lors de l'ouverture de la fenêtre, le panneau *listeGauche* contient la liste des analogies du dictionnaire, les panneaux *listeCentrale* et *listeDroite* sont vides. L'initialisation du contenu du panneau de gauche est effectuée dans la méthode *ouvrirFenetre* par le message *items:* de la classe *CwList*. Ce message prend en paramètre un tableau de chaînes de caractères. C'est bien ce que renvoie la méthode *initialiseListeGauche* de la classe *ConsulteDico* (cf. ci-après), qui obtient d'abord, avec la méthode *keys* de la classe *Dictionary*, une instance de la classe *Set* répertoriant tous les mots-clés du dictionnaire consulté puis envoie à cet objet le message *asArray* pour obtenir un tableau (chaque élément du tableau correspond à un élément de la liste).

```
initialiseListeGauche
  "message envoyé au panneau de gauche pour initialiser
   son contenu avec la liste des mots-clés du dictionnaire"
  ^ dictionnaire keys asArray
```

Pour les panneaux *listeCentrale* et *listeDroite*, le contenu dépendra du mot-clé qui sera sélectionné par l'utilisateur dans le panneau précédent, leur initialisation n'est donc pas effectuée dans la méthode *ouvrirFenetre*. Nous verrons par la suite, comment définir des méthodes d'initialisation de ces panneaux.

6.4.5 - Synchroniser les panneaux

Il nous reste maintenant à synchroniser les panneaux, c'est-à-dire à spécifier les enchaînements d'opérations provoqués par les actions de l'utilisateur. Si, par exemple, celui-ci effectue la sélection d'un mot dans le panneau *listeGauche*, on

doit voir immédiatement apparaître les mots proches correspondants dans le panneau *listeCentrale*. Il faut donc que le fait de cliquer sur un mot du panneau *listeGauche* déclenche l'exécution d'une méthode qui fasse apparaître dans le panneau *listeCentrale*, les analogies correspondant au mot sélectionné. Comment définir cette synchronisation entre les deux panneaux ?
L'envoi du message *addCallback:e receiver:self selector:#m clientData:nil* à un panneau indique le sélecteur de la méthode *m* à exécuter quand le panneau est affecté par l'événement *e*.
Si nous revenons à la méthode *ouvrirFenetre* de la figure 6.23, nous constatons que pour le panneau gauche, l'événement correspondant à la sélection d'un élément (*XmNsingleSelectionCallback*) déclenche l'exécution de la méthode *selectListe1:clientData:callData:*. Examinons une manière de définir cette méthode :

```
selectListe1:widget clientData:clientD callData:callD
  "cette méthode est activée à chaque changement de sélection du panneau listeGauche.
   Elle déclenche la mise à jour du panneau listeCentrale"
  "widget  représente le panneau-liste dans lequel on a sélectionné un élément
   clientData  est un objet de type CwListCallBackData qui décrit l'événement
   callData  représente des données complémentaires d'appel qui ne sont pas utilisées ici"
  self panneauCentral items:
  (dictionnaire at:(widget selectedItems at:1)) subStrings
```

Le contenu du panneau *listeCentrale* dépendra du mot-clé qui sera sélectionné par l'utilisateur dans le panneau précédent. Ce mot-clé est accessible par la méthode *selectedItems* de la classe *CwList*, qui renvoie une liste ordonnée des éléments sélectionnés dans le panneau. Le contenu du panneau est obtenu à partir du dictionnaire consulté (méthode *at:*) puis transformée en un tableau de mots.
Le raisonnement effectué pour la synchronisation des panneaux *listeGauche* et *listeCentrale*, est également appliqué aux panneaux *listeCentrale* et *listeDroite* ainsi qu'aux panneaux *listeDroite* et *listeGauche*.

6.4.6 - Améliorer la synchronisation

En l'état actuel du développement de notre application, il manque à notre synchronisation la mise à jour du panneau *listeBas* dont le contenu doit, rappelons-le, indiquer la chaîne des mots successivement sélectionnés.

Nous allons, dans le paragraphe suivant, finaliser l'application, en assurant cette synchronisation et en gérant un nombre non limité de sélections consécutives.

6.5 - L'application finale

Examinons la figure 6.24. Elle représente une étape du déroulement de l'application, que l'on peut reconstituer en examinant le panneau du bas :

- L'utilisateur a sélectionné au départ le mot *ire*, puis dans les mots proches de ce premier mot, le mot *colère*.

- Parmi les mots proches de *colère*, il a sélectionné le mot *fureur*.
- Enfin, parmi les mots proches de *fureur*, il a choisi le mot *rage*.
- Il est alors « revenu en arrière » d'un mot-clé, sans modifier les sélections et les trois panneaux verticaux montrent respectivement la deuxième, troisième et quatrième sélection d'une liste de quatre sélections présentée dans le panneau du bas.

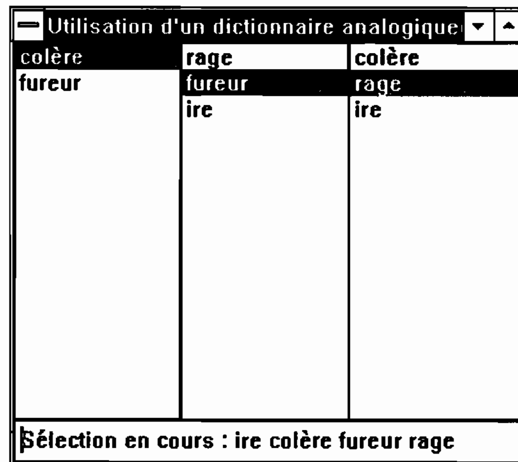


Figure 6.24 : une étape du déroulement de l'application

Cet exemple nous montre que l'application doit gérer non pas trois sélections successives (dans les trois panneaux verticaux) mais une suite variable de sélections, dont les trois panneaux verticaux présenteront, à chaque instant trois éléments contigus. Le panneau du bas indiquera, lui, l'état complet de cette suite. Cette suite est variable car toute nouvelle sélection la modifie : par exemple si, partant de l'état représenté à la figure 6.24, on sélectionne dans le panneau vertical central le mot *ire*, les sélections *fureur* et *rage* (de rangs 3 et 4) disparaissent et la fenêtre se modifie de la manière suivante :

- le panneau gauche conserve la même liste de mots ;
- le contenu du panneau central conserve la même liste de mots mais la sélection passe de *fureur* à *ire* ;
- le panneau droit est complètement remis à jour, pour afficher les mots proches de *ire*, parmi lesquels aucune sélection n'est encore effectuée ;
- le panneau du bas indique la suite de sélections : *ire colère ire*.

Pour prendre en considération ces contraintes de l'application, il nous faut définir de nouvelles variables d'instances pour la classe *ConsulteDico*.

6.5.1 - Les nouvelles variables d'instance de la classe *ConsulteDico*

La figure 6.25 présente une redéfinition de la classe *ConsulteDico*. Par rapport à la précédente définition de la figure 6.19, nous pouvons remarquer :

- que les variables *dictionnaire*, *nomDuFichier* et *fenetre* sont toujours présentes ;
- que la variable *indexSelectionGauche* n'est pas la seule nouvelle variable. Cinq autres variables sont définies : elles se nomment *suiteDeSuivi*, *listeGauche*, *listeCentrale*, *listeDroite* et *listeBas*. Nous décrirons également leur utilisation.

```
Object variableSubclass: #ConsulteDico
  instanceVariableNames:
    'dictionnaire nomDuFichier fenetre
     suiteDeSuivi          "liste des sélections en cours"
     indexSelectionGauche  "index de la sélection du panneau gauche dans
                            suiteDeSuivi"
     listeGauche           "désigne le panneau vertical gauche"
     listeCentrale         "désigne le panneau vertical central"
     listeDroite           "désigne le panneau vertical droit"
     listeBas              "désigne le panneau horizontal bas"
  classVariableNames: ''
  poolDictionaries: ''
```

Figure 6.25 : la nouvelle définition de la classe *ConsulteDico*

La variable *suiteDeSuivi* désignera, à chaque instant, la suite des sélections effectuées. Nous lui donnerons la forme d'une instance de la classe *OrderedCollection* et nous conviendrons que, relativement à cette suite, la variable d'instance *indexSelectionGauche* désignera toujours le rang de l'élément qui correspond à la sélection du panneau *listeGauche*. Ainsi, la sélection du panneau *listeCentrale* sera obtenue en consultant, dans *suiteDeSuivi*, l'élément de rang *indexSelectionGauche+1* et celle de *listeDroite* correspondra au rang *indexSelectionGauche+2*.

C'est par abus de langage que nous avons indiqué que les éléments de *suiteDeSuivi* correspondent à des sélections. Ils sont, en fait, associés à des panneaux (visibles ou non à l'écran) dont chacun peut être dans l'un des trois états suivants :

1 panneau vide,
2 panneau contenant une liste de mots dont aucun n'est sélectionné,
3 panneau contenant une liste de mots, dont l'un a été sélectionné et doit apparaître, s'il est visible, en inversion vidéo.

Nous conviendrons, dans ces conditions, que tout élément de *suiteDeSuivi* indique, par sa valeur si le panneau qu'il représente contient ou non une sélection effective :

valeur *nil* : le panneau est dans l'état 1 ou l'état 2,

valeur *unMot* : le panneau est dans l'état 3 et le mot *unMot* y est sélectionné.

Conformément à ces conventions, nous devons redéfinir la méthode *initialise* qui sert à préparer une instance de *ConsulteDico* nouvellement créée. Nous le faisons de la manière suivante :

```
initialise
  "méthode privée, appelée par new ou lors du chargement d'un nouveau fichier;
   initialise la variable d'instance dictionnaire comme un dictionnaire vide;
   initialise suiteDeSuivi pour indiquer que les panneaux ne comporteront, au départ,
   aucune sélection;
   initialise indexSelectionGauche pour indiquer que le premier élément de suiteDeSuivi
   désigne le panneau gauche;
   laisse toutes les autres variables d'instance à nil"
 dictionnaire := Dictionary new.
 suiteDeSuivi := OrderedCollection
                 with: nil with: nil with: nil.
 indexSelectionGauche := 1
```

En effet, quand on ouvre la fenêtre, le panneau de gauche contiendra la liste de toutes les clés du dictionnaire, sans aucune sélection. La variable d'instance *indexSelectionGauche* doit donc désigner le premier élément de *suiteDeSuivi* et cet élément a la valeur *nil* (aucune sélection). La variable *suiteDeSuivi* a deux éléments supplémentaires qui correspondent aux panneaux central et gauche, vides au départ. Ces deux éléments ont donc aussi la valeur *nil*.

6.5.2 - Des variables d'instance pour désigner les panneaux

La figure 6.25 introduit également quatre nouvelles variables d'instance : *listeGauche*, *listeCentrale*, *listeDroite* et *listeBas*, que nous n'avons pas encore présentées. Ces variables vont désigner chacun des quatre panneaux et elles seront initialisées au moment de l'ouverture de la fenêtre. En effet, pour assurer la synchronisation entre les panneaux, il nous faudra adresser directement des messages à ces derniers. Il est donc nécessaire que l'objet propriétaire de l'application « connaisse » chacun des panneaux de la fenêtre.

A propos de la fenêtre, remarquons que, jusqu'ici, nous n'avons pas utilisé la variable d'instance *fenetre* qui la désigne. Nous le ferons pour pouvoir modifier le libellé de la barre de titre, à chaque changement de fichier, au cours du déroulement de l'application. La figure 6.26 montre comment on peut procéder. Sur cette figure, on trouve d'abord le début des nouvelles versions des méthodes *chargeDuDisque* et *sauveSurDisque* (déjà étudiées en 6.2.2). On constate que chacune d'entre elles fait appel à la méthode *nomDuFichier:* qui permet d'affecter une valeur à la variable d'instance *nomDuFichier* (cf. 6.2.1). Comme le nom du fichier est affiché, dans la barre de titre de la fenêtre, il convient, à chaque exécution de la méthode *nomDuFichier:*, de mettre à jour la fenêtre. C'est ce que fait la méthode *nomDuFichier:*, présentée au bas de la figure 6.26, en adressant à la variable d'instance *fenetre* successivement deux messages:

- le message *title:* initialise le libellé de la barre de titre,
- le message *updateTitle* provoque la mise à jour de la barre de titre à l'écran.

```
chargeDuDisque
  "détermine le nom du fichier de chargement et garnit la variable d'instance dictionnaire
   renvoie true si tout s'est bien passé, false si le fichier était de format incorrect"
 |fichier ligne1 ligne2|
 self nomDuFichier:
    (CwTextPrompter prompt: 'nom du fichier de chargement ?'
                    answer: 'd:\chap6\sentimts.txt').
 ... "la suite de la méthode n'est pas décrite ici"

sauveSurDisque
  "recopie le dictionnaire dans le fichier désigné par nomDuFichier"
 |sauvegarde "représentera le fichier de sauvegarde" |
 "on demande d'abord à l'utilisateur d'indiquer ou de confirmer le nom du
  fichier de sauvegarde"
 self nomDuFichier:
    (CwTextPrompter prompt: 'nom du fichier pour la sauvegarde ?'
                    default: nomDuFichier).
 ... "la suite de la méthode n'est pas décrite ici"

nomDuFichier: uneChaine
  "affecte uneChaine à la variable d'instance nomDuFichier
   change la barre de titre de la fenêtre pour indiquer le nouveau fichier"
 nomDuFichier := uneChaine.
 fenetre notNil    "si on a ouvert une fenêtre pour l'application"
    ifTrue: [fenetre title: 'Utilisation du dictionnaire '
                             ,nomDuFichier;
                     updateTitle]
```

Figure 6.26 : la gestion des noms de fichiers

6.5.3 - La version finale de la méthode *ouvrirFenetre*

Dans la version finale de la méthode *ouvrirFenetre*, on remarquera que chaque panneau créé est affecté à la variable d'instance qui lui correspond (*listeGauche*, *listeCentrale*, *listeDroite* et *listeBas*).

On peut constater également que chaque panneau, sauf le panneau du bas, indique une méthode pour la synchronisation (*selectListe1.clientData:callData:*, *selectListe2:clientData:callData:*, *selectListe3:clientData:callData:*) ainsi qu'un menu. Les trois menus sont stockés dans trois nouvelles variables d'instances de la classe *DicoAnalogies* (*menuGauche*, *menuCentral*, *menuDroit*).

Nous n'étudierons pas ici la totalité des méthodes mises en oeuvre pour cette version finale de l'application. Nous nous contenterons ici de donner deux exemples, en présentant les opérations de synchronisation associées à un changement global dans le panneau droit, puis la gestion d'un menu. La totalité du code de l'application est présentée en annexe.

```
ouvrirFenetre
  "ouvre une fenêtre pour l'application (version 5)"
  | tableau |
  [...] "le début de la méthode n'est pas décrit ici"
 listeGauche addCallback: XmNsingleSelectionCallback
             receiver: self
             selector: #selectListe1:clientData:callData:
             clientData: nil.
 listeGauche addEventHandler: ButtonMenuMask
             receiver:self
             selector: #buttonGauche:clientData:event:
             clientData:nil.
 listeGauche manageChild.
[...]
tableau := Array new:8.
 1 to: 2 do: [ : w | tableau at:w put:XmPUSHBUTTON].
 tableau at:3 put:XmSEPARATOR.
  4 to: 5 do: [ : w | tableau at:w put:XmPUSHBUTTON].
 tableau at:6 put:XmSEPARATOR.
 7 to: 8 do: [ : w | tableau at:w put:XmPUSHBUTTON].
 menuGauche := listeGauche createSimplePopupMenu: 'menuG'
                            argBlock:[ : w |
        w buttons: #('Charger' 'Sauvegarder' 'separator'
                     'Ajouter' 'Modifier' 'separator'
                     '<--' '-->');
          buttonTypes: tableau].
 menuGauche addCallback: XmNsimpleCallback
            receiver: self
            selector: #menuGaucheItem:clientData:callData:
            clientData: nil.
[...] "la suite de la méthode n'est pas décrite ici"
```

Figure 6.27 : dernière version de la méthode *ouvrirFenetre*

6.5.4 - Effet d'une sélection dans le panneau droit

Quand on sélectionne un mot dans le panneau droit, on veut faire apparaître la liste des mots proches correspondant à cette sélection. Conformément à l'esprit de l'application, cette liste doit apparaître à droite de la sélection. Or, le panneau droit étant le dernier des trois panneaux verticaux, il est nécessaire de faire « glisser » vers la gauche les informations des panneaux selon le principe suivant :

- l'ancien affichage du panneau gauche disparaît ;
- l'ancien affichage du panneau central apparaît dans le panneau gauche ;
- l'ancien affichage du panneau droit apparaît dans le panneau central ;
- le panneau droit montre la liste des mots proches correspondant à la sélection qui venait d'être effectuée dans ce même panneau ;

- le panneau du bas complète l'affichage de la liste des sélections en cours avec le dernier mot sélectionné.

On constate donc que l'effet d'une sélection dans le panneau droit doit entraîner une modification de tous les autres panneaux. La méthode *changeDroit:* sera chargée d'assurer les modifications. Cette méthode est présentée à la figure 6.28.

```
changeDroit: motSelectionne
  "méthode exécutée avec un changement global dans le panneau droit"
 suiteDeSuivi at: (indexSelectionGauche + 2)
              put: motSelectionne.
  "on supprime les éléments de la liste qui sont au delà du panneau droit"
 [(suiteDeSuivi size) > (indexSelectionGauche + 2)]
     whileTrue: [suiteDeSuivi removeLast].
  "on ajoute un élément qui représentera le nouveau panneau droit après le décalage vers
   la gauche"
 suiteDeSuivi add: nil.
  "on fait maintenant glisser les sélections vers la gauche :
   l'ancien panneau central deviendra le nouveau panneau gauche"
 indexSelectionGauche := indexSelectionGauche + 1.
 self ajustePanneaux
```

Figure 6.28 : la méthode exécutée après une sélection dans le panneau droit

Commentons les différentes étapes de l'exécution de la méthode *changeDroit:*.

- Le mot correspondant à la sélection dans le panneau droit est transmis en argument à cette méthode. La première opération consiste à enregistrer ce mot dans l'élément de *suiteDeSuivi* qui correspond au panneau droit (élément dont l'index est *indexSelectionGauche+2*).
- Ensuite, il faut éventuellement tronquer *suiteDeSuivi*. En effet, si le panneau droit ne représentait pas le dernier élément de *listeDeSuivi* (cf. exemple de la figure 6.24), la sélection doit annuler toutes les sélections antérieures au delà du panneau droit.
- La prochaine opération ajoute un nouvel élément à la fin de *suiteDeSuivi* pour représenter le nouveau panneau qui doit apparaître à droite de la sélection.
- On fait alors glisser les sélections en augmentant d'une unité la valeur de *indexSelectionGauche*.
- Il reste à mettre à jour tous les panneaux. Cette opération s'effectue en adressant à l'objet propriétaire de l'application le message *ajustePanneaux*, qui correspond à la méthode décrite à la figure 6.29. Cette méthode déclenchera, bien entendu, les méthodes d'initialisation / mise à jour de chaque panneau : *listeGauche*, *listeCentrale*, *listeDroite* et *listeBas* mais, elle devra également, pour chacun des panneaux verticaux, rétablir si nécessaire la dernière sélection.

```
ajustePanneaux
  "met à jour tous les panneaux en conservant les sélections existantes"
  "mise à jour du panneau gauche"
  self panneauGauche.
  self remetSelection: indexSelectionGauche
       pour: listeGauche.
  "mise à jour du panneau central
 self panneauCentral.
  self remetSelection:(indexSelectionGauche + 1)
       pour:listeCentrale.
  "mise à jour du panneau droit"
  self panneauDroit.
  self remetSelection: (indexSelectionGauche + 2)
       pour:listeDroite.
  "mise à jour du panneau du bas"
  self panneauBas
```

Figure 6.29 : une méthode pour remettre à jour tous les panneaux

Etudions par exemple les opérations effectuées par la méthode de la figure 6.29 pour le panneau gauche. Pour ce panneau, cette méthode évalue :

```
self panneauGauche.
self remetSelection: indexSelectionGauche
     pour: listeGauche.
```

La méthode *panneauGauche* correspond au protocole suivant :

```
panneauGauche
  "message envoyé au panneau gauche pour initialiser ou mettre à jour son contenu"
  (self contenuCorrespondantA:indexSelectionGauche) notNil
    ifTrue:[ listeGauche items:
       (self contenuCorrespondantA:indexSelectionGauche)]
```

dans lequel la méthode *contenuCorrespondantA:* est une méthode qui sera utilisée par chacun des panneaux verticaux en fonction de l'argument transmis (de *indexSelectionGauche* à *indexSelectionGauche*+2 pour les trois panneaux). La méthode *contenuCorrespondantA:* est présentée à la figure 6.30. Nous laisserons au lecteur le soin de l'étudier, en s'aidant des commentaires qui l'accompagnent.

Après avoir mis à jour le panneau gauche, il nous reste à rétablir, si nécessaire la sélection qui figurait dans la liste qui vient d'être affichée. Une sélection est à rétablir, si l'élément correspondant au panneau dans *suiteDeSuivi* n'a pas la valeur *nil*. Dans ce cas, la valeur de cet élément est le mot qui doit apparaître dans le panneau comme étant sélectionné. Pour rétablir une telle sélection, il est nécessaire d'adresser, au panneau lui-même, le message *selectItem:notify:* qui correspond à une méthode d'instance de la classe *CwList*.

```
contenuCorrespondantA: indexSelection
  " renvoie, pour mettre à jour un panneau, le contenu correspondant
   à l'élément désigné par indexSelection dans suiteDeSuivi,
   ne met pas à jour la sélection du panneau "
 |selectionPrecedente|
 " si le panneau à remplir n'est pas le premier, selectionPrecedente
   désignera la sélection dans le panneau à gauche du panneau à remplir "
 indexSelection = 1
   ifTrue: [ " c'est le panneau le plus à gauche : on renvoie le dictionnaire entier "
             ^ self initialiseListeGauche]
   ifFalse: [ " s'il n'y a aucune sélection à gauche, le panneau doit rester vide "
              selectionPrecedente :=
                   suiteDeSuivi at: (indexSelection - 1).
              selectionPrecedente = nil
                 ifTrue: [^ nil]
                 ifFalse:
                   [^ (dictionnaire at: selectionPrecedente)
                      subStrings]]
```

Figure 6.30 : déterminer les informations à afficher dans un panneau

Dans la figure 6.29, pour le panneau gauche, l'opération qui rétablit la sélection est effectuée avec l'évaluation de :

```
self remetSelection: indexSelectionGauche
     pour: listeGauche
```

et la méthode *remetSelection:pour:* correspond à :

```
remetSelection: indexSelection pour: unPanneau
  " remet, dans le panneau unPanneau, la sélection indiquée par indexSelection "
  |cle|
  cle := suiteDeSuivi at: indexSelection.
  cle notNil
    ifTrue: [unPanneau selectItem:cle notify:false]
```

6.5.5 - Le menu des panneaux verticaux

Dans chaque panneau vertical le même menu est disponible (cf. figure 6.31). Les deux premières options permettent la gestion de fichier et déclenchent les méthodes *chargeDuDisque* et *sauveSurDisque*, que nous avons déjà étudiées. Les trois options suivantes correspondent à la mise à jour du dictionnaire et permettent d'ajouter, de supprimer ou de modifier des analogies. Nous ne les décrirons pas ici.

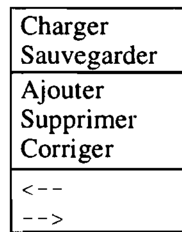


Figure 6.31 : le menu

Nous étudierons seulement les deux dernières options qui permettent le glissement des informations vers la gauche (option <--) ou bien vers la droite (option -->). Elles correspondent respectivement aux méthodes *allerAGauche* et *allerADroite* de la figure 6.32.

```
allerAGauche
  "cette méthode permet de revenir en arrière d'un panneau  "
 indexSelectionGauche = 1
   ifTrue: [CwMessagePrompter warn:'Impossible'. ^nil].
 indexSelectionGauche := indexSelectionGauche - 1.
 self ajustePanneaux

allerADroite
  "cette méthode permet d'avancer d'un panneau vers la droite"
 ((indexSelectionGauche +2) = suiteDeSuivi size)
    ifTrue: [CwMessagePrompter warn:'Impossible'. ^nil].
 indexSelectionGauche := indexSelectionGauche + 1.
 self ajustePanneaux
```

Figure 6.32 : les méthodes pour le « gissement » des informations

La méthode *allerAGauche* vérifie que l'opération demandée est possible : si le panneau de gauche (désigné dans *suiteDeSuivi* par *indexSelectionGauche*) affiche déjà les informations correspondant au premier élément de *suiteDeSuivi*, il est impossible de glisser vers la gauche. Si le glissement est possible, il est obtenu simplement, en diminuant d'une unité la valeur de *indexSelectionGauche* puis en envoyant à l'objet propriétaire de l'application le message *ajustePanneaux*.

De même, la méthode *allerADroite* n'autorisera le glissement que si le panneau de droite ne correspond pas encore au dernier élément de *suiteDeSuivi*. A l'inverse de *allerAGauche*, pour effectuer le glissement, elle augmentera d'une unité la valeur de *indexSelectionGauche*. Et, bien entendu, elle enverra également, à l'objet propriétaire de l'application, le message *ajustePanneaux*.

Annexe - utilisation d'un dictionnaire analogique : l'application

Nous présentons ici l'implémentation complète proposée pour l'application étudiée au chapitre 6. La définition de la classe et les méthodes sont celles qui ont été obtenues avec l'option *file out* du « Classes Browser », qui produit, avec IBM Smalltalk, le fichier *cnsltdco.st*. Toutefois, les méthodes ne sont pas présentées ici dans l'ordre alphabétique. L'ordre que nous avons choisi est celui qui nous semble permettre au lecteur une meilleure compréhension des mécanismes de l'application.

Définition de la classe et création d'une instance

```
Object subclass: #ConsulteDico
    instanceVariableNames: '
      dictionnaire          "désigne le dictionnaire examiné"
      nomDuFichier          "nom du fichier contenant le dictionnaire examiné"
      fenetre               "désigne la fenêtre ouverte pour l'utilisateur"
      suiteDeSuivi          "liste des sélections en cours"
      indexSelectionGauche  "index de la sélection du panneau gauche dans suiteDeSuivi"
      listeGauche           "désigne le panneau vertical gauche"
      listeCentrale         "désigne le panneau vertical central"
      listeDroite           "désigne le panneau vertical droit"
      listeBas              "désigne le panneau du bas"
      menuGauche            "désigne le menu du panneau vertical gauche"
      menuCentral           "désigne le menu du panneau vertical central"
      menuDroit '           "désigne le menu du panneau vertical droit"
      classVariableNames: ''
      poolDictionaries: 'CwConstants CldtConstants'

new  "new est la seule méthode de classe définie pour ConsulteDico"
    "crée une instance de la classe ConsulteDico
    les variables d'instance sont initialisées par appel de la méthode d'instance initialise"
    |d| d := super new.
    "la méthode new utilisée ici est celle de Object, pour éviter un appel récursif"
    d initialise.
    ^ d
```

initialise

```
    "méthode privée, appelée par new et à chaque chargement d'un nouveau fichier;
    initialise la variable d'instance dictionnaire comme un dictionnaire vide;
    initialise suiteDeSuivi pour indiquer que les panneaux ne comporteront, au départ,
    aucune sélection;
    initialise indexSelectionGauche pour indiquer que le premier élément de suiteDeSuivi
    désigne, au départ, le panneau gauche;
    laisse toutes les autres variables d'instance à nil"
    dictionnaire := Dictionary new.
    suiteDeSuivi := OrderedCollection with: nil with: nil with: nil.
    indexSelectionGauche := 1
```

Lancement de l'application

ouvrirFenetre

```
    "ouvre une fenêtre pour l'application"
    |form |
    "création du cadre de l'application"
    fenetre := CwTopLevelShell createApplicationShell: 'MaFenetre' argBlock:[ : w |
                    w title: 'Utilisation d''un dictionnaire analogique', nomDuFichier].
    "création de l'espace de travail de l'application"
    forme := fenetre createForm: 'MaForme' argBlock: nil.
    forme manageChild.
    "mise en place des quatre panneaux de l'application"
    listeGauche := forme createList: 'MaListeGauche' argBlock: [ : w |
                    w leftAttachment: XmATTACHFORM;
                    rightAttachment: XmATTACHPOSITION;
                    rightPosition: 33; bottomPosition: 90;
                    topAttachment: XmATTACHFORM ;
                    bottomAttachment: XmATTACHPOSITION;
                    selectionPolicy: XmSINGLESELECT;
                    items: (dictionnaire keys asArray) ].
    listeCentrale := forme createList: 'MaListeCentrale' argBlock:[ : w |
                    w leftAttachment: XmATTACHPOSITION;
                    leftPosition: 33; rightPosition: 66;
                    rightAttachment: XmATTACHPOSITION;
                    topAttachment: XmATTACHFORM;
                    bottomAttachment: XmATTACHPOSITION;
                    bottomPosition: 90;
                    selectionPolicy: XmSINGLESELECT].
    listeDroite := forme createList: 'MaListeDroite' argBlock: [ : w |
                    w leftAttachment: XmATTACHPOSITION;
                    leftPosition: 66; bottomPosition: 90;
                    rightAttachment: XmATTACHFORM;
                    topAttachment: XmATTACHFORM ;
                    bottomAttachment: XmATTACHPOSITION;
                    selectionPolicy: XmSINGLESELECT].
```

```
listeBas := forme createText: 'MaListeBas' argBlock:[ : w | w
                    editMode: XmMULTILINEEDIT;
                    width: 300;
                    leftAttachment: XmATTACHFORM;
                    rightAttachment:XmATTACHFORM;
                    topAttachment: XmATTACHPOSITION;
                    topPosition: 90;
                    bottomAttachment: XmATTACHFORM ;].
"traitement à effectuer lorsque l'on sélectionne un élément dans la liste de gauche"
listeGauche         addCallback: XmNsingleSelectionCallback receiver: self
                    selector: #selectListe1:clientData:callData:
                    clientData: nil.
"mise en place du menu contextuel de la liste gauche"
listeGauche         addEventHandler: ButtonMenuMask receiver: self
                    selector:  #buttonGauche:clientData:event:
                    clientData:nil.
listeGauche manageChild.
listeCentrale       addCallback: XmNsingleSelectionCallback receiver: self
                    selector: #selectListe2:clientData:callData:
                    clientData: nil.
listeCentrale       addEventHandler: ButtonMenuMask receiver: self
                    selector:  #buttonCentral:clientData:event:
                    clientData:nil.
listeCentrale manageChild.
listeDroite         addCallback: XmNsingleSelectionCallback receiver: self
                    selector: #selectListe3:clientData:callData:
                    clientData: nil.
listeDroite         addEventHandler: ButtonMenuMask receiver: self
                    selector:  #buttonDroit:clientData:event:
                    clientData:nil.
listeDroite manageChild.
listeBas manageChild.
"création des menus de l'application"
menuGauche := listeGauche  createSimplePopupMenu: 'menu gauche' argBlock: [ : w |
                           w buttons:  #('Charger' 'Sauver'  'Ajouter' 'Modifier' '<----' '---->')].
menuGauche          addCallback: XmNsimpleCallback receiver: self
                    selector: #menuGaucheItem:clientData:callData:
                    clientData: nil.
menuCentral := listeCentrale  createSimplePopupMenu: 'menu central' argBlock:[ : w |
                           w buttons:  #('Charger' 'Sauver'  'Ajouter' 'Modifier' '<----' '---->')].
menuCentral         addCallback: XmNsimpleCallback receiver: self
                    selector: #menuCentralItem:clientData:callData:
                    clientData: nil.
menuDroit := listeDroite  createSimplePopupMenu: 'menu droit' argBlock:[ : w |
                           w buttons:  #('Charger' 'Sauver'  'Ajouter' 'Modifier' '<----' '---->')].
menuDroit           addCallback: XmNsimpleCallback receiver: self
                    selector: #menuDroitItem:clientData:callData:
                    clientData: nil.
fenetre realizeWidget.
```

Affichage dans les panneaux

Le contenu d'un panneau vertical est la liste complète des mots-clés du dictionnaire, si ce panneau est le premier décrit dans *suiteDeSuivi*. Sinon, ce panneau donne les mots proches de la sélection du panneau précédent. A défaut de sélection, le panneau affiche une liste vide. Le contenu du panneau du bas doit reproduire les mots de *suiteDeSuivi*.

```
contenuCorrespondantA: indexSelection
    "renvoie, pour mettre à jour un panneau, le contenu correspondant à l'élément désigné
    par indexSelection dans suiteDeSuivi. Cette méthode ne met pas à jour la sélection du
    panneau"
    |selectionPrecedente|
    "si le panneau à remplir n'est pas le premier, selectionPrecedente désignera la sélection
    qui a été effectuée dans le panneau situé à gauche du panneau à remplir"
    indexSelection = 1
     ifTrue: ["c'est le panneau le plus à gauche : on renvoie le dictionnaire entier"
            ^ (dictionnaire keys asArray)]
     ifFalse: ["s'il n'y a aucune sélection à gauche, le panneau doit rester vide"
               selectionPrecedente := suiteDeSuivi at: (indexSelection - 1).
               selectionPrecedente = nil
                    ifTrue: [ ^ nil]
                    ifFalse: [ ^ (dictionnaire at: selectionPrecedente) subStrings]]

selectListe1: widget clientData: clientData callData: callData
    " méthode exécutée lorsque l'on sélectionne un élément dans le panneau de gauche"
    "mise à jour de la variable suiteDeSuivi"
    suiteDeSuivi at:indexSelectionGauche put: (widget selectedItems at:1).
    "mise à jour des panneaux central, droit et bas en fonction de la sélection"
    self changeGauche: (widget selectedItems at:1)

selectListe2: widget clientData: clientData callData: callData
    "méthode exécutée lorsque l'on sélectionne un élément dans le panneau central"
    "mise à jour de la variable suiteDeSuivi"
    suiteDeSuivi at:(indexSelectionGauche + 1) put: (widget selectedItems at:1).
    "mise à jour des panneaux droit et bas en fonction de la sélection"
    self changeCentral: (widget selectedItems at:1)

selectListe3: widget clientData: clientData callData: callData
    "méthode est exécutée lorsque l'on sélectionne un élément dans le panneau de droite"
    "mise à jour des panneaux en fonction de la sélection"
    self changeDroit: (widget selectedItems at:1)
    self panneauBas

panneauGauche
    "message envoyé au panneau gauche pour initialiser ou mettre à jour son contenu"
    (self contenuCorrespondantA: indexSelectionGauche) notNil
            ifTrue:[ listeGauche items:
                              (self contenuCorrespondantA:(indexSelectionGauche ))]
```

```
panneauCentral
    "message envoyé au panneau gauche pour initialiser ou mettre à jour son contenu"
    (self contenuCorrespondantA: indexSelectionGauche + 1) notNil
            ifTrue:[ listeCentrale items:
                        (self contenuCorrespondantA:(indexSelectionGauche + 1))]

panneauDroit
    "message envoyé au panneau gauche pour initialiser ou mettre à jour son contenu"
    (self contenuCorrespondantA: indexSelectionGauche + 2) notNil
            ifTrue:[ listeDroite items:
                        (self contenuCorrespondantA:(indexSelectionGauche + 2))]

panneauBas
    "message envoyé au panneau du bas pour initialiser ou mettre à jour son contenu"
    | affichage chaine|
    chaine := ''.
    (suiteDeSuivi at:1)=nil
            ifTrue: [ 'Aucune sélection' ]
            ifFalse:[ suiteDeSuivi do: [ :element |
                    element = nil ifFalse:[ chaine := chaine,' ',element]]].
    listeBas setString:chaine
```

Synchronisation des panneaux

Une sélection dans le panneau gauche entraîne une modification de chacun des autres panneaux. Une sélection du panneau central entraîne une modification du panneau droit et du panneau du bas. Une sélection dans le panneau droit fait « glisser » les panneaux vers la gauche : on doit modifier tous les panneaux et rétablir les sélections adéquates (traitement réalisé par la méthode *ajustePanneaux*).

```
changeGauche: cleChoisie
    "méthode exécutée avec un changement global dans le panneau gauche"
    suiteDeSuivi at: indexSelectionGauche put: cleChoisie.
    "mise à jour du panneau gauche"
    suiteDeSuivi at: indexSelectionGauche put: motSelectionne.
    "mise à jour du panneau central"
    suiteDeSuivi at: (indexSelectionGauche + 1) put: nil.
    self panneauCentral.
    "mise à jour du panneau droit"
    suiteDeSuivi at: (indexSelectionGauche + 2) put: nil.
    self panneauDroit.
    "suppression des éléments inutiles de suiteDeSuivi"
    [suiteDeSuivi size > (indexSelectionGauche + 2)]
            whileTrue: [suiteDeSuivi removeLast].
    "mise à jour du panneau du bas"
    self panneauBas
```

```
changeCentral: cleChoisie
    "méthode exécutée avec un changement global dans le panneau central"
    suiteDeSuivi at: (indexSelectionGauche + 1) put: cleChoisie.
    "mise à jour du panneau droit"
    suiteDeSuivi at: (indexSelectionGauche + 2) put: nil.
    self panneauDroit.
    "suppression des éléments inutiles de suiteDeSuivi"
    [suiteDeSuivi size > (indexSelectionGauche + 2)]
            whileTrue: [suiteDeSuivi removeLast].
    "mise à jour du panneau du bas"
    self panneauBas
changeDroit: motSelectionne
    "méthode exécutée avec un changement global dans le panneau droit"
    suiteDeSuivi at: (indexSelectionGauche + 2) put: motSelectionne.
    "on supprime les éléments de la liste qui sont au delà du panneau droit"
    [(suiteDeSuivi size) > (indexSelectionGauche + 2)]
            whileTrue: [suiteDeSuivi removeLast].
    "on ajoute un élément qui représentera le nouveau panneau droit après le
    décalage vers la gauche"
    suiteDeSuivi add: nil.
    "on fait maintenant glisser les sélections vers la gauche :
     l'ancien panneau central deviendra le nouveau panneau gauche"
    indexSelectionGauche := indexSelectionGauche + 1.
    self ajustePanneaux
ajustePanneaux
    "met à jour tous les panneaux en conservant les sélections existantes"
    "mise à jour du panneau gauche"
    self panneauGauche.
    self remetSelection: indexSelectionGauche pour: listeGauche.
    "mise à jour du panneau central"
    self panneauCentral.
    self remetSelection: (indexSelectionGauche + 1) pour: listeCentrale.
    "mise à jour du panneau droit"
    self panneauDroit.
    self remetSelection: (indexSelectionGauche + 2) pour: listeDroite.
    "mise à jour du panneau du bas"
    self initialisePanneauBas
remetSelection: indexSelection pour: unPanneau
    "remet, dans le panneau unPanneau, la sélection indiquée par indexSelection"
    |cle|
    cle := suiteDeSuivi at: indexSelection.
    cle notNil
     ifTrue: [unPanneau selectItem:cle notify:false]
```

Appel des menus

Les méthodes d'interface avec les fichiers (*chargeDuDisque* et *sauveSurDisque*) sont communes à tous les menus. Il en va de même des méthodes de « glissement » (*allerAGauche* et *allerADroite*). Comme on le verra dans les paragraphes suivants, les méthodes de mise à jour du dictionnaire, pour l'addition et la modification, sont propres à chaque panneau mais font toutes appel aux mêmes méthodes de base : *ajouteAPartirDe:avecIndex:* et *modifieAPartirDe:avecIndex:*

```
buttonGauche:widget clientData:clientData event: event
    "affichage du menu dans le panneau de gauche"
    event button = 3  ifFalse:[ ^self].
    menuGauche menuPosition: event;
                manageChild

buttonCentral:widget clientData:clientData event: event
    "affichage du menu dans le panneau central"
    event button = 3  ifFalse:[ ^self].
    menuCentral menuPosition: event;
                manageChild

buttonDroit:widget clientData:clientData event: event
    "affichage du menu dans le panneau de droite"
    event button = 3  ifFalse:[ ^self].
    menuDroit menuPosition: event;
              manageChild

menuGaucheItem:widget clientData:clientData callData:callData
    "associe les options du menu à des traitements"
    self perform: ( #( #chargeDuDisque  #sauveSurDisque #ajoutG #modificationG
                    #allerAGauche #allerADroite)  at: clientData + 1)

menuCentralItem:widget clientData:clientData callData:callData
    "associe les options du menu à des traitements"
    self perform: ( #( #chargeDuDisque  #sauveSurDisque #ajoutG #modificationG
                    #allerAGauche #allerADroite)  at: clientData + 1)

menuDroitItem:widget clientData:clientData callData:callData
    "associe les options du menu à des traitements"
    self perform: ( #( #chargeDuDisque  #sauveSurDisque #ajoutG #modificationG
                    #allerAGauche #allerADroite)  at: clientData + 1)
```

Interface avec les fichiers

On notera que la méthode qui charge un dictionnaire à partir d'un fichier doit mettre à jour la fenêtre (barre de titre et panneaux. On rappelle (cf 6.1.2) que le format choisi pour les fichiers-dictionnaires est celui d'un fichier-texte, avec la convention suivante. Une analogie étant représentée par un mot-clé *m* et une liste de mots considérés comme proches de *m*, on la représentera par deux lignes dans le fichier, la première donnant le mot *m* précédé par le libellé « analogies pour », la seconde, commençant par un retrait de deux espaces pour améliorer la lisibilité, donnera la liste des mots proches de *m*.

```
nomDuFichier: uneChaine
    "affecte uneChaine à la variable d'instance nomDuFichier
     si une fenêtre est ouverte, on change son label pour indiquer le nouveau fichier"
    nomDuFichier := uneChaine.
    fenetre notNil
        ifTrue: [ fenetre  title: 'Utilisation du dictionnaire ', uneChaine;
                           updateTitle]

chargeDuDisque
    "détermine le nom du fichier de chargement et garnit la variable d'instance dictionnaire
     renvoie true si tout s'est bien passé, false si le fichier était de format incorrect"
    |fichier ligne1 ligne2|
    self nomDuFichier: (CwTextPrompter prompt: 'nom du fichier de chargement ?'
                                       answer: 'd:\chap6\sentimts.txt').
    fichier := EstdCodeStream pathName: nomDuFichier.
    [fichier atEnd]
        whileFalse: [ ligne1 := fichier fileStream nextLine subStrings.
                      (ligne1 size < 3)
                          ifTrue:[ ^false].
                      fichier atEnd
                          ifTrue:[ ^false].
                      ligne2 := fichier fileStream nextLine trimBlanks.
                      dictionnaire at: (ligne1 at: 3) put: ligne2].
    self panneauGauche.
    self panneauCentral.
    self panneauDroit.
    self panneauBas.
    ^true

sauveSurDisque
    "recopie le dictionnaire dans le fichier désigné par nomDuFichier"
    |sauvegarde "représentera le fichier de sauvegarde"|
    "on demande d'abord à l'utisateur d'indiquer ou de confirmer le nom du fichier"
    nomDuFichier isNil
      ifTrue:[self nomDuFichier:
                  (CwTextPrompter prompt: 'nom du fichier pour la sauvegarde ?'
                                  answer: 'd:\chap6\dico.txt')].
    sauvegarde := EstdCodeStream pathName: nomDuFichier.
    dictionnaire associationsDo:
        [: couple "désignera chaque analogie enregistrée"|
         sauvegarde  fileStream nextPutAll: 'analogies pour ', couple key; cr;
                                nextPutAll: '  ', couple value; cr].
    self panneauGauche.
    self panneauCentral.
    self panneauDroit.
    self panneauBas.
    sauvegarde close.
```

Glissement latéral des panneaux

```
allerADroite
    "cette méthode avance d'un panneau vers la droite"
    ((indexSelectionGauche +2) = suiteDeSuivi size)
            ifTrue: [CwMessagePrompter warn: 'Impossible'  ^nil].
    indexSelectionGauche := indexSelectionGauche + 1.
    self ajustePanneaux

allerAGauche
    "cette méthode revient un panneau en arrière"
    indexSelectionGauche = 1
            ifTrue: [CwMessagePrompter warn: 'Impossible'. ^nil].
    indexSelectionGauche := indexSelectionGauche - 1.
    self ajustePanneaux
```

Ajout d'analogies

Les méthodes de base sont *ajoute:pour:*, qui permet d'ajouter un mot proche d'un mot-clé du dictionnaire et *ajouteRelationsEntre:et:*, qui enregistre une relation symétrique entre deux mots (en faisant appel deux fois à *ajoute:pour:*). Les méthodes *ajoutG*, *ajoutC* et *ajoutD* correspondent aux options « Ajouter » des menus des trois panneaux verticaux. Elles font toutes les trois appel à la même méthode *ajouteAPartirDe:avecIndex:*

```
ajoute: motAssocie pour: motCle
    "ajoute la relation motCle/motAssocie dans le dictionnaire récepteur
    crée l'association si motCle n'était pas répertorié
    renvoie true en cas de succès et false si la relation était déjà enregistrée"
    |listeMotsAssocies|
    (dictionnaire includesKey: motCle)
            ifTrue: [ "motCle est déjà une clé enregistrée dans le dictionnaire"
                    listeMotsAssocies := dictionnaire at: motCle.
                    "listeMotsAssocies reçoit la chaîne des mots en relation avec motCle"
                    "maintenant, on regarder si motAssocie est enregistré avec de motCle"
                    (listeMotsAssocies subStrings includes: motAssocie)
                            ifTrue:  [ ^false "car on a trouvé motproche"]
                            ifFalse:  [dictionnaire at: motCle
                                                    put: listeMotsAssocies, ' ', motAssocie.
                                    "ajoute motAssocie aux mots associés déjà existants"
                                    ^ true]]
            ifFalse: ["motCle n'est une clé déjà enregistrée dans le dictionnaire"
                    dictionnaire add: (Association key: motCle value: motAssocie).
                    ^ true]
```

```
ajouteRelationsEntre: mot1 et: mot2
	"ajoute au dictionnaire, si elles n'y sont pas déjà, les relations mot1/mot2 et mot2/mot1,
	 renvoie false si l'une des deux relations était déjà présente, sinon renvoie true"
	|reponse "indiquera si tout s'est bien passé"|
	reponse := self ajoute: mot1 pour: mot2.
	^ (self ajoute: mot2 pour: mot1) & reponse

ajoutC
	"opération d'ajout déclenchée dans le panneau gauche"
	self ajouteAPartirDe: listeCentrale avecIndex: (indexSelectionGauche + 1)

ajoutD
	"opération d'ajout déclenchée dans le panneau gauche"
	self ajouteAPartirDe: listeDroite avecIndex: (indexSelectionGauche + 2)

ajoutG
	"opération d'ajout déclenchée dans le panneau gauche"
	self ajouteAPartirDe: listeGauche avecIndex: indexSelectionGauche

ajouteAPartirDe: unPanneau avecIndex: indexSelection
	"méthode chargée d'enregistrer un ajout dans le dictionnaire; son exécution peut être
	 déclenchée par l'un des trois panneaux verticaux qui se transmet lui-même en argument
	et communique son index dans suiteDeSuivi"
	|cle motsProposes motsAjoutes texte motsExistants|
	cle := suiteDeSuivi at: indexSelection.
	cle isNil
		ifTrue: [  texte := 'pour :(tape le mot de départ ci dessous)'. motsExistants := #()]
		ifFalse: [  texte := '(confirme ou modifie le mot de départ)'.
				motsExistants := (dictionnaire at: cle) subStrings].
	cle := CwTextPrompter  prompt: ('ajout d''une analogie ', texte)  answer: cle.
	motsProposes :=
		(CwTextPrompter  prompt: 'tape le(s) mot(s) proches séparés par des espaces'
					answer: '').
	motsProposes notNil
		ifTrue:[ motsProposes := motsProposes subStrings asSet.
		motsAjoutes := motsProposes  reject:[: candidat |
						motsExistants includes:candidat].
		motsAjoutes do:[: mot | self ajouteRelationsEntre: cle et:mot].
		self ajustePanneaux]
```

Modification du dictionnaire (ajout et suppression)

Pour les suppressions, les opérations de base correspondent aux deux méthodes *supprime:pour:* et *supprimeRelationsEntre:et:*. Les méthodes *modificationG*, *modificationC* et *modificationD* correspondent aux options « Modifier » des menus des trois panneaux verticaux. Elles font toutes les trois appel à la même méthode *modifieAPartirDe:avecIndex:*

```
supprime: motAssocie pour: motCle
    "supprime la relation motCle/motAssocie dans le dictionnaire récepteur.
    supprime l'association complète si motAssocie etait la seule analogie de motCle
    renvoie true en cas de succès et false si la relation n'était pas enregistrée"
    |listeMotsAssocies nouvelleListe|
    (dictionnaire includesKey: motCle)
            ifFalse:  [^ nil]
            ifTrue:   [ "motCle est déjà une clé enregistrée dans le dictionnaire"
                        listeMotsAssocies := dictionnaire at: motCle.
                        "listeMotsAssocies reçoit la chaîne des mots en relation avec motCle"
                        "maintenant, il faut regarder si motAssocie est enregistré pour motCle"
                        listeMotsAssocies := listeMotsAssocies subStrings.
                        (listeMotsAssocies includes: motAssocie)
                           ifFalse:  [^false "car on n'a pas trouvé motproche"]
                           ifTrue: ["s'il n'y a plus que motAssocie, on supprime l'association"
                                 listeMotsAssocies size = 1
                                     ifTrue:  [ dictionnaire removeKey: motCle.
                                                ^ true]
                                     ifFalse: [  nouvelleListe :=''.
                                                 listeMotsAssocies do:
                                     [:mot| mot = motAssocie
                                     ifFalse:[ nouvelleListe := nouvelleListe,mot, ' ']].
                                                dictionnaire at: motCle put: nouvelleListe.
                                                ^ true]]]

supprimeRelationsEntre: mot1 et: mot2
    "supprime, si elles existent dans le dictionnaire, les relations mot1/mot2 et mot2/mot1
     renvoie false si l'une des deux relations était déjà présente, sinon renvoie true"
    |reponse "indiquera si tout s'est bien passé"|
    reponse := self supprime: mot1 pour: mot2.
    ^ (self supprime: mot2 pour: mot1) & reponse

modificationG
    "opération de modification déclenchée dans le panneau gauche"
    self modifieAPartirDe: listeGauche avecIndex: indexSelectionGauche

modificationC
    "opération de modification déclenchée dans le panneau central"
    self modifieAPartirDe: listeCentrale avecIndex: (indexSelectionGauche + 1)

modificationD
    "opération de modification déclenchée dans le panneau droit"
    self modifieAPartirDe: listeDroite avecIndex: (indexSelectionGauche + 2)
```

```
modifieAPartirDe: unPanneau avecIndex: indexSelection
    "méthode chargée d'enregistrer une modification dans le dictionnaire;
     son exécution peut être déclenchée par l'un des trois panneaux verticaux
     qui se transmet lui-même en argument et communique son index dans suiteDeSuivi"
    |cle motsProches motsChoisis motsSupprimes motsAjoutes texte|
    cle := suiteDeSuivi at: indexSelection.
    cle isNil ifTrue: [ texte := 'pour :(tape le mot de départ ci dessous)']
              ifFalse: [ texte := '(confirme ou modifie le mot de départ)'].
    cle := CwTextPrompter prompt: ('modification d''analogie ', texte) answer: cle.
    motsProches := dictionnaire at: cle ifAbsent: [motsProches := ''].
    motsChoisis := (CwTextPrompter prompt: 'modifie le ou les mots proches'
                                   answer: motsProches).
    motsChoisis notNil
     ifTrue: [motsChoisis := motsChoisis subStrings asSet.
              motsProches := motsProches subStrings.
              motsSupprimes := motsProches reject:[: candidat |
                                          motsChoisis includes:candidat].
    motsAjoutes := motsChoisis reject:[: candidat | motsProches includes:candidat].
    motsAjoutes do:[: mot | self ajouteRelationsEntre: cle et:mot].
    motsSupprimes do:[: mot | self supprimeRelationsEntre: cle et:mot].
    (motsSupprimes size > 0)
              ifTrue:[self metAJourSuiteDeSuivi].
    self ajustePanneaux]
```

```
metAJourSuiteDeSuivi
    "après des suppressions de relations, cette méthode annule, s'il y en a, les sélections qui
     correspondent aux mots supprimés; elle renvoie true si suiteDeSuivi a été modifiée"
    |index motCourant motPrecedent modification|
    "on parcourt suiteDeSuivi en validant ou invalidant chaque mot rencontré"
    index := 1.
    [(suiteDeSuivi at: index) notNil] whileTrue:
            [motCourant := suiteDeSuivi at: index. "on est sûr que motCourant n'est pas nil"
             index = 1 ifTrue:[ "on est au début de suiteDeSuivi : la sélection
                                doit faire partie du dictionnaire"
                               modification := (dictionnaire includesKey: motCourant) not]
                       ifFalse:[ "on n'est pas au début de suiteDeSuivi : motCourant doit
                                 être abandonné s'il n'est plus un mot proche de la sélection
                                 précédente"
                                 motPrecedent := suiteDeSuivi at: (index - 1).
                                 modification := ((dictionnaire at: motPrecedent) subStrings
                                                 includes: motCourant) not].
            "ici, si modification est vraie, c'est que l'on avait une sélection à invalider"
            modification
                    ifTrue:  [suiteDeSuivi at: index put: nil " et on sort de l'itération"]
                    ifFalse: [index := index + 1 "et on continue l'itération"]
            ].
    "sortie de l'itération : si index est inférieur à la taille de suiteDeSuivi,
    il faut tronquer suiteDeSuivi tout en conservant au moins trois éléments"
    (index = suiteDeSuivi size) ifTrue: [^ modification].
    [index < 3] whileTrue: [index := index + 1. suiteDeSuivi at: index put: nil].
    "suppression des éléments inutiles de suiteDeSuivi"
    [suiteDeSuivi size > index] whileTrue: [suiteDeSuivi removeLast].
    "on doit encore ajuster indexSelectionGauche"
    indexSelectionGauche > (suiteDeSuivi size - 2)
            ifTrue: [indexSelectionGauche := suiteDeSuivi size - 2].
    ^ modification
```

Bibliographie

[Bro 87] Fred Brooks.
No Silver Bullet - Essence and Accidents of Software Engineering
IEEE Computing, April 1987

[Dig 86] Smalltalk/V : Tutorial and Programming Handbook
Digitalk inc., 1986

[IBM 94] IBM Smalltalk Prorammer's Reference
IBM Corporation, 1994

[GoR 89] A. Goldberg & D. Robson
Smalltalk-80 : The Language
Addison Wesley, 1989

[LaP 90] R. Lalonde & JR Pugh
Inside Smalltalk (volumes 1 et 2)
Prentice Hall

[Mas 89] G. Masini et al
Les langages à objets
InterEditions 1989

[PiW 88] L.J Pinson & R.S. Wiener
An Introduction to Object Oriented Programming and Smalltalk
Addison Wesley, 1988

Index alphabétique

$ (méthode) 22
* (méthode) 28-29, 54
+ (méthode) 53
/ (méthode) 53, 55
// (méthode) 55
\\ (méthode) 55
= (méthode) 70, 132
== (méthode) 106, 132
@ (méthode) 25
, (méthode) 106
add: (méthode) 97, 101, 114
add:after: (méthode) 114
add:afterIndex: (méthode) 114
add:before: (méthode) 114
add:beforeIndex: (méthode) 114
addAll: (méthode) 121
addAllFirst: (méthode) 114
addAllLast: (méthode) 114
addFirst: (méthode) 114
addLast: (méthode) 114
adjustSize (méthode) 128
adresse 126
after: (méthode) 114
after:ifNone: (méthode) 114
argument
    d'un bloc 33
    d'une méthode 35, 72
Array (classe)
    créer une instance 58
    manipuler une instance de 56
    sous-classe de Collection 57
    tableau 11, 23
asArray (méthode) 55, 100
asArrayOfSubstrings (méthode) 20, 66, 88-89, 172
asBag (méthode) 100
ASCII (code) 67, 75-77, 162
asFloat (méthode) 51-53
asIndexedCollection (méthode) 100
asInteger (méthode) 42-43
asLowercase (méthode) 175
asSet (méthode) 100
associationAt:ifAbsent: (méthode) 102
associationDo: (méthode) 102-104
asSortedCollection (méthode) 100
asString (méthode) 71
Astre (classe) 109
asUpperCase (méthode) 21, 66, 68
at: (méthode) 12, 22, 30, 102, 114, 118
at:ifAbsent: (méthode) 102
at:put: (méthode) 12, 30, 57, 62, 101, 114
atAllPut: (méthode) 107
atAll:Put: (méthode) 107
atEnd (méthode) 142, 172
Bag (classe)
    classe 74, 95
    manipuler une instance de 97
    méthodes 118
basicAt: (méthode) 127
basicAt:put: (méthode) 127

basicSize (méthode) 127
become: (méthode) 74, 129
before: (méthode) 114
before:ifNone: (méthode) 114
between:and: (méthode) 2, 16
blocs 21, 41
Boolean (classe) 42-43
browse (méthode) 17
Browse Disk 153
browser 17, 59, 71, 73, 76-77
buffer 164
ByteArray (classe) 74
caractères
    classe Character 74
    manipuler une chaîne 66
    stockage en mémoire 126
cascade 26, 58
CfsDirectoryDescriptor 175
CfsFileDescriptor 175-176
chaîne 2-4, 7
CharacterConstants (dictionnaire partagé) 157, 163
checkIndex: (méthode) 134
class (méthode) 10, 17-18, 51-52, 58-59, 69, 71, 73, 76-77, 92-93
Class Hierarchy Browser 59, 74
classe
    abstraite 13, 42, 47, 96, 105, 108
    créer 74-75
    généralités 2, 8-10
    sauver les méthodes d'une 77
classVariableNames: 38
clé 95
cliquer 5, 6, 18
close (méthode) 78, 165, 173
collect (méthode) 108
Collection (classe)
    ajouter des éléments 121
    classe 95
    généralités 46, 56-57, 65, 78, 81, 86
    méthodes 100, 118
    sous-classes 95-96
    stockage en mémoire 126, 135
contents (méthode) 150
convertir
    des instances de collection 100
    nombres en caractères 139
    octets en nombres 161
    méthode de hachage 126
copie
    d'un fichier 156
    d'une collection 107, 140
copy (méthode) 107
copyFrom:to: (méthode) 62, 107
copyFrom:to:with: (méthode) 107
copySelection (méthode) 184-185
copyWith: (méthode) 107
copyWithout: (méthode) 107
cr (méthode) 158
Cr (abréviation) 163
createForm: 178
createText: 181
crochets 8, 21, 33
curseur 5-7
cutSelection (méthode) 184-185
CwComposite 179-180
CwForm 178-180
CwList 179-180
CwPrimitive 179-180
CwShell 179-180
CwText 179-180
CwTextPrompter 171
CwTopLevelShell 178
CwWidget 179
degreesToRadians (méthode) 52
denominator 27-29, 49, 53-54
denominator: (méthode) 28, 49-50
detect (méthode) 113
dictAdd: 171
dictAt:put: 171
Dictionary (classe)
    classe 78, 96, 100-101
    sous-classe 130, 132

elements (variable d'instance) 119
méthodes 131
le dictionnaire Smalltalk 113
différence avec la classe IdentityDictionary 132
dictionnaires partagés 162

Directory (classe) 155

Display 6-7, 20

do it 20

do: (méthode) 32, 50, 83, 146, 183, 198

drive: (méthode) 36

edit (méthode) 67

éditeur 4, 6, 12, 63, 67, 169

élément 95

elementCount 86, 170-171

énumération 95

encapsulation 15

equal (méthode) 16, 82

error: (méthode) 62, 64

errorInBounds (méthode) 134

errorNotIndexable (méthode) 118

EsLinearHashSet (classe) 171

EstdCodeStream (classe) 172

EtFileNamePrompter (classe) 175

expressions 2-4

false 16, 42-44

fenêtre
d'une application 181, 195-197
généralités 3-7
menu 182
ouverture d'un flux sur une collection 140

fichier
copier 156
créer 153
généralités 80, 83
lire 155
modifier 157
nom 66
sauvegarder 163
sauvegarde Smalltalk 77
utiliser 168-169, 172, 175-176

file d'attente (simulation) 116

File (classe)
classe 155, 163
variables d'instance
lastByte 163
writeLimit 164
pageStart 164
position 164

file (méthode) 155

FileControlBlock (classe) 155

fileExtension 66

fileIn (méthode) 78

fileName 66

FileStream (classe) 38, 77-78, 153, 155

find:ifAbsent: (méthode) 124

findElementIndex: (méthode) 124, 127

findFirst (méthode) 106

findLast (méthode) 106

first (méthode) 106, 116

firstIndex (méthode) 62

FixedSizeCollection (classe)
classe 57, 66, 77, 105, 108
méthodes 135
sous-classe 108, 135

Float (classe) 13-14, 22, 29, 46-49, 51-55

floatError (méthode) 52

flush (méthode) 165

flux 138
ouvrir 140
parcourir 142

fonction de hachage 125-127

fraction 13-15, 22, 27-29, 31, 47-51, 53-55

from:to: (méthode) 55

from:to:by: (méthode) 55

fromInteger: (méthode) 51, 53

grow (méthode) 128

growEmptyBy: 171

guillemets 8

hachage
fonction 126
gestion des collisions 127
notion 81-82, 124, 169

hash (méthode) 128

héritage 11-12, 14-16, 68
hiérarchie des classes 12, 17
IdentityDictionary (classe) 132
ifFalse: (méthode) 15, 41, 43
ifTrue: (méthode) 8-9, 15, 41-43
ifTrue:ifFalse: (méthode) 15, 41
implementedBySubclass (méthode) 121
includes: (méthode) 98, 100
includesAssociation: (méthode) 100
includesKey: (méthode) 100
index 132
IndexedCollection (classe)
    classe 57, 66, 69, 96, 105
    méthodes 134
indexOf: (méthodes) 106
indexOf:ifAbsent: (méthode) 106
indice 95
initialize (méthode) 119
inject:into: (méthode) 105
inspect (méthode) 50-51
inspection 50-51, 83-87
Integer (classe) 74
Interval (classe) 96
isEmpty (méthode) 77, 146
isKindOf: (méthode) 92
isNil (méthode) 27, 173
itération 9-10, 34, 44-46, 81
key:value: (méthode) 78
keyAtValue: (méthode) 102
keys (méthode) 134
keys: (méthode) 102
keysDo: (méthode) 103-104
last (méthode) 106
Lf (abréviation) 157, 163
lineDelimiter (méthode) 38, 157, 163
lineDelimeter: (méthode) 157
listes instances de collection 113, 135
Magnitude (classe) 13-19, 47, 79, 92-93
mantisse 51
matrices 59-65
mémoire (utilisation)
    accès aux objets 126
    hachage 124, 126-127
menu 67, 73, 182
message
    binaire 2, 25-26, 44, 48, 53, 73
    unaire 2, 8, 25-26, 43, 45
    à mots-clés 8, 15, 25-26, 44
méthode 7-9
    d'instance 11
    de classe 11
    privée 119
model 169
Model/View/Controller 188
MouseButton 36, 38
MVC 188
name (méthode) 176
new (méthode) 97
new: (méthode) 57, 75, 126
next (méthode) 142
next: (méthode) 144, 150
next:put: (méthode) 149
nextBytePut: (méthode) 160
nextLine (méthode) 160, 172
nextMatchfor: (méthode) 144
nextPut: (méthode) 148, 152
nextPutAll: (méthode) 3-4, 7, 12, 26, 39, 56, 80-81, 88, 148, 173
nextTwoBytesPut: (méthode) 160
nextWord (méthode) 158
nil 127
notNil (méthode) 27, 197, 200-201
Number (classe) 14, 18, 21, 27, 29, 46-48, 51-56, 93, 121
numerator 27-29, 49, 53-54
numerator: (méthode) 28, 49-50
numerator:denominator: (méthode) 28, 49, 51
Object (classe) 13, 16-17, 27, 30, 38, 42, 50, 57, 66, 79, 84, 187, 189-190, 195
occurrencesOf: (méthode) 98, 102, 120

octets
    généralités 27
    manipulation 76, 160
    conversion en entiers 160
on: (méthode) 141
on:from:to: (méthode) 141
opendir:pattern:mode: (méthode) 175
openMenu (méthode) 182-184
OrderedCollection (classe)
    classe 105, 113, 195-196
    méthode 135
    variables d'instance 135
        endPosition 135
        startPosition 135
outputToPrinter (méthode) 67
pageStart (FileStream) 164
panneau 17-20, 51, 59-60, 79, 84, 86, 92-93, 170-171, 181-182, 185-187, 190, 192-202
pathName: (méthode) 36-37, 78, 155, 172-173, 202
peekFor: (méthode) 144
pile 116
Point (classe) 25
pointeur 27, 31-33, 37, 46, 58, 126
polymorphisme 14-17, 19, 29, 43, 48, 55, 72
poolDictionary 162
position (variable d'instance de Stream) 151, 164
primitives 47, 49, 51-52, 54, 73, 89, 146
printOn: (méthode) 63, 67, 158
prompt:default: (méthode) 79
Prompter (classe) 42-43, 46, 79-81
protocole 18, 20
putSpaceAtEnd (méthode) 135
putSpaceAtStart (méthode) 135
radiansToDegrees (méthode) 52
radix (méthode) 48
ReadStream (classe) 38, 141
ReadWriteStream (classe) 146
realizeWidget (méthode) 179
reject: (méthode) 131
remove:ifAbsent: (méthode) 98, 115
removeAssociation: (méthode) 103
removeFirst (méthode) 115
removeIndex: (méthode) 115
removeKey: (méthode) 103, 113
removeKey:ifAbsent: (méthode) 103
removeLast (méthode) 115
rename: (méthode) 175
replaceFrom:to:with: (méthode) 107, 114
replaceFrom:to:with:startingAt: (méthode) 107
replaceFrom:to:withObject: (méthode) 107
reversed (méthode) 106
reverseDo: (méthode) 107
secteur 165
select: (méthode) 131
sélecteur 8, 11, 13, 15-16, 18
sélection 198-201
selectionPolicy: (méthode) 192
self 8-9, 14-16, 122
senders (méthode) 19
Set (classe)
    classe 96, 100
    croissance d'une instance 128
    méthodes 123
setLimits (méthode) 152
setString: (méthode) 181-182
setToEnd (méthode) 143
show: (méthode) 149
show it 20
simulation 116
size (méthode) 97, 121
skipTo: (méthode) 145
Smalltalk (dictionnaire) 113
sortBlock (méthode) 117
sortBlock: (méthode) 117
sortBlock: (variable d'instance) 117
SortedCollection (classe) 117

souris 153, 188
space (méthode) 77
splitPath: (méthode) 175
squared (méthode) 14, 47-48
Stream (classe)
  classe 71, 137, 172
  méthodes 136
  variable d'instance
    readLimit 138, 151
    writeLimit 148, 151
    collection 149, 151
  sous-classes 146, 151
String (classe) 8, 10-11, 15, 20-21, 30, 35, 45, 65, 68, 172, 174-176, 181
subclass: (méthode) 27, 38
super 85-87, 190
Symbol 21, 58
symbole 5, 8-9, 21, 36, 38, 65, 92
synchronisation 193, 196-197
System Transcript 99
tab (méthode) 158
tableau 56
tampon (mémoire) 139, 185
template 19
timesRepeat: (méthode) 46
title: (méthode) 178
to:by: (méthode) 55
transferts 139
trim: (méthode) 67
trimBlanks (méthode) 67, 172, 175-176
truncated (méthode) 24, 26
UndefinedObject (classe) 11, 27, 31, 33
upTo: (méthode) 144, 176
value (méthode) 34, 40, 43, 46, 78, 82, 88, 92
value: (méthode) 34, 42, 78, 85, 89
value:value: (méthode) 34, 42
values 102-103
variableByteSubclass 68, 75
variables 3-4, 8-9
  de classe 38
  dictionnaires partagés 37
  globales 36
  partagées 35
  temporaires 33
variableSubclass 59, 75
variableSubclass: 38, 59, 77, 83, 195
variableWordSubclass 76
whileFalse: (méthode) 44-45
whileTrue: (méthode) 9, 44-46, 71, 80
widget 179-180
Windows 167, 177-178, 188
with: (méthode) 11, 57, 64-65, 69, 83, 85-86, 91, 176, 196
with:with: (méthode) 57-58, 65, 100, 135
with:with:with: (méthode) 11, 57, 65, 84
with:with:with:with: (méthode) 65, 68
withCrs (méthode) 67
withDo: (méthode) 107
with:from:to: (méthode) 147
WriteStream (classe) 146
writtenOn 38
XmPUSHBUTTON 184
XmSEPARATOR 184
XmSINGLESELECT 191-192
yourself (méthode) 122
zeroDivisor (méthode) 24

MASSON Éditeur
120, Bd Saint-Germain
75280 Paris Cedex 06
Dépôt légal : mars 1996

Normandie Roto Impression s.a.
61250 Lonrai
N° d'imprimeur : 960155
Dépôt légal : février 1996

Manuels Informatiques Masson

Programmer objets avec Smalltalk

Gilles Clavel • Nathalie Lopez • Luc Veillon

La programmation orientée objets reste difficile à aborder pour un débutant. Cet ouvrage en fournit une approche progressive et pédagogique qui s'appuie sur des exemples pratiques. Pour ces exemples, les auteurs ont utilisé l'environnement Smalltalk qui présente deux avantages : il ne nécessite pas de connaissances techniques préalables et il n'est pas non plus un prototype de laboratoire accessible aux seuls universitaires.

Depuis la première édition de cet ouvrage, la situation industrielle a sensiblement évolué. Smalltalk ne sert plus à illustrer les concepts objets : il est devenu un outil à part entière pour le développement de logiciel. Désormais, trois implémentations de Smalltalk se partagent le marché, mais la plupart des caractéristiques du langage leur sont communes. Elles sont présentées dans les cinq premiers chapitres de la présente édition, avec des exemples en Smalltalk V et IBM Smalltalk. Quant à l'interface graphique, qui diffère au sein de chaque implémentation, les auteurs ont retenu celle d'IBM Smalltalk.

Le livre s'articule en trois parties :

– découverte des concepts de base que l'on retrouvera dans la plupart des environnements orientés objets ;

– approfondissement des concepts précédents, avec une explication des mécanismes internes, en s'appuyant sur l'architecture de Smalltalk ;

– construction d'une application disposant d'une interface graphique. On montre ainsi la facilité d'emploi des langages orientés objets dès que l'on veut manipuler des objets complexes tels que des fenêtres, menus, listes, dictionnaires...

Cet ouvrage s'adresse plus particulièrement aux étudiants en informatique, aux élèves des écoles d'ingénieurs et à leurs enseignants. Il est également accessible à tout utilisateur de micro-ordinateur désireux d'apprendre un langage d'une conception radicalement différente de celle de langages comme Basic ou Pascal. Il pourra servir de base et d'aide-mémoire aux programmeurs expérimentés, se formant aux environnements orientés objets.

Professeur à l'Institut national agronomique, Gilles Clavel est directeur consultant de la société IMA-informatique. Nathalie Lopez y est responsable des formations méthodes objet. Luc Veillon est chargé de recherches à l'ORSTOM et responsable informatique du laboratoire ERMES.

LEXIS NUMERIQUE

ISBN : 2-225-85157-3