

Programmation à Objets avec Smalltalk-80

Pascal ANDRE

Université de Nantes
2, rue de la Houssinière - BP 92208
44322 Nantes Cedex 03

pascal.andre@univ-nantes.fr

RÉSUMÉ. Ce cours vise à donner aux stagiaires un aperçu de la programmation à objets, avec un langage représentatif de ce type de programmation : SMALLTALK-80. Après avoir donné les bases d'un modèle à objets, nous explorons le monde de Smalltalk-80, c'est à dire le modèle à objets, l'interface utilisation, les classes, l'environnement de développement VisualWorks et la méthode de programmation. Ce cours fait partie d'un cours sur le génie logiciel et la notation UML et de travaux pratiques de développement à objets. Des exemples pratiques illustrent ce cours.

MOTS-CLÉS : Génie Logiciel, Modélisation à objets, Programmation à objets, Smalltalk-80.

ABSTRACT. This course is a general overview of object-oriented programming, with a representative and pure object oriented language: SMALLTALK-80. After a short presentation of the language concepts, we explore the Smalltalk-80 world, the rich interface library and the development tools of VisualWorks. This course is a companion book of a course on Software Engineering and the a deeper UML Notation. It works with practical examples of Smalltalk programs.

KEY WORDS : Software Engineering, Development Methods, Object-Orientation, Smalltalk-80

Table des matières

1	Introduction	5
1	Introduction à Smalltalk-80	6
2	Le modèle à objets de Smalltalk-80	9
1	Concepts et principes de base	9
2	Objet et classe	9
3	Envoi de message	10
4	Éléments du langage	11
5	Affectation, égalité, copie	12
6	Héritage	13
7	Visibilité des attributs	15
8	Méthodes comme des objets	16
9	Méta-objets	16
10	Définition d'une classe	17
3	La programmation avec Smalltalk	21
1	Exemple introductif	21
2	Quelques hiérarchies de classes	36
4	L'environnement de programmation de Smalltalk-80	49
1	Mise en route	49
2	Outils principaux et fenêtres de l'environnement	53
3	Représentation textuelle du code	61
4	Programmation avec VisualWorks	63
5	Le modèle MVC	77
1	Principes	77
2	Architecture MVC	80
3	Exemples d'utilisation du MVC	80
4	Précisions sur le MVC	83
5	Interfaces MVC avec VisualWorks	85
6	Le développement d'applications avec VisualWorks	95
1	Dessin de l'interface	95
2	Création des classes du modèle	102
3	Liaison entre le modèle et l'interface	103
4	Vérification et compléments	105
5	Exécution et tests	105
6	Bilan	105
7	Conclusion	107

A	Exercices	111
1	Langage	111
2	Classes	111
3	Programmation à objets	112
4	Applications	113
5	Un peu plus	113
B	Compléments	115
1	La souris	115
2	Références autour de Smalltalk	115
3	VisualWorks 3.0/linux	118
C	Solution des exercices	123
1	Conception et programmation d'une gestion de match de tennis	123
2	Générateur de compilateurs	142

Chapitre 1

Introduction

La programmation à objets est issue des travaux sur le langage SIMULA dans les années 1960. La technique n'est pas donc récente mais elle a largement évolué notamment avec des langages comme Smalltalk (1970). Elle a aussi bénéficié des travaux sur représentation des connaissances en intelligence artificielle (Minsky, 1975) et le travail coopératif (systèmes distribués, parallélisme) (Hewitt, 1973). Ces différents courants sous-entendent toutefois des sémantiques variées du concept d'objet : un objet est une unité de connaissance, une unité de calcul, un module, une valeur d'un type, etc. Par exemple, la notion d'objet est utilisée en intelligence artificielle pour représenter des connaissances (par exemple, un objet est un *frame*) et pour mettre en œuvre les mécanismes de raisonnement (par exemple, un objet est un *acteur*). On a aussi vu fleurir des langages hybrides. La technique est devenue mature et reconnue dans les années 80 et aujourd'hui nombre d'applications sont développées en utilisant une approche objet. Dans ce document, nous nous restreignons aux modèles à objet avec classes, ceux qu'on utilise en Génie Logiciel. Selon la classification de [Weg90], les langages à classes sont caractérisés par trois concepts essentiels : les objets, les classes et l'héritage. En ce sens, un langage tel qu'Ada, qui possède des mécanismes d'abstraction de données et d'encapsulation (les paquetages) n'est pas considéré comme un langage à classes. Deux autres "styles" à objets existent : les frames et les acteurs. Consulter [MNC⁺90] pour une description générale des langages à objets. Le lecteur trouvera aussi dans le chapitre 2 de [And01] un exposé sur les modèles à objets (concepts) et le développement à objets. Enfin, le rapport [AR98] constitue une bonne introduction à la modélisation à objets. Par abus de langage, le complément de nom "à objets" désignera une approche avec des classes.

L'approche à objets a marqué une rupture avec l'approche structurée qui dominait le développement du logiciel depuis 1960, et qui est encore largement employée. L'univers de l'application est structuré en termes d'objets et non plus en termes de procédures. L'algorithme et la structure de données de la programmation structurée ont fait place à une collection d'objets qui collaborent, dans la programmation à objets. La programmation à objets est une approche modulaire avec un grain de modularité fine, l'objet, qui encapsule données et traitements. Rappelons que qu'un bon programme modulaire groupe les morceaux de programmes par affinité (cohésion forte) et limite les interactions entre modules (couplage faible). Avoir de petites unités facilite leur écriture, leur compréhension, leur preuve et leur maintenance. La modification du code par ajout d'objets ne perturbe pas fondamentalement l'organisation du système, celui est donc extensible. Les classes fournissent un mécanisme d'abstraction qui améliore la conception du logiciel en augmentant l'usage possible des composants. L'héritage renforce l'abstraction et l'utilisation verticale du code (autre forme d'extensibilité). L'envoi de message favorise aussi l'abstraction et la concision du code. Le parallélisme est naturellement induit par les objets. Ceci étant, la mise en œuvre dans des architectures systèmes traditionnelles pose le problème de la répartition explicite des objets. Les avantages attendus en termes de qualité du logiciel sont principalement l'extensibilité, la réutilisabilité. Mais la

modularité favorise aussi la lisibilité, l'intégrité, la compatibilité et la vérifiabilité. L'objectif est donc de capitaliser les efforts de développement en favorisant l'évolution et la réutilisation de composants logiciels. Pour cela, il faut une certaine culture de développement du composant [Mey89, Mey97]. L'objet est aussi bien adapté à la programmation à grande échelle (gros programmes, nombreux participants, développement sur la durée, formation importante) car il permet la répartition des activités.

Le langage Smalltalk est dans la lignée directe des langages à classes. On peut même affirmer que c'est LE langage de référence pour la programmation à objets au même titre que le langage Pascal est la référence en programmation structurée. Il a eu une grande influence sur les langages à objets plus récents tels que Java. De plus Smalltalk est uniforme, sa seule structure de contrôle est l'envoi de message, ce qui facilite l'écriture et la compréhension du code.

Le document est structuré comme suit. Dans la section 1, nous présentons rapidement Smalltalk-80, qui est à la fois un langage, un système d'exploitation et un environnement. Smalltalk est un langage à objets avec une syntaxe et une sémantique. Il est inspiré des la programmation fonctionnelle et de la programmation impérative. Le typage est dynamique. Smalltalk-80 contient toutes les fonctionnalités d'un système d'exploitation : gestion du processeur, gestion de la mémoire, gestion des périphériques. L'ensemble est réalisé par des classes. Ce qui donne une implantation très agréable du système d'exploitation. Smalltalk-80 est un environnement de programmation comprenant des outils avancés d'édition, des outils de gestion de classes et de méthodes (rangement, recherche, modification) et enfin des outils de mise au point sophistiqués (débugueurs, évaluateurs, inspecteurs).

Nous n'étudions pas Smalltalk-V qui est un langage différent de Smalltalk-80. Nous étudions ensuite plus en profondeur les concepts du modèle à objets de Smalltalk-80 dans la section 2. Dans la section 4, nous détaillons l'environnement de programmation de Smalltalk-80. Cet environnement est riche et le système entier est directement accessible. Dans la section 2, nous étudions quelques hiérarchies importantes du système. Enfin, dans la section 5, nous étudions rapidement le modèle Model/View/Controller, la méthode de développement d'interfaces homme/machine de Smalltalk.

Le discours est valable pour les versions 4.0 et suivantes de Smalltalk-80. Le système VisualWorks permet en plus de construire directement et aisément des interfaces graphiques. Nous ne parlerons pas ici des spécificités de cette boîte à outils. Ce rapport est une version actualisée pour Windows d'un rapport précédent [And96]. La bibliographie de référence est la suivante : la description du langage et du système dans sa version 2.5 sera abordée avec profit dans [BS96]. Deux ouvrages généraux sur Smalltalk-80 sont [Gol83, Lal94]. Enfin, concernant les versions récentes de Smalltalk-80 et la programmation sous VisualWorks, consulter [HH95, How95, Sha97].

1 Introduction à Smalltalk-80

Smalltalk-80 [BS96, Gol83] est à la fois un langage, un système d'exploitation et un environnement de programmation. **Smalltalk, ObjectWorks et VisualWorks sont des marques déposées de ParcPlace Systems, Inc.** La richesse de la bibliothèque de classes prédéfinies et sa facilité d'utilisation en font un outil idéal pour le prototypage d'applications. A la différence d'Eiffel ou C++, toutes les classes sont accessibles directement dans l'environnement de travail et donc redéfinissables.

Smalltalk est un langage à objets avec une syntaxe et une sémantique expressive. Il est inspiré des la programmation fonctionnelle (Lisp) et de la programmation impérative (Simula). On retrouve ces influences au travers des nombreux concepts du langage (objets, classes, métaclasses, méthode comme objet, super-méthode, héritage simple, processus, typage dynamique, etc.) et des **outils de développement** (ramasse-miette, évaluation d'expressions

quelconques, inspecteurs, débogueurs). Smalltalk comprend aussi un riche environnement de classes pour la **persistance** et de **traitement d'exceptions**.

Smalltalk-80 contient toutes les fonctionnalités d'un système d'exploitation : gestion du processeur, gestion de la mémoire, gestion des périphériques. L'ensemble est réalisé par des classes. Ce qui donne une implantation très agréable du système d'exploitation. Les objets sont stockés dans un environnement appelé **Workspace**. L'allocation dans cet espace est implicite à la création d'objets (méthode **new**). La récupération se fait par un **ramasse-miette** et non par des destructeurs comme en C++.

Smalltalk-80 est un environnement de programmation comprenant des outils avancés d'édition, des outils de gestion de classes et de méthodes (rangement, recherche, modification) et enfin des outils de mise au point sophistiqués (débogueurs, évaluateurs, inspecteurs). Smalltalk-80 permet de gérer différentes applications dans l'environnement. Ces applications sont appelées **projets**. Nous n'en parlerons pas ici. L'environnement est la donnée de trois fichiers : un fichier contenant les classes de base du système, un fichier contenant les objets et un fichier **journal** contenant les modifications du système depuis la création de l'environnement. Ces fichiers sont détaillés dans la section 1. L'utilisation de l'environnement se fait interactivement avec la souris, qui prend dans Smalltalk-80 une importance cruciale. Trois boutons activent trois menus contextuels, sensiblement différents des menus déroulants à la Windows ou à la Macintosh. Ces points sont détaillés dans la section 1.5.

L'interface avec des fichiers se fait par la commande **fileIn** appliquée à des descriptions textuelles Smalltalk (fichier avec le suffixe **.st**). Il est conseillé de rentrer les classes dans l'ordre *super-classe* \rightarrow *sous-classe* pour éviter les liens indéfinis. Cette commande exécute aussi les méthodes d'initialisation de classe.

A la différence d'Eiffel [Mey97], les noms de variables et de méthodes ne contiennent pas le caractère souligné, qui dans une version précédente de Smalltalk, désignait le destinataire du résultat d'une méthode (sorte d'affectation). La convention est d'écrire la première lettre du premier mot en minuscule et la première lettre des mots suivants en majuscule. Certains noms ne doivent pas être interprétés, ils sont désignés par des symboles, ils sont préfixés par #.

La programmation en Smalltalk consiste à découvrir les classes du système et à les adapter à ses propres besoins. C'est là tout l'esprit du développement à objets.

Chapitre 2

Le modèle à objets de Smalltalk-80

1 Concepts et principes de base

Smalltalk est un langage à objets pur. Cela signifie qu'il y a seulement deux concepts de base : l'**objet** et l'**envoi de message**. Tout le reste du langage est dérivé de ces deux concepts.

2 Objet et classe

Smalltalk est un langage purement à objets à objets.

Principe 2.1 (objet) *Toute entité en Smalltalk est un objet. Tout objet a une **identité** propre et implicite.*

Chaque objet a un numéro (une identité) distinct dans le système. Ce qui différencie deux objets, c'est leur identité. Un **objet** sert à stocker et accéder à des informations propres et à envoyer ou répondre à des messages. Un objet a un état et un comportement. L'état est l'ensemble des valeurs que détient l'objet. Le comportement est l'ensemble des opérations (procédures ou fonctions), appelées **méthodes** que l'objet peut réaliser. Par exemple, prenons un point $p1 = (5, 6)$ défini par ses coordonnées cartésiennes. L'état est décrit par les deux coordonnées 5 et 6 dans le plan. Le comportement est **déplacer**, **distance**, etc. On dit qu'un objet **encapsule** des données (l'état) et des traitements sur ces données (le comportement) au sein d'une même entité.

Principe 2.2 (classe) *Tout objet est instance d'une classe.*

La **classe** regroupe les objets ayant même structure (même forme d'état) et même comportement. Par exemple, notre point $p1$ est défini dans une classe **Point**. La **structure** est définie par un ensemble de **variables d'instances**. Par exemple, la structure du point $p1 = (5, 6)$ est formée de deux variables x et y . Le **comportement** est décrit par un ensemble de méthodes. Une **méthode** est une abstraction procédurale (procédure, fonction) définie par un **profil** et un **corps**. Le profil comprend un nom de méthode, appelé **sélecteur** de la méthode et des noms de variables en paramètres. Syntactiquement, une méthode se présente comme suit :

```
sélecteur et nom des arguments  
  "commentaires décrivant la méthode"  
  | noms de variables temporaires |  
  
instructions
```

Chaque objet détient les valeurs de ses variables d'instances mais son comportement est situé au niveau de sa classe. L'objet est relié à sa classe par la **relation d'instanciation**. Nous avons donc schématiquement l'implantation suivante :

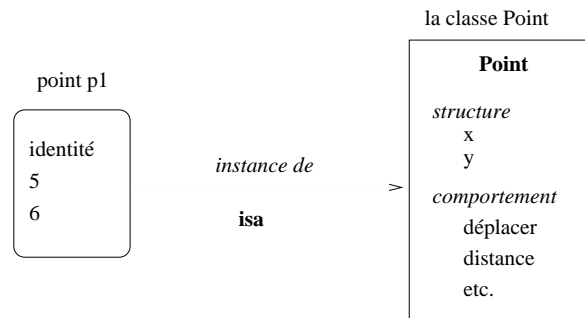


Figure 1 : Relation d'instanciation du point *p1*

Le point *p1* a pour coordonnées $x = 5$ et $y = 6$. Ses méthodes sont regroupées dans la classe **Point** avec qui il est relié par une relation d'instanciation. L'état d'un objet est alors défini par la valeur de ses variables d'instance. C'est ce qui permet de comparer deux objets d'une même classe. En fait, chaque objet est unique dans le système : l'objet possède une identité.

La classe représente à la fois l'ensemble de ses instances et un modèle des instances. Dans le comportement on distingue les méthodes applicables aux objets, ce sont les **méthodes d'instance**, des méthodes applicables à la classe elle-même, appelées **méthodes de classe**. Les méthodes de classe servent notamment à créer les objets de la classe. On dit qu'un objet est instancié par la classe. Les termes instance et objet sont synonymes. Une classe qui n'a pas de méthode d'instanciation est dite **classe abstraite**. De même, une méthode dont la sémantique n'est pas définie est dite **méthode abstraite**.

3 Envoi de message

Principe 3.1 (envoi de message) *Un objet n'est accessible que par envoi de message.*

L'envoi de message est un appel de méthode d'un objet. L'objet destinataire de l'envoi de message est appelé **receveur**. On parle de **sélection simple** car il n'y a qu'un seul receveur. Syntactiquement, l'envoi de message s'écrit différemment selon le nombre de paramètres. Chaque paramètre est introduit par un mot-clé suffixé par ':'.

```
méthode unaire   : receveur sélecteur
méthode binaire  : receveur sélecteur: argument
méthode n-aire  : receveur sélecteur1: argument1 ... sélecteurn: argumentn
```

Le sélecteur est l'union des mots-clés **sélecteur = sélecteur₁:... sélecteur_n**.

Exemple :

corner: aPoint

"Rend un rectangle dont l'origine est le receveur et l'extrémité est le point donné en paramètre. C'est une forme infixée de création de rectangles."

↑Rectangle origin: self corner: aPoint

La seule **structure de contrôle** est l'envoi de message. Deux envois de message sont séparés par un ';'. Deux envois de messages consécutifs au même receveur sont séparés par un ';'(cascade). Les structures de contrôle habituelles (**alternatives, itérations**) sont donc implantées par des envois de message. Plus précisément ce sont des méthodes de la classe **Boolean** pour les alternatives et de la classe **Collection** pour les itérations.

4 Eléments du langage

4.1 Variables

En Smalltalk, les **variables** sont déclarées par un nom. Leur type n'est connu qu'à l'exécution. On distingue plusieurs sortes de variables :

- les variables liées à une expression en cours d'évaluation : **variables locales**,
- les variables liées à une méthode : **variables locales**,
- les variables propres à une instance : **variables d'instance**,
- les variables partagées par les instances d'une classe : **variables de classes** ,
- les variables partagées par un ensemble d'instance, appelées aussi variables de **pool**.

Par rapport à d'autres langages, Smalltalk possède deux types de variables d'instances : les variables référant à un objet (classique) et les **variables indexées**, contenant un tableau de variable. Les variables indexées sont gérées directement par les primitives du langage. Nous n'en dirons pas plus. Sachez cependant que les variables indexées facilitent le codage et l'efficacité d'accès pour certains objets tels que les collections ou les vues.

Chaque variable est accessible dans la portée sous-entendue par son nom : une variable locale est visible dans le bloc qui la déclare, une variable d'instance est visible par une instance, une variable de classe est visible par la classe, ses sous-classes et ses instances. Ces différents types de variables seront examinés dans les sections suivantes. Attention, il ne faut pas confondre les variables d'instance de la méta-classe des variables de classe. Les premières sont propres à chaque méta-classe : si elles sont modifiées elles le sont uniquement pour la classe. Les secondes sont communes à la classe et ses sous-classes : si une méthode modifie la valeur alors toutes les classes ont cette nouvelle valeur.

4.2 Réflexion : la variable self

Il est possible pour le receveur d'un message de s'envoyer un message. La construction syntaxique pour cela est la pseudo variable **self**, qui désigne le receveur. Nous définissons ainsi des **méthodes récursives**, comme la fonction factorielle. La fonction factorielle est définie dans la classe des entiers. Le corps de la méthode est une alternative sur la valeur de l'entier. Si l'entier est nul alors la factorielle est égale à 1 sinon elle est égale au produit de l'entier par la factorielle de son précédent.

factorielle

```
"rend la valeur factorielle du receveur "  
self = 0  
ifTrue: [↑1]  
ifFalse: [↑self * (self -1) factorielle ]
```

En Smalltalk, une expression alternative est codée par un envoi de message à un objet **expr_bool** de type booléen (de classe **Boolean**) : **expr_bool ifTrue:[bloc1] ifFalse:[bloc2]**. Cet envoi de message a deux paramètres de type bloc. Un bloc est une suite d'instruction évaluée uniquement à l'exécution. La notion de bloc est détaillée dans la section 4.2. L'envoi de message est évalué comme suit : si le booléen est vrai le bloc **bloc1** est exécuté sinon le bloc **bloc2** est exécuté. Notez que le retour du résultat d'une opération se fait en préfixant par **^**. Il s'agit d'une sortie inconditionnelle de la méthode en cours.

4.3 Objets littéraux

En programmation, on distingue les types de base des autres types (types structurés, types abstraits, classes). Les types de base sont issus de la représentation interne de la machine (exemples : booléens, entiers, réels). En Smalltalk, tout est unifié dans le concept d'objet : les types de base sont encapsulés dans des classes. En fait, lorsqu'on regarde les méthodes

des objets des classes de ces objets, nous nous apercevons qu'elles font appel à des fonctions primitives inaccessibles. Il y a donc en Smalltalk des types de base (des types dont les valeurs ne sont pas des objets). Par exemple, la plupart des classes feuilles de la hiérarchie d'héritage suivante sont des classes primitives.

```
Object ()
  Magnitude ()
  ArithmeticValue ()
  Number ()
  Fraction ('numerator' 'denominator')
  Integer ()
    LargeNegativeInteger ()
    LargePositiveInteger ()
    SmallInteger ()
  LimitedPrecisionReal ()
    Double ()
    Float ()
  Point ('x' 'y')
  Character ()
  Date ('day' 'year')
  LookupKey ('key')
  Association ('value')
  Time ('hours' 'minutes' 'seconds')
```

Des **objets littéraux** sont des objets connus et désignés par leur valeur (unique) : `nil` désigne un objet vide, `55` pour un entier, `#toto` pour un symbole, `'toto'` pour une chaîne de caractères, etc.

5 Affectation, égalité, copie

L'affectation d'un objet à une variable se fait par `variable := objet`, où `objet` est un objet résultat d'une expression. En fait, la variable reçoit une valeur qui est l'identité de l'objet résultat. Cette identité est l'adresse de l'objet dans le système (adressage unique et global). L'accès à l'objet se fait donc indirectement par son adresse. Cette indirection a un effet sur la copie et la comparaison d'objets. Ces protocoles sont définis dans la classe `Object`.

La copie d'objet est soit uniquement sur la valeur des variables d'instances (copie de la structure et valuation des variables d'instance par les identités) soit une copie complète et en profondeur des valeurs de l'objet (récursif). La première est appelée **shallow copy** et la seconde est appelée **deep copy**.

De même l'égalité entre deux variables désignant des objets porte sur l'identité (`==`) ou une notion redéfinissable de l'égalité (`=`). Cette redéfinition de `=` permet de prendre en compte la copie superficielle ou la copie en profondeur.

Prenons l'exemple d'un ensemble de points, déclarés comme suit.

```
| p1 p2 p3 s1 s2 s3 | "variables locales |'a l' expression "
```

```
p1 ←Point x: 5 y: 6. "version instanciation "
```

```
p2 ←7@8. "version littérale "
```

```
p3 ←p1.
```

```
s1 ←Set new.
```

```
s1 add: p1; add: p2; add: p3.
```

```
s2 ←s1 shallowCopy.
```

```
s3 ←s2 deepCopy.
```

```
s4 ←Set new.
```

```
s4 add: s1; add: s2; add: s3.
```

```
s4 printString
```

Le résultat est `'Set(Set(506 708) Set(506 708) Set(506 708))'`

Noter que le type ensemble est générique et peut contenir toutes sortes d'objets. L'ajout de l'occurrence `p3` dans `s1` n'a pas eu lieu car `p1` y était déjà et que les ensembles ont des éléments en un seul exemplaire.

Examinons dans la figure 2 l'effet des différentes copies et affectation.

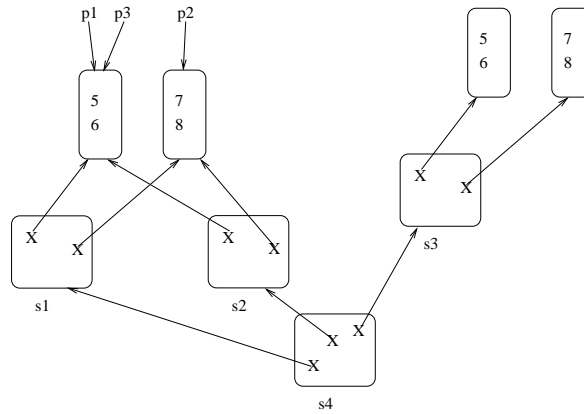


Figure 2 : Un exemple de copie en Smalltalk

6 Héritage

L'innovation la plus importante du modèle objet est l'héritage. Voici une définition extraite de [Mey97].

Définition 6.1 (héritage) *L'héritage est un mécanisme permettant de définir une nouvelle classe (la sous-classe) à partir d'une classe existante (la super-classe) par extension ou restriction.*

L'extension se fait en rajoutant des méthodes dans le comportement ou des variable d'instances dans la structure. Par exemple, un employé est une personne qui a un contrat de travail. La restriction consiste à réduire l'espace des valeurs définies par la classe en posant des contraintes (conditions logiques à vérifier) sur ces valeurs. Par exemple, un carré est un rectangle dont les quatre côtés sont égaux.

L'héritage induit le **polymorphisme** : une instance de la sous-classe est aussi une instance de la super-classe. Cette instance peut donc utiliser sans les définir les méthodes de la super-classe.

Dans la pratique, l'héritage peut être utilisé à tort et à travers (il y a parfois même confusion entre héritage et utilisation). On peut redéfinir les méthodes et changer leur profils. On doit donc donner des règles précises de contrôle de l'héritage pour conserver un contrôle de type sûr. Des règles ont été données dans [Car88, Mey97, ACR94].

Smalltalk autorise uniquement l'**héritage simple** : chaque classe hérite d'au plus une super-classe. La structure est héritée. Les variables sont redéfinissables implicitement puisque le typage est dynamique. Par contre, une variable déclarée dans une super-classe ne peut être redéclarée dans la sous-classe. Les méthodes sont systématiquement héritées des super-classes. Elles sont redéfinissables.

L'unique **source** du graphe d'héritage est la classe `Object` : toutes les classes héritent de cette classe. Elle même hérite de l'objet indéfini `nil`.

6.1 Recherche de méthode

Lors d'un envoi de message `expr sél args`, le mécanisme suivant est exécuté.

1. le receveur est calculé par évaluation de l'expression `expr`
2. la méthode de sélecteur `sél` est recherchée à partir de la classe du receveur (classe courante).
 - (a) si elle est trouvée, elle est appliquée avec les arguments `arguments`
 - (b) sinon la méthode de sélecteur `sél` est recherchée dans la super-classe de la classe courante et ainsi de suite jusqu'à la trouver ou arriver dans la classe `Object`.
 - (c) si la recherche échoue dans la classe `Object`, alors le message `doesNotUnderstand:` est envoyé avec comme paramètre le sélecteur `sél`. Nous rejoignons là le problème du **traitement des exceptions**.

L'utilisation de récursion par la pseudo variable `self` apporte ici toute sa puissance : la méthode la plus spécifique sera celle utilisée à l'exécution.

6.2 Super méthode

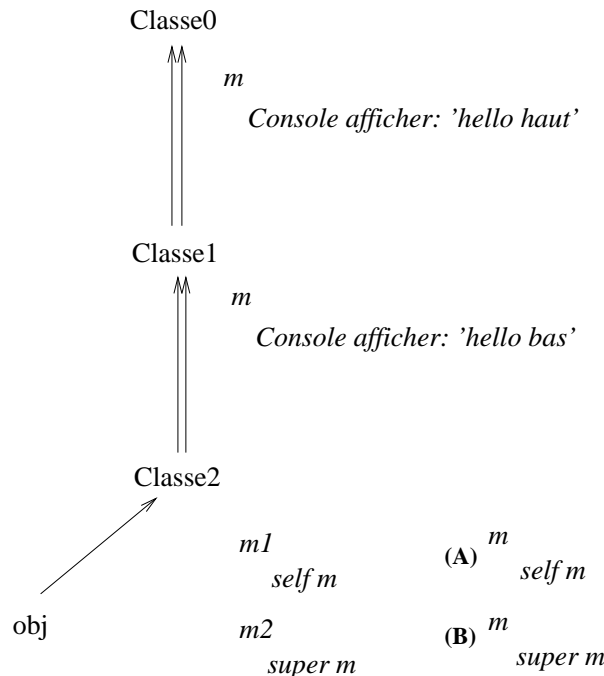


Figure 3 : La super-méthode en Smalltalk

La super méthode est une simplification du concept de méthode qualifiée en CLOS [BDG⁺88]. Elle permet de combiner les différentes définitions d'une méthode par la pseudo-variable `super`. Cette caractéristique est liée à la recherche de méthode : la recherche de la méthode est double. La première recherche fournit la méthode principale qui se trouve dans une classe `Classe` qui est soit la classe du receveur soit une super-classe de la classe du receveur. Avec la pseudo-variable `super`, l'interpréteur commence alors une seconde recherche de la méthode à appliquer. L'évaluation des paramètres se fait sur le résultat de cette seconde recherche. Par exemple supposons la hiérarchie de classes et d'instances de la figure 3. La flèche simple représente la relation d'instanciation. La flèche double représente la relation d'héritage. L'évaluation de l'envoi de message `obj m1` affiche `hello bas` sur la console. L'évaluation de l'envoi de message `obj m2` affiche `hello haut` sur la console. Dans l'alternative (A), l'envoi de messages `obj m` boucle. Dans l'alternative (B), l'envoi de messages `obj m` affiche `hello bas` sur la console.

6.3 Instanciation et classes abstraites

La méthode d'instanciation par défaut est la méthode de classe `new`. Plusieurs variantes existent selon les paramètres possible. Cette méthode est définie dans la classe `Object` redéfinie dans de nombreuses sous-classes d'`Object`. En général, sa définition se fait en appelant une méthode d'instance `initialize` qui initialise les valeurs par défaut de l'objet créé. Voici un exemple classique.

```
new
  "méthode d'instanciation "
  ↑super new initialize
```

Les classes abstraites sont utiles à la structuration du graphe d'héritage. Elles permettent de factoriser des comportements communs de haut-niveau. Par exemple, la classe `Magnitude` (`Comparable` en Eiffel) rassemble tous les objets munis des opérations de comparaison (`=`, `>`, `>=`, etc.).

Définition 6.2 (classe abstraite)

Une classe abstraite est une classe non feuille du graphe d'héritage et qui n'est jamais instanciée dans le système.

Par exemple considérons des étudiants et des enseignants. La classe des personnes est une classe abstraite, en effet un individu particulier sera instanciée à partir d'une des sous-classes spécifiques. La classe `Personne` définit le comportement commun à toutes ses sous-classes. Classe abstraite et métaclasse sont parfois confondus, à tort : une classe abstraite peut être une métaclasse ou n'être qu'une classe ordinaire et vice-versa. Nous pouvons dire qu'une classe abstraite est un nœud interne du graphe d'héritage et une métaclasse un nœud interne du graphe d'instanciation.

Il n'y a pas de distinction entre classe abstraite et classe instanciable en Smalltalk. Cette distinction est forcée en redéfinissant la méthode `new`. Nous ne ferons qu'indiquer la différence en commentaire. Les méthodes abstraites sont implantées par `self subclassResponsability`. Le masquage est possible en Smalltalk par `self shouldNotImplement`.

Nous expliquerons plus en détail le mécanisme d'instanciation et le graphe d'instanciation dans la section 9.

6.4 Variables et méthodes de classe

Dans les langages de classes on utilise parfois la terminologie méthode et variable d'instance ou méthode et variable de classe. Une méthode de classe, par opposition à une méthode d'instance est une méthode prévue pour s'appliquer à une classe et non pas à une instance. C'est le cas par exemple de la méthode `new`. Les variables de classes sont utilisées pour représenter des informations commune à l'ensemble des instances d'une classe. Si l'information n'est visible qu'en lecture alors une méthode d'instance est une solution suffisante. Une **variable de classe** est une variable qui est partagée (lecture et/ou écriture) par toutes les instances de cette classe. Il est possible de la voir comme appartenant à la classe et non plus à ses instances. La présence de méta-objets facilite la description de variables et de méthodes de classes. En l'absence de méta-objets, les variables et méthodes de classe ne sont pas accessibles par envoi de message habituel : il s'agit d'opérations primitives. Nous approfondissons ces concepts dans les sections 10 et 9.

7 Visibilité des attributs

En Smalltalk, les variables d'instance ne sont pas visible à l'extérieur d'un objet. Il faut écrire des méthodes pour y accéder : accesseurs en lecture et écriture. Par contre, elles sont

accessibles dans les sous-classes. Il y a ici des différences notables avec Eiffel et C++.

8 Méthodes comme des objets

Smalltalk considère les méthodes comme des objets à part entière (méthodes d'ordre supérieur). En général une méthode est restreinte à une procédure exécutable. L'idée est de rajouter des informations diverses : syntaxe, filtre pour les arguments, documentation, application etc. L'intérêt est une plus grande uniformité du langage et surtout la possibilité d'exploiter ces informations supplémentaires pour améliorer la convivialité du langage. Ces méthodes sont manipulées au travers des **blocs**.

Un bloc est une expression du langage pouvant contenir des variables locales et des paramètres. Il en existe deux classes : `BlockClosure` et `BlockContext`. La cloôture correspond à l'évaluation d'un bloc de contexte par un processus.

Il est décrit syntaxiquement par un envoi de message à une instance de la classe `BlockContext` ou "à la volée" par

```
[:param1 ... :paramn || variables | instructions].
```

Un bloc est un objet à part entière : il peut recevoir des messages ou être passé en paramètre. Par exemple, considérons les multi-ensemble définis par la classe `Bag`. Un multi-ensemble contient des éléments e_i en n_i exemplaires ($n_i > 0$). La méthode `sortedCounts`, extraite du protocole `accessing` de la classe `Bag` (multi-ensemble) est définie comme suit :

`sortedCounts`

"Answer a collection of counts with elements, sorted by decreasing count. "

```
| counts |
counts ← SortedCollection sortBlock: [:x :y | x >= y].
contents associationsDo:
[: assn | counts add: (Association key: assn value value: assn key)].
↑counts
```

Cette méthode `sortedCounts` rend une collection triée dont les éléments sont des couples (nombre d'occurrence, élément). Nous avons deux utilisations de blocs. Dans le premier cas, le bloc `[:x :y | x >= y]` est passé en paramètre d'un accesseur `sortBlock:`. Il sera ensuite stocké comme variable d'instance. Dans le second cas, le bloc passé en paramètre de la méthode `associationsDo:` qui est un itérateur sur les éléments du multi-ensemble. Elle a pour effet d'appliquer le bloc à chaque élément de l'ensemble. L'élément est perçu comme un paramètre et nommé par la variable `assn`.

Un bloc est exécuté par le message `value` ou ses variantes munies d'arguments (classe `BlockClosure`). Les paramètres de ces arguments sont évalués à l'entrée du bloc.

Une méthode est un bloc particulier. L'évaluation d'un bloc de type méthode se fait en lui envoyant le message `perform` ou ses variantes (`perform`, `perform: with:`, `perform: withArgs:`) munies d'arguments.

9 Méta-objets

Cette notion apparaît dès qu'une classe est considérée comme un objet. La classe a les droits d'un objet :

1. Une classe est activable par envoi de message. Les opérations liées aux classes deviennent alors des méthodes de classe. Parmi ces méthodes, deux sont particulièrement intéressantes :

- (a) La méthode d'instantiation `new`.
 - (b) La méthode de définition d'une méthode d'instance `define-method`.
2. Une classe est instance d'une classe qui définit sa description abstraite et son comportement (notamment sa relation d'héritage). La classe d'une classe est appelée *méta-classe*. Appelons `MétaClasse`, la méta-classe par défaut, du système, définie approximativement dans la figure 4. Nous supposons des types `Champ`, `Invariant`, `Méthode`, `Classe` qui peuvent être des classes. `UneClasse := new MétaClasse, "UneClasse", L'identificateur UneClasse, à gauche du symbole d'affectation (:=), désigne une variable, qui a pour valeur la nouvelle classe créée. Les "[,]" symbolisent les listes (pas de variables de classes ici). L'invariant par défaut est vrai. Les méthodes d'instance seront définies ultérieurement.`

Définition 9.1 (méta-classe) *Une méta-classe est une classe dont les instances sont des classes.*

```

structure    champs : Liste[Champ]
            contrainte : Invariant
            variablesDeClasse : Liste[Champ]
            méthodes : Dictionnaire[Méthode]
            méthodesDeClasse : Dictionnaire[Méthode]
méthodes    new : String Liste[Champ] Invariant Liste[Champ] → Classe

```

Figure 4 : *La classe MétaClasse*

La présence de méta-classes rend le langage souvent plus uniforme, en particulier la création des instances peut se faire par envoi de message. Un autre avantage est de pouvoir paramétrer et redéfinir le comportement du système objet. La présence de méta-classes s'accompagne souvent d'un **protocole des méta-objets**. Celui-ci donne une définition quasi-réflexive des classes et des méta-classes primitives du système. Il permet de plus une reconfiguration partielle du système. Ceci est possible par la définition de nouvelles méta-classes ayant des comportements différents de ceux prédéfinis, par exemple, au niveau de l'héritage ou de la création et de l'initialisation des instances. Les méta-objets n'existent pas dans tout les langages de classes, par exemple il n'y en a pas en Eiffel, en SCOOPS ou en C++. Dans un système avec méta-objets comme ObjVlisp [Coi87] la distinction entre classe et méta-classe disparaît dans la mesure où une classe est aussi une instance. Les variables et méthodes de classes ne seront plus rangées dans la structure de la classe comme dans la figure 4 mais dans la méta-classe.

En Smalltalk, chaque classe `Classe` est instance d'une seule méta-classe appelée `Classe class`. La méta-classe contient la structure (variables de classes, variables d'instance de la méta-classe) et le protocole de classe (méthodes de classes). Dans le protocole de la méta-classe, figurent les méthodes d'instanciation, les méthodes d'initialisations des variables de la structure de la classe et des méthodes globales accessibles sans créer d'instances de la classe (par exemple écriture sur la console, lancement du ramasse-miette, etc.).

10 Définition d'une classe

La définition d'une classe comprend :

- la super-classe
- les déclarations de variables d'instances,
- les déclarations de variables de classes,

- les déclarations de variables de pool,
- la catégorie de la classe (les classes sont rangées par catégories pour faciliter leur accès et leur regroupement).
- le commentaire de la classe,
- les méthodes de la classe
- les déclarations de variables d'instances de la méta-classe
- les méthodes de la méta-classe

Toutes ces informations sont transmises par envoi de message. La création d'une classe se fait par exemple par un envoi de message à sa super-classe. Par exemple, décrivons sommairement la classe `Point`.

ArithmeticValue subclass: #Point

```
instanceVariableNames: 'x y '
classVariableNames: ''
poolDictionaries : ''
category: 'Graphics-Geometry'
```

Point comment:

```
'Class Point represents an x-y pair of numbers usually designating a location
on the screen.'
```

Instance Variables:

```
x <Integer> usually x coordinate
y <Integer> usually y coordinate'
```

Les méthodes d'instances sont rangées dans des protocoles pour faciliter leur accès et la lisibilité des programmes. On retrouve habituellement les protocoles suivants :

- `initialize-release` : contient les méthodes d'initialisation et suppression des objets (il n'y a pas de suppression explicite comme en C++).
- `accessing` : contient les méthodes d'accès en lecture ou écriture pour les variables d'instance.
- `comparing` : contient les méthodes de comparaison avec d'autres objets.
- `testing` : contient les méthodes de tests divers (appartenance...).
- `copying` : contient les méthodes de copies (redéfinies).
- `converting` : contient les méthodes de conversion ou de changement de classe pour un objet (`as<Classe>`)
- `displaying` : contient les méthodes d'affichage (classes vues).
- `adding-removing` : contient les méthodes d'ajout suppression d'éléments (classes collections).
- ... : protocoles propres à chaque classe
- `printing` : contient les méthodes de représentation sous forme de chaînes de caractères (description).
- `private` : contient les méthodes non utilisables de l'extérieur (c'est un artifice car la notion de méthode privée n'existe pas en Smalltalk-80).

Examinons quelques éléments de la méta-classe `Point class` : le méthode d'instanciation `x: y:`, la méthode générale délivrant un point nul `zero`.

Point class

```
instanceVariableNames: ''
```

```
x: xInteger y: yInteger
```

```
↑self basicNew setX: xInteger setY: yInteger ! !
```

```
zero
```

```
↑0@0
```

Les méthodes de classes sont aussi rangées dans des protocoles. On retrouve habituellement les protocoles suivants :

- `instance creation` : contient les méthodes d'instanciation
- `class initialization` : contient les méthodes d'initialisation des variables de classe ou d'instance de la méta-classe.
- `defaults` : contient les méthodes implantant des valeurs par défaut (soit des variables de classe, ou d'instance de la méta-classe ou soit des calculs externes cachés).
- `Signal constants` : contient les méthodes de gestion de signaux d'exceptions.
- `general inquiries` : contient les méthodes générales de la classe.
- `backward compatibility` : contient les méthodes de compatibilité avec des versions antérieurs.
- `examples` : contient les méthodes d'exemples d'utilisation de la classe.
- `private` : contient les méthodes de classe non utilisables.

Chapitre 3

La programmation avec Smalltalk

1 Exemple introductif

Cette section est à la fois une introduction à la programmation à objets et à la programmation avec Smalltalk. Nous partons d'un algorithme classique de programmation structurée pour aboutir à une programme à objets. Nous passons par une version intermédiaire, qui est quasiment une traduction de programme Pascal en Smalltalk-80. Cette étape intermédiaire illustre la syntaxe du langage et la représentation des structures de contrôles de la programmation structurée par l'unique structure de contrôle qu'est l'envoi de message. Le lecteur trouvera dans d'autres ouvrages des exemples introductifs similaires et instructifs : jeu de Nim [Roy94], tournoi de tennis [Ner90].

Enoncé informel et programme structuré

Le but du programme est de calculer des niveaux pluviométriques annuels dans des villes à partir de relevés mensuels saisis au préalable. Les résultats sont affichés à l'écran. Ce programme est un exercice sur l'utilisation des procédures et des vecteurs donné en Deug B à Nantes. Voici le programme Pascal correspondant.

Listing 3.1 – Programme Pascal de calcul pluviométrique

```
program pluvio;
(* Deug B Nantes *)
const nb lieux=20;
type vect_char=array[1..nb lieux] of string;
    vect_int=array[1..nb lieux] of integer;
    vect_ps=array[1..12] of integer;
var lieux:vect_char;
    mois:vect_int;
    hauteurs:vect_int;
    nbsaisis:integer;
    somme:vect_ps;

procedure saisir_relevés(var lieux:vect_char; var mois:vect_int;
                        var hauteurs:vect_int; var nbsaisis:integer);
var l:string;
    h,m:integer;

begin
    writeln('donnez votre fiche: lieu, mois, hauteur. ');
    readln(l,m,h);
    nbsaisis←0;
    while (l<>'z') and (nbsaisis < nb lieux) do begin
        nbsaisis←nbsaisis+1;
        lieux[nbsaisis]←l;
        while (m>12) or (m<1) do begin
```

```

        writeln(' E R R E U R mois, recommencer ');
        readln(m)
    end;
    mois[nbsaisis]←m;
    hauteurs[nbsaisis]←h;
    writeln(' fiche suivante (z pour sortir) ');
    readln(l,m,h)
end
end;    (* saisir_relevés *)

procedure addition(mois:vect_int; haut:vect_int; nbrel:integer;
    m:integer; var som:integer);
var j:integer;

begin
    som←0;
    for j←1 to nbrel do
        if mois[j]=m then som←som+haut[j]
    end;    (* addition *)

procedure exploiter_relevés(mois:vect_int; haute:vect_int;
    nbrel:integer; var somm:vect_ps);
var i,sm:integer;

begin
    for i←1 to 12 do begin
        addition(mois,haute,nbrel,i,sm);
        somm[i]←sm
    end
end;    (* exploiter_relevés *)

procedure affiche_so(s:vect_ps);
var i:integer;

begin
    for i←1 to 12 do
        writeln(' somme du mois ',i,' est ',s[i])
    end;    (* affiche_so *)

begin    (* Début du programme *)
    saisir_relevés (lieux , mois,hauteurs,nbsaisis);
    exploiter_relevés(mois,hauteurs,nbsaisis,somme);
    affiche_so(somme)
end.    (* Fin du programme *)

```

Evolution de l'énoncé

Les modifications suivantes sont envisagées par le demandeur :

1. Les relevés doivent pouvoir être lus dans des fichiers textes en plus de la possibilité de les saisir au clavier.
2. Le nombre maximal de relevés risque d'évoluer selon les années.
3. L'auteur des relevés sera ajouté plus tard pour établir des statistiques.
4. Actuellement la saisie des fiches est faite en une fois, mais elle pourra se faire en plusieurs fois, directement à partir des stations.
5. On souhaite appliquer les mêmes traitements pour les relevés de température pour calculer les moyennes annuelles. Deux cas sont envisagés : saisie des deux valeurs sur une même fiche ou saisie de deux fiches différentes.
6. On souhaite établir maintenant des statistiques sur deux ans.

Traduction du programme en Smalltalk

Une seule classe, appelée `PluvioSimple`, est nécessaire pour implante le programme principal. La figure 5 synthétise, via la notation UML, cette classe. Les attributs et opérations soulignées font partie des protocoles de classe. Les attributs sont des variables en Smalltalk (privées par défaut). Les opération sont des méthodes en Smalltalk (publiques par défaut). L'exécution du programme est simplement une instantiation de la classe.

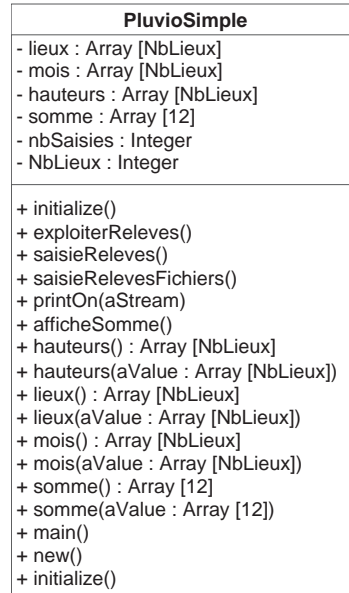


Figure 5 : *Classe PluvioSimple*

Les types ne sont pas déclarés explicitement en Smalltalk (le typage est dynamique). Les constantes peuvent être représentées

- par des variables globales de Smalltalk. Cette solution est à proscrire car la constante subsisterait après la fin du programme.
- par des variables de classe (ou d'instance de la métaclasse). Cette solution convient car, pour toute exécution (une instance de la classe est une exécution du programme) la valeur de la constante ne varie pas.
- par des variables d'instances (en fait comme une variable globale du programme Pascal initialisée dans la procédure principale). Cette solution est moins pratique car il faut initialiser la variable (qui représente la constante globale) à chaque lancement de programme (chaque instantiation de la classe).

Les variables globales sont représentées par des variables d'instance : elle oint une nouvelle valeur à chaque exécution du programme (chaque création d'une instance de la classe). Les procédures sont représentées par des méthodes d'instance de la classe. Le programme principal est représenté par une méthode de classe `main` "à la C", qui est la méthode d'instanciation de la classe.

Le code programme Smalltalk ne se visualise pas comme un programme Pascal. Un programme s'écrit par au moins une classe. La classe étant un objet à part entière, les données et le code sont répartis dans les différentes valeurs d'objet de la classe. La représentation suivante du code est donc une présentation personnalisée du code.

Classe `PluvioSimple` L'environnement Smalltalk génère aussi une version textuelle des classes avec des caractères séparateurs (' et ! notamment). Cette version est également chargeable dans l'environnement Smalltalk (voir les détails dans le chapitre 4).

Listing 3.2 – Traduction Smalltalk du programme de calcul pluviométrique

```
'From VisualWorks(R), Release 2.5 of September 26, 1995 on January 5, 1999 at 9:53:21 am'!
```

```
Object subclass: #PluvioSimple
```

```
  instanceVariableNames: 'lieux mois hauteurs somme nbSaisies '
```

```
  classVariableNames: 'NbLieux '
```

```
  poolDictionaries: ''
```

```
  category: 'PluvioMetrie'!
```

```
PluvioSimple comment:
```

```
'Cette classe implante quasi-systématiquement le programme pluvio.
```

```
Il s'agit d'une écriture Smalltalk sans conception en classes d'un programme impératif.
```

```
Le programme est implanté par une classe PluvioSimple
```

```
avec les méthodes de saisie, addition, exploitation et affichage.
```

```
Des variantes sont possibles du point de vue des interfaces.
```

```
Les variables d'instance sont :
```

```
lieux : Array of String
```

```
mois : Array of 1..12
```

```
hauteur : Array of Integer
```

```
nbSaisies : Integer
```

```
Les tableaux en Smalltalk n'étant pas bornés statiquement, la constante nblieux ne semble pas utile. Ceci étant, la taille des tableaux (size) doit être fixée explicitement, il faut l'augmenter de temps en temps.
```

```
Du coup, nous avons décidé de fixer une taille 'nblieux' à l'initialisation.
```

```
La constante nblieux étant partagées par toutes les instances de PluvioSimple, nous en avons fait une variable de classe, dont la valeur est fixée par l'initialisation de la classe.
```

```
La somme est implantée par une variable d'instance pour respecter le texte mais
```

```
comme elle est calculée à partir des trois tableaux, la méthode 'exploiterRelevés' suffit.
```

```
La principale difficulté est la saisie des entiers.
```

```
Nous passons par le scanner Smalltalk qui rend un tableau d'arguments interprétés.
```

```
Une autre possibilité consiste à utiliser une nouvelle interface avec ds champs (InputField) ont les types sont précisés.
```

```
Optimisation : lecture par fichiers.
```

```
Les interfaces de sorties peuvent être améliorées en utilisant l'exemple des histogrammes.'!
```

```
!PluvioSimple methodsFor: 'initialize-release'!
```

```
initialize
```

```
  lieux ←Array new: NbLieux.
```

```
  mois ←Array new: NbLieux.
```

```
  hauteurs ←Array new: NbLieux.
```

```
  somme ←Array new: 12 withAll: 0.
```

```
  nbSaisies ←0! !
```

```
!PluvioSimple methodsFor: 'user'!
```

```
exploiterRelevés
```

```
  "Cette méthode calcule les sommes à partir des relevés "
```

```
  somme ←Array new: 12 withAll: 0.
```

```
  (1 to: nbSaisies)
```



```
do: [:r | somme at: (mois at: r)
      put: (somme at: (mois at: r))
      + (hauteurs at: r)]!
```

saisieRelevés

```
"Cette méthode saisit les relevés .
La saisie se termine par le lieu z"
"Tester avec Stream"
"InputField facilite la lecture typee de valeurs "

| res val m l h |
res ←Dialog
  request: 'Donnez votre fiche : lieu mois hauteur (dernière fiche a pour lieu z)'
  initialAnswer: 'z'
  onCancel: ['z']. "Le probleme est la lecture des entiers "
val ←Scanner new scanTokens: res.
      "tableau interprété des valeurs lues : l = val [1], m = val [2], h = val [3] "
l ←(val at: 1) asString.
nbSaisies ←0.
[1 = 'z' | (nbSaisies > NbLieux)]
whileFalse:
  [m ←val at: 2.
   h ←val at: 3.
   nbSaisies ←nbSaisies + 1.
   lieux at: nbSaisies put: l.
   [m < 1 | m > 12]
   whileTrue:
     [Dialog warn: 'Erreur mois (1..12)'.
      m ←(Scanner new scanTokens: (Dialog request: 'Donnez le mois' initialAnswer: '1'))
      at: 1].
   mois at: nbSaisies put: m.
   hauteurs at: nbSaisies put: h.
   res ←Dialog
     request: 'Fiche suivante (z pour finir)'
     initialAnswer: 'z'
     onCancel: ['z'].
   val ←Scanner new scanTokens: res.
   l ←(val at: 1) asString]!
```

saisieRelevésFichiers

```
"Cette méthode saisit les relevés .
La saisie se termine par le lieu z"
"Quelques astuces : "
"curs ←Cursor currentCursor. "
"valeur ←(stream upTo: Character cr) tokensBasedOn: Character
space. "
"val = tableau interprété des valeurs lues : l = val [1], m = val [2], h =
val [3] "

| val aString fn fileStream indice l m h |
aString ←Dialog requestFileName: 'Please type a file name: ' default: 'releve.txt'.
aString isEmpty
ifTrue: [↑self]
ifFalse:
[fn ←aString asFilename.
 fn definitelyExists
   ifFalse: [Filename errorReporter inaccessibleSignal raiseErrorString: 'File does not exist']
   ifTrue: [fn isReadable ifFalse:
```

```

[Filename errorReporter inaccessibleSignal raiseErrorString: 'File is readProtected']
ifTrue: [fn isDirectory
ifTrue: [Filename errorReporter inaccessibleSignal raiseErrorString: 'File is a directory']
ifFalse:
  [Cursor read
  showWhile:
    [fileStream ←fn readStream.
    val ←Scanner new scanTokens: fileStream contents.
    fileStream close ].
  indice ←1.
  l ←(val at: indice) asString.
  nbSaisies ←0.
  [l = 'z' | (nbSaisies > NbLieux) | (indice + 2 > val size)]
  whileFalse:
    [m ←val at: indice + 1.
    h ←val at: indice + 2.
    m < 1 | m > 12
    ifTrue:
      [Transcript cr; show: 'Erreur mois ', m printString ,
        ' n''est pas dans l''intervalle (1..12) pour ', l printString , ' non retenu'.
      Screen default ringBell]
    ifFalse:
      [nbSaisies ←nbSaisies + 1.
      lieux at: nbSaisies put: l.
      mois at: nbSaisies put: m.
      hauteurs at: nbSaisies put: h].
    indice ←indice + 3.
    l ←(val at: indice) asString ]]]]! !

```

!PluvioSimple methodsFor: 'printing'!

printOn: aStream

```

"Append to the argument aStream a sequence of characters that
  identifies the receiver."
"Amplifies method in superclass to add name string."
"nextPutAll au lieu de print pour éviter les quotes"

```

```

aStream cr.
super printOn: aStream.
aStream cr; nextPut: $(.
aStream nextPutAll: 'NbLieux : ', NbLieux printString; cr.
nbSaisies = 0
ifTrue: [aStream nextPutAll: 'Aucune saisies']
ifFalse:
  [aStream nextPutAll: 'nbSaisies : ', nbSaisies printString; cr.
  aStream tab; nextPutAll: 'lieux'; tab; nextPutAll: 'mois'; tab; nextPutAll: 'hauteurs'.
  1 to: nbSaisies do: [:i | aStream cr; tab; nextPutAll: (lieux at: i) printString;
  tab; nextPutAll: (mois at: i) printString; tab; nextPutAll: (hauteurs at: i) printString]].
"self afficheSomme."
aStream nextPut: $)! !

```

!PluvioSimple methodsFor: 'displaying'!

afficheSomme

```

"Affiche les relevés mensuels"

```

```

(1 to: 12)
do: [:i | Transcript cr; show: 'Somme du mois de ', i printString; tab; show: (somme at: i) printString]! !

```

```

!PluvioSimple methodsFor: 'accessing'!

hauteurs
    ↑hauteurs!

hauteurs: aValue
    hauteurs ←aValue!

lieux
    ↑lieux!

lieux : aValue
    lieux ←aValue!

mois
    ↑mois!

mois: aValue
    mois ←aValue!

somme
    ↑somme!

somme: aValue
    somme ←aValue! !
"-----"!

PluvioSimple class
    instanceVariableNames: ''!

!PluvioSimple class methodsFor: 'examples'!

main
    "programme principal "
    "PluvioSimple main "

    | pluvio |
    pluvio ←self new saisieRelevésFichiers.
    pluvio printOn: Transcript.
    pluvio exploiterRelevés.
    pluvio afficheSomme.
    ↑pluvio! !

!PluvioSimple class methodsFor: 'instance creation'!

new
    ↑super new initialize! !

!PluvioSimple class methodsFor: 'initialize-release'!

initialize
    "PluvioSimple initialize "

    NbLieux ←30! !

PluvioSimple initialize !

```

Le code avec les bons exemples produit le résultat suivant :

```

donner votre fiche: lieu, mois, hauteur : paris, 12, 10
donner votre fiche: lieu mois hauteur : nantes, 10, 234
donner votre fiche: lieu mois hauteur : paris, 12, 10
donner votre fiche: lieu mois hauteur : paris, 12, 10
donner votre fiche: lieu mois hauteur : nantes, 5, 123
donner votre fiche: lieu mois hauteur : nantes, 10, 6
donner votre fiche: lieu mois hauteur : z
Relevés :
Fiche (1) lieu paris - mois 12 - hauteur 10
Fiche (2) lieu nantes - mois 10 - hauteur 234
Fiche (3) lieu paris - mois 12 - hauteur 10
Fiche (4) lieu paris - mois 12 - hauteur 10
Fiche (5) lieu nantes - mois 5 - hauteur 123
Fiche (6) lieu nantes - mois 10 - hauteur 6
Totaux :
somme du mois 1 est 0
somme du mois 2 est 0
somme du mois 3 est 0
somme du mois 4 est 0
somme du mois 5 est 123
somme du mois 6 est 0
somme du mois 7 est 0
somme du mois 8 est 0
somme du mois 9 est 0
somme du mois 10 est 240
somme du mois 11 est 0
somme du mois 12 est 30

```

Synthèse de la traduction

Le programme précédent a mis en évidence la traduction des fonctions de bases d'un programme structuré.

- Les structures de contrôle sont définies par des envois de messages aux instances des classes booléens, entiers ou blocs. Les blocs sont un élément essentiel dans la programmation avec Smalltalk.
- Les entrées/sorties utilisent l'interface graphique de Smalltalk (fenêtre, souris, clavier) dont nous reparlerons dans la partie consacrée au MVC.
- Les procédures sont assimilées à des méthodes.
- Les calculs sont réalisés par envois de messages.

Analyse du programme

Le programme de départ respectait certains critères de qualité d'un programme structuré, qui ont une influence sur son évolution (section 1).

- Utilisation de constantes pour mettre en évidence les paramètres du programme et rangement de ces constantes dans un seul endroit. La constante `nlieux` permet de paramétrer les tailles de tableaux et les contrôles associés. Si on avait mis simplement la valeur 20 partout, alors la modification du programme pour passer à 30 est à opérer sur l'ensemble du programme.

De ce fait, l'évolution numéro 2 se fait simplement en changeant valeur de la constante `nlieux`.

- Le programme principal est concis et construit à partir d'appels de procédures. Chaque procédure correspond bien à une activité indépendante et cohérente : saisie, calcul et affichage. La lisibilité en est accrue.

De ce fait, l'évolution numéro 1 se fait simplement en ajoutant une procédure `saisie_fichier`,

qui lit les fiches dans un fichier. Dans le programme principal, on demande à l'utilisateur le type de saisie à effectuer. Voici sa traduction en Smalltalk.

```
!PluvioSimple methodsFor: 'user'!
```

```
saisieRelevésFichiers
```

```
"Cette méthode saisit les relevés .
La saisie se termine par le lieu z"
"Quelques astuces : "
" curs ← Cursor currentCursor. "
" valeur ← (stream upTo: Character cr) tokensBasedOn: Character
space. "
" val = tableau interprété des valeurs lues : l = val [1], m = val [2], h =
val [3] "

| val aString fn fileStream indice l m h |
aString ← Dialog requestFileName: 'Please type a file name: ' default: 'releve.txt'.
aString isEmpty
ifTrue: [↑self]
ifFalse:
[fn ← aString asFilename.
fn definitelyExists ifFalse: [Filename errorReporter inaccessibleSignal
raiseErrorString: 'File does not exist']
ifTrue: [fn isReadable ifFalse: [Filename errorReporter inaccessibleSignal
raiseErrorString: 'File is readProtected']
ifTrue: [fn isDirectory
ifTrue: [Filename errorReporter inaccessibleSignal raiseErrorString:
'File is a directory']
ifFalse:
[Cursor read
showWhile:
[fileStream ← fn readStream.
val ← Scanner new scanTokens: fileStream contents.
fileStream close].
indice ← 1.
l ← (val at: indice) asString.
nbSaisies ← 0.
[1 = 'z' | (nbSaisies > NbLieux) | (indice + 2 > val size)]
whileFalse:
[m ← val at: indice + 1.
h ← val at: indice + 2.
m < 1 | m > 12
ifTrue:
[Transcript cr; show: 'Erreur mois ', m printString , '
n'est pas dans l'intervalle (1..12) pour ',
l printString , ' non retenu'.
Screen default ringBell]
ifFalse:
[nbSaisies ← nbSaisies + 1.
lieux at: nbSaisies put: l.
mois at: nbSaisies put: m.
hauteurs at: nbSaisies put: h].
indice ← indice + 3.
l ← (val at: indice) asString ]]]] !
```

- Le passage des paramètres, quoique un peu lourd en ce sens que qu'on aurait pu travailler directement sur les variables globales, garantit une bonne utilisation des procédures et une réutilisation de ces procédures dans un contexte différent.

- L'évolution numéro 3 est plus délicate car elle implique des modifications dans les déclarations, dans les procédures et dans le programme principal. On ajoute un nouveau tableau pour les auteurs. La saisie est modifiée (un paramètre supplémentaire) et une procédure est ajoutée pour le traitement statistique dont on ignore le contenu. Le programme principal comprend deux traitements : exploiter les relevés, calculer des statistiques. Leur coordination peut être séquentielle, alternative ou itérative.
- L'évolution numéro 4 induit une itération dans le programme principal sur la saisie des relevés (avec ou pas des exploitation intermédiaires). Il faut déplacer l'initialisation du nombre de saisies de la procédure de saisie vers le programme principal.
- L'évolution numéro 5b est simplement une recopie du programme en changeant des noms (hauteur devient température) et en changeant la procédure d'exploitation qui ne calcule plus une somme mais une moyenne (attention à la division par 0 s'il n'y a pas de relevés saisis).
- Les statistiques sur deux ans doublent les structures de données. On peut utiliser des matrices (année, lieu) et passer en paramètre des procédures l'année considérée. Cela ne pose pas de difficultés mais la modification porte sur tout le code.

Le programme initial pose aussi un certain nombre de problèmes.

- Le résultat est mono-bloc. Son organisation rend la lecture du code relativement aisée mais c'est moins le cas lorsque le programme figure sur quelques centaines de lignes. La lisibilité du programme est un critère essentiel à son évolution.
- L'utilisation de tableaux bornés, qui est une contrainte dans le langage cible, ajoute des éléments non négligeables dans le code : constante, bornes de tableaux, vérifications de dépassement... En Smalltalk, on peut disposer d'une variété de collections d'objets qui vont nous abstraire de ces contingences. On utilisera des ensembles (**Set**). L'évolution numéro 2 est implicitement résolue.
- L'évolution numéro 1 qui utilise les entrées/sorties sur fichiers (ou sur un autre support d'entrée-sortie) se fait par l'intermédiaire de flots de données.

Les autres évolutions sont facilitées lorsqu'on s'abstrait du code en mettant en évidence des concepts de l'application. En particulier, on distingue ici des fiches contenant des informations et un programme de calcul de résultats à partir de fiches. Nous étudions ce point de vue dans la section suivante.

1.1 Conception du programme en Smalltalk

Dissocier la notion de fiche de l'exploitation des résultats facilite l'évolution de programme.

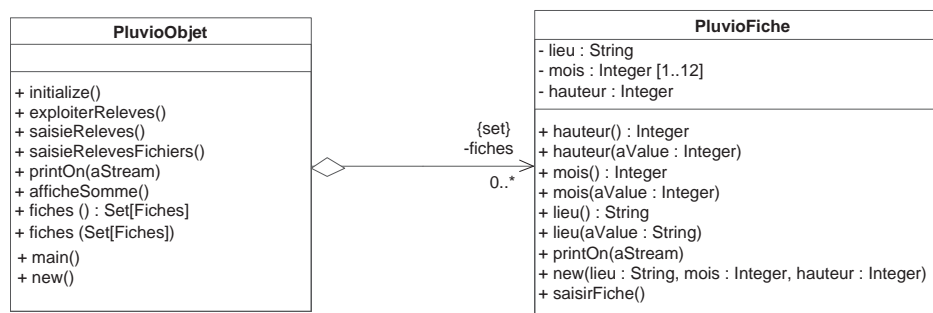


Figure 6 : Conception objet du programme *pluvio*

Le programme se présente schématiquement selon le diagramme de la figure 6. On remarque que l'interface des classes est largement simplifiée, ce qui facilite la lecture, le test, l'évolution et la réutilisation du code. L'agrégation est orientée, elle est implantée par une variable d'instance `fiches` contenant un ensemble d'objets `Fiche` dans la classe `PluvioObjet`.

- Une fiche regroupe l'ensemble des informations relative à un relevé. La saisie de la fiche est un traitement propre à cette fiche (une méthode). La prise en compte des évolutions numéro 3 et 5a se fait facilement par héritage (figure 7). Avec une classe abstraite `Fiche`, on peut mettre en œuvre un *pattern* `Fiche-Exploitation`, réutilisable dans différents contextes.
- La saisie fichier reste une opération générale car on souhaite centraliser l'utilisation du fichier.
- Les évolutions numéro 4, 5b et 6a concernent uniquement la classe `OPluvioObjet`, les modifications deviennent "locales" à cette classe.

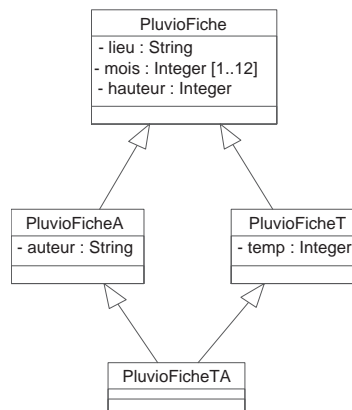


Figure 7 : Héritage des fiches

Détaillons maintenant le code des différentes classes.

Classe `PluvioFiche`

Listing 3.3 – Code Smalltalk des fiches pluviométriques

'From VisualWorks(R), Release 2.5 of September 26, 1995 on January 5, 1999 at 9:53:28 am'!

Object subclass: `#PluvioFiche`

instanceVariableNames: 'lieu mois hauteur '

classVariableNames: ''

poolDictionaries: ''

category: 'PluvioMetrie'!

`PluvioFiche` comment:

'Cette classe implante en objet le programme pluvio.

Il s'agit d'une écriture Smalltalk avec conception en classes d'un programme impératif.

Le programme est implanté par une classe `PluvioObjet` avec les méthodes de saisie, addition, exploitation et affichage. Des variantes sont possibles du point de vue des interfaces.

Les variables d'instance sont :

lieu : String

mois : Integer 1..12

hauteur : Integer

La principale difficulté est la saisie des entiers.

Nous passons par le scanner Smalltalk qui rend un tableau d'arguments interprétés.

Une autre possibilité consiste à utiliser une nouvelle interface avec ds champs (InputField) dont les types sont précisés.

'!

```

!PluvioFiche methodsFor: 'accessing'!

hauteur
    ↑hauteur!

hauteur: aValue
    hauteur ←aValue!

lieu
    ↑lieu!

lieu: aValue
    lieu ←aValue!

mois
    ↑mois!

mois: aValue
    mois ←aValue!

!PluvioFiche methodsFor: 'printing'!

printOn: aStream
    "Append to the argument aStream a sequence of characters that
    identifies the receiver ."
    "Amplifies method in superclass to add name string."
    "nextPutAll au lieu de print pour éviter les quotes "

    aStream cr.
    super printOn: aStream.
    aStream cr; nextPut: $(.
    aStream nextPutAll: lieu; tab; nextPutAll: mois printString; tab; nextPutAll: hauteur printString.
    aStream nextPut: $)! !
"-----"!

PluvioFiche class
    instanceVariableNames: ''!

!PluvioFiche class methodsFor: 'instance creation'!

newLieu: aString mois: anInteger hauteur: anInteger1
    "créé une nouvelle fiche (sans contrôle)"

    | fiche |
    fiche ←self new.
    fiche lieu: aString.
    fiche mois: anInteger.
    fiche hauteur: anInteger1.
    ↑fiche!

```



```

saisirFiche
  " saisit une fiche au clavier "

  | res val |
  res ←Dialog
      request: 'Donnez votre fiche : lieu mois hauteur (annuler pour ne pas en saisir)'
      initialAnswer: 'z'
      onCancel: ['z']. "Le problème est la lecture des entiers "
  val ←Scanner new scanTokens: res.
  (val at: 1) asString == 'z'
  ifTrue: ["tableau interprété des valeurs lues : l = val [1], m = val [2], h =
    val [3]"
    ↑self
      newLieu: 'z'
      mois: 0
      hauteur: 0]
  ifFalse:
    [| m h |
    m ←val at: 2.
    [m < 1 | m > 12]
    whileTrue:
      [Dialog warn: 'Erreur mois (1..12)'.
      m ←(Scanner new scanTokens: (Dialog request: 'Donnez le mois' initialAnswer: '1'))
      at: 1].
    h ←val at: 3.
    [h < 0]
    whileTrue:
      [Dialog warn: 'Erreur hauteur négative '.
      h ←(Scanner new scanTokens: (Dialog request: 'Donnez la hauteur' initialAnswer: '1'))
      at: 1].
    ↑self
      newLieu: (val at: 1)
      mois: m
      hauteur: h]! !

```

Classe PluvioObjet

Listing 3.4 – Code Smalltalk dde calcul pluviométrique

'From VisualWorks(R), Release 2.5 of September 26, 1995 on January 5, 1999 at 9:53:34 am'!

```

Object subclass: #PluvioObjet
  instanceVariableNames: 'fiches '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PluvioMetrie'!

```

PluvioObjet comment:

'Cette classe implante en objet le programme pluvio.

Il s'agit d'une écriture Smalltalk avec conception en classes d'un programme impératif.

Le programme est implanté par une classe PluvioObjet
avec les méthodes de saisie, addition, exploitation et affichage.
Des variantes sont possibles du point de vue des interfaces.

Les variables d'instance sont :

fiches : Set of PluvioFiche

Nous n'utilisons pas de tableaux et évitons ainsi bon nombre de contrôles.
 La taille des ensembles est dynamique.
 Il n'y a pas d'ordre dans les fiches.

La somme est implantée par une méthode d'instance mais il est possible de la stocker comme variable d'instance pour gagner en efficacité.
 Toutefois cela implique une vérification cohérence permanente entre les fiches et la somme.
 La difficulté de saisie des fiches est reportée dans la classe PluvioFiche.

La principale difficulté est la saisie des entiers.
 Nous passons par le scanner Smalltalk qui rend un tableau d'arguments interprétés.
 Une autre possibilité consiste à utiliser une nouvelle interface avec des champs (InputField) dont les types sont précisés.

Optimisation : lecture par fichiers.

Les interfaces de sorties peuvent être améliorées en utilisant l'exemple des histogrammes.!

!PluvioObjet methodsFor: 'printing'!

printOn: aStream

"Append to the argument aStream a sequence of characters that identifies the receiver."
"Amplifies method in superclass to add name string."
"nextPutAll au lieu de print pour éviter les quotes"

```
aStream cr.
super printOn: aStream.
aStream cr; nextPut: $(.
fiches size == 0
  ifTrue: [aStream nextPutAll: 'Aucune saisies']
  ifFalse :
    [aStream nextPutAll: 'nbSaisies : ', fiches size printString; cr.
    aStream tab; nextPutAll: 'lieux'; tab; nextPutAll: 'mois'; tab; nextPutAll: 'hauteurs'.
    fiches do: [:fiche | fiche printOn: aStream]].
aStream nextPut: $)! !
```

!PluvioObjet methodsFor: 'initialize-release'!

```
initialize
  fiches ←Set new! !
```

!PluvioObjet methodsFor: 'accessing'!

```
fiches
  ↑fiches!
```

```
fiches: aSetOfFiche
  fiches ←aSetOfFiche! !
```

!PluvioObjet methodsFor: 'user'!

exploiterRelevés

"Cette méthode calcule les sommes à partir des relevés"

```
| somme |
```

```
somme ←Array new: 12 withAll: 0.
fiches do: [: fiche | somme at: fiche mois put: (somme at: fiche mois)
           + fiche hauteur].
↑somme!
```

saisieRelevés

```
"Cette méthode saisit les relevés .
La saisie se termine par le lieu z"
```

```
| fiche |
fiche ←PluvioFiche saisirFiche .
[fiche lieu = 'z']
whileFalse:
  [fiches add: fiche .
   fiche ←PluvioFiche saisirFiche ]!
```

saisieRelevésFichiers

```
"Cette méthode saisit les relevés .
La saisie se termine par le lieu z"
"Quelques astuces : "
" curs ←Cursor currentCursor ."
" valeur ←(stream upTo: Character cr) tokensBasedOn: Character
space." "probleme du retour chariot"
" val = tableau interprété des valeurs lues : l = val [1], m = val [2], h =
val [3] "
```

```
| val aString fn fileStream indice |
aString ←Dialog requestFileName: 'Please type a file name: ' default: 'releve.txt'.
aString isEmpty
  ifTrue: [↑self]
  ifFalse :
    [fn ←aString asFilename.
     fn definitelyExists ifFalse: [Filename errorReporter inaccessibleSignal
                                   raiseErrorString: 'File does not exist']
     ifTrue: [fn isReadable ifFalse: [Filename errorReporter inaccessibleSignal
                                      raiseErrorString: 'File is readProtected']
             ifTrue: [fn isDirectory
                      ifTrue: [Filename errorReporter inaccessibleSignal
                               raiseErrorString: 'File is a directory']
                      ifFalse :
                        [Cursor read
                         showWhile:
                           [fileStream ←fn readStream.
                            val ←Scanner new scanTokens: fileStream contents.
                             fileStream close].
                         indice ←1.
                         [(val at: indice) asString = 'z' | (indice + 2 > val size)]
                         whileFalse:
                           [| fiche |
                            fiche ←PluvioFiche
                               newLieu: (val at: indice) asString
                               mois: (val at: indice + 1)
                               hauteur: (val at: indice + 2).
                            fiche mois < 1 | (fiche mois > 12)
                            ifTrue:
                              [Transcript cr; show: 'Erreur mois ', fiche mois printString ,
                               ' n'est pas dans l'intervalle (1..12) pour ',
                               fiche hauteur printString , ' non retenu'.
```

```

        Screen default ringBell]
        ifFalse: [ fiches add: fiche ].
        indice ← indice + 3]]]]]! !

!PluvioObjet methodsFor: 'displaying'!

afficheSomme
    "Affiche les relevés mensuels"

    | somme |
    somme ← self exploiterRelevés.
    (1 to: 12)
        do: [:i | Transcript cr; show: 'Somme du mois de ', i printString; tab;
            show: (somme at: i) printString]! !

"-----"!

PluvioObjet class
    instanceVariableNames: ''!

!PluvioObjet class methodsFor: 'examples'!

main
    "programme principal "
    "PluvioObjet main"

    | pluvio |
    pluvio ← self new saisieRelevésFichiers .
    pluvio printOn: Transcript.
    pluvio afficheSomme.
    ↑pluvio! !

!PluvioObjet class methodsFor: 'instance creation'!

new
    ↑super new initialize! !

```

1.2 Bilan

À travers ce petit exemple, nous avons mis en évidence qu'un découpage en objet permet de définir une meilleure abstraction du code. Ce découpage facilite la lisibilité, la conception, le test des différents modules. La cohésion des objets et l'héritage facilitent l'extension de l'application et la réutilisation de code. Noter aussi que la notation UML est un bon outil de documentation du code.

2 Quelques hiérarchies de classes

Nous présentons très sommairement quelques hiérarchies de classes importantes en Smalltalk-80. Nous les étudions dans l'environnement lui même.

2.1 Collection

La hiérarchie des collections implante les types de données relatifs aux structures de données de haut niveau de la programmation : tableaux, listes, ensembles, multi-ensembles, collections ordonnées ou triées...

La collection est un outil fondamental d'abstraction en modélisation. En programmation à objets, toutes les implantations imaginables de ces types de données sont réalisables par adaptation de la hiérarchie suivante.

- Les classes Smalltalk relatives aux conteneurs sont organisées en une structure arborescente descendant de **Collection**.
- Les collections sont implicitement génériques et hétérogènes.
- Quelques exemples :

Collection : classe abstraite de base.

Set : ensembles.

Bag : multi-ensembles.

ArrayedCollection : tableaux.

OrderedCollection : liste.

Dictionary : fonction de correspondance, les "*maps*".

Les collections sont spécialisées pour les caractères ou les pixels.

En C++ ou Java l'organisation est plus parcellaire.

Les collections en Smalltalk sont autant dirigées vers le programmeur que le système lui-même.

- Structures de données classiques : ensembles, listes, tableaux, dictionnaires (= *map*)
- Types obtenus par héritage : **String**, **Symbol**,
- Collections pour le système : dictionnaires de méthodes **MethodDictionary**, matrices de pixels, **ObjectRegistry**, **SignalCollection**, **BinaryStorageBytes** ...
- Collections personnalisables, homogènes, hétérogènes
- Itérateurs intégrés (OCL s'en est inspiré)

Object ()

Collection ()

Bag ('contents')

KeyedCollection ()

ColorPreferencesCollection ()

ChainedColorPreferences ('base' 'node')

ColorPreferencesDictionary ('preferences' 'constantCodeArray')

LookPreferences ('foregroundColor' 'backgroundColor' 'selectionForegroundColor' 'selectionBackgroundColor' 'borderColor' 'hiliteColor' 'shadowColor')

MethodDictionary ('tally' 'valueArray')

Palette ()

ColorPalette ()

FixedPalette ('redShift' 'redMask' 'greenShift' 'greenMask' 'blueShift' 'blueMask')

MappedPalette ('hasColor' 'palette' 'inverseMap' 'mapResolution' 'size')

MonoMappedPalette ('whitePixel' 'blackPixel')

CoveragePalette ('maxPixelValue')

StopsDictionary ('values' 'size')

LensContainer ('lensSession' 'transporter' 'type')

LensBaseContainer ('cache')

LensCompositeContainer ('components')

SequenceableCollection ()

ArrayedCollection ()

Array ()

BOSSReaderMap ('baseIndex' 'storage')

DependentsCollection ()

ScannerTable ('value0' 'defaultValue' 'letterValue' 'digitValue' 'separatorValue')

```

SegmentedCollection ('compressed')
  LargeArray ()
  LargeWordArray ()
CharacterArray ()
String ()
  ByteEncodedString ()
  ByteString ()
  ISO8859L1String ()
  MacString ()
  OS2String ()
  GapString ('string' 'gapStart' 'gapSize')
  Symbol ()
  ByteSymbol ()
  TwoByteSymbol ()
  TwoByteString ()
  Text ('string' 'runs')
IntegerArray ()
ByteArray ()
  BinaryStorageBytes ()
  BOSSBytes ()
  WordArray ()
List ('dependents' 'collection' 'limit' 'collectionSize')
  DependentList ()
  LensStreamList ('stream' 'chunkSize')
RunArray ('runs' 'values' 'cacheRun' 'cacheRunStart')
TableAdaptor ('dependents' 'baseCollection' 'adaptors' 'columnSize'
  'transposed')
TwoDList ('dependents' 'collection' 'rows' 'columns' 'transposed')
WeakArray ('dependents')
Interval ('start' 'stop' 'step')
  SlidingInterval ()
  TextLineInterval ('internalSpaces' 'paddingWidth')
LinkedList ('firstLink' 'lastLink')
  HandlerList ()
  Semaphore ('excessSignals')
OrderedCollection ('firstIndex' 'lastIndex')
  FontDescriptionBundle ()
  LinkedOrderedCollection ('backup')
  SortedCollection ('sortBlock')
  SortedCollectionWithPolicy ('sortPolicy')
  SPActiveLines ('y')
  SPSortedLines ('currentTag' 'startPoint' 'currentPoint' 'edgeHead'
  'edgeTail')
Set ('tally')
Dictionary ()
  CEnvironment ()
  IdentityDictionary ('valueArray')
  PropertyListDictionary ('basicSize')
  WeakDictionary ('executors' 'accessLock')
  HandleRegistry ()
  ExternalRegistry ()
  WeakAssociationDictionary ()
  LinkedWeakAssociationDictionary ('backup')
  ExternalDictionary ()
LensLinkedDictionary ('bucketTally' 'hashTable')
  LensProtectedLinkedDictionary ('accessLock')
LensRegistry ('keyArray')
  LensWeakRegistry ('accessLock')

```

```

    LensObjectRegistry ()
    PoolDictionary ('definingClass')
    SystemDictionary ('organization')
IdentitySet ()
    ObjectRegistry ()
    SignalCollection ()

```

2.2 Component

Le composant visuel est la brique de base de l'interface graphique. Il est détaillé dans la section 5. Il comprend des formes, des zones textuelles, des vues imbriquées... et des boutons enfichables.

```

Object ()
VisualComponent ()
    DragHandle ('isPrimary' 'wrapper' 'subject' 'selector' 'block' 'extent'
                'dragModeBlock')
    Icon ('image' 'mask')
    Label ('text' 'attributes' 'width' 'offset' 'needsScan')
        AlignmentLabel ('compositionWidth' 'alignment')
        LabelAndIcon ('icon' 'gap')
    LDMVisualTreeConnection ('orientation' 'parent' 'children' 'preferredBounds')
    LDMBroomConnection ('builder' 'refreshArea' 'model')
    OpaqueImage ('figure' 'shape')
    OpaqueImageWithEnablement ('inactiveFigure' 'inactiveShape' 'isActive')
    PixelArray ()
    CachedImage ('image' 'retainedMedium')
    Image ('bits' 'width' 'height' 'depth' 'bitsPerPixel' 'palette'
           'maxPixelValue' 'rowByteSize')
        Depth16Image ()
        Depth1Image ()
        Depth24Image ()
        Depth2Image ()
        Depth32Image ()
        Depth4Image ()
        Depth8Image ()
    RowVisual ('bounds' 'object' 'index' 'inhibit' 'descriptors' 'frozen'
              'frozenTranslation' 'left' 'right' 'desc' 'view' 'renderers'
              'models')
    RowLabelVisual ()
    TextLines ('textStyle')
        ComposedText ('text' 'compositionWidth' 'compositionHeight' 'wordWrap'
                     'fontPolicy' 'lineTable' 'fitWidth')
        InputFieldComposedText ('realWidth' 'passwordChar')
    TextList ('width' 'lines')
    VisualBlock ('block' 'bounds' 'paint')
    VisualPart ('container')
        CompositePart ('components' 'preferredBounds')
            BorderDecorator ('component' 'widgetFlags' 'policy')
                TableDecorator ('columnLabelComponent' 'rowLabelComponent'
                                'borderColumnLabels' 'borderRowLabels' 'tableInterface'
                                'etchedLabelBorders')
            ComboBoxView ('editor' 'menuButton' 'dropDownWindowModel')
            ComposingComposite ('leftSpace' 'spaceBetweenItems' 'extraSpaceTop'
                               'extraSpaceBottom' 'extraSpaceBetweenLines')
            MenuBar ('menuChannel' 'performer' 'menuButtons' 'nextMenuButton')
                CUAMenuBar ()
                MacMenuBar ()

```

```

    MotifMenuBar ()
    Win3MenuBar ()
CUAScrollBar ()
DependentComposite ('model')
    CompositeView ('controller')
        LDMBrowserBodyView ('decorator')
        LDMCompositeView ()
            LDMGraphCompositeView ('builder')
        LDMElementView ('dependents' 'element' 'hasIn' 'hasOut'
            'povIcons' 'firstPovIconX' 'touchFlag' 'selected' 'label'
            'recycle')
    MenuItemView ('menuItemViews' 'parentMenuItemView' 'selectedValue'
        'selectionFinal' 'highlightedMenuItemView' 'usedSelection
        Memory' 'parentMenuBarButtonView')
    CUAMenuView ('shortcutColumnIndent')
    MacMenuView ()
    MotifMenuView ('usedAccessCharacters' 'shortcutColumnIndent'
        'backgroundColor')
    Win3MenuView ('shortcutColumnIndent')
    UIPainterView ()
NoteBookComposite ('binder' 'tabBar' 'bottomTabBar' 'subCanvas')
ReComposingComposite ()
SubCanvas ()
DependentPart ('model')
View ('controller')
    AutoScrollingView ('scrollOffset')
    ComposedTextView ('displayContents' 'startBlock' 'stopBlock'
        'selectionShowing' 'displaySelection')
        TextCollectorView ()
        TextEditorView ('state')
            InputFieldView ('inset' 'converter' 'editTextCache')
                ComboBoxInputFieldView ()
    TableView ('state' 'table' 'columnInfo' 'rowInfo' 'selection
        Channel' 'selectionIndex' 'targetIndex' 'grid' 'column
        Widths' 'rowHeights' 'visualBlock' 'textStyle'
        'displayStringSelector' 'showCGrid' 'showRGrid'
        'strokedSelection' 'isSlave' 'lpDictionary')
        GeneralSelectionTableView ('selectionStyle' 'zOrder')
    BitView ('pixmap' 'policy' 'scale' 'gridding')
        ColorBitView ()
    BooleanWidgetView ('transitionHolder' 'latestValue')
        ActionButton ('onImage' 'offImage')
        LabeledBooleanView ('label' 'onImage' 'offImage')
            LDMArrowView ()
    ColoredArea ('select' 'wasSelected')
    DirectBitView ('policy')
    LauncherView ('selectionIndex')
    MenuItemView ('menuView')
        CUAMenuItemView ('composedLabel' 'composedCharacter' 'access
            Character' 'accessIndicator' 'accessIndicatorThickness')
        MacMenuItemView ('composedLabel' 'composedCharacter')
        MotifMenuItemView ('composedLabel' 'composedCharacter' 'access
            Character' 'accessIndicator' 'accessIndicatorThickness')
        Win3MenuItemView ('composedLabel' 'composedCharacter' 'access
            Character' 'accessIndicator' 'accessIndicatorThickness')
    NotifierView ()
    Scrollbar ('marker' 'alignment')
        EmulationScrollBar ('dragging')

```



```

CUAScrollBarSlider ()
EmulationFixedThumbScrollBar ()
  MacScrollBar ('isActive' 'windowActive')
  Win3ScrollBar ('isActive')
MotifScrollBar ()
SimpleView ('state')
  BasicButtonView ('isInTransition' 'latestValue' 'referenceValue')
  ComboBoxButtonView ('editorView' 'list' 'forceScrollBar'
    'maxLines' 'textStyle' 'lastSelectionIndex' 'multiChar
    Search' 'continuousAccept')
  CUAComboboxButtonView ()
  MacComboBoxButtonView ()
  MotifComboBoxButtonView ()
  Win3ComboBoxButtonView ()
LabeledButtonView ('label' 'textStyle')
  CheckButtonView ()
    CUACheckButtonView ()
    DefaultLookCheckButtonView ()
    MacCheckButtonView ()
    MotifCheckButtonView ()
    Win3CheckButtonView ()
  PushButtonView ()
    ActionButtonView ()
      CUAActionButtonView ()
      MacActionButtonView ()
      MotifActionButtonView ()
      UndecoratedActionButtonView ('hiliteSelection')
      Win3ActionButtonView ()
    MenuBarButtonView ('menuHolder' 'menuBar')
      CUAMenuBarButtonView ('accessCharacter')
      MacMenuBarButtonView ()
      MotifMenuBarButtonView ('accessCharacter')
      Win3MenuBarButtonView ('accessCharacter')
    MenuButtonView ('currentChoice' 'defaultChoice'
      'menu' 'vcBlock')
      CUAMenuButtonView ()
      MacMenuButtonView ()
      MotifMenuButtonView ()
      UndecoratedMenuButtonView ('hiliteSelection')
      Win3MenuButtonView ()
  RadioButtonView ()
    CUARadioButtonView ()
    DefaultLookRadioButtonView ()
    MacRadioButtonView ()
    MotifRadioButtonView ()
    Win3RadioButtonView ()
  ScrollerButtonView ()
    CUAScrollerButtonView ('direction')
    MotifScrollerButton ('directionKey')
    VisualPairButton ('offImage' 'onImage')
ScrollingView ('scrollOffset')
  SelectionView ('sequence' 'selectionChannel' 'selection
    Index' 'targetIndex' 'grid' 'textStyle'
    'displayStringSelector')
  DataSetView ('columnDescriptors' 'right' 'starts'
    'numFrozen' 'editor' 'editorWrapper' 'editValue'
    'editCellChannel' 'showingSelection'
    'useHiliteSelection' 'rowSelectorIndex' 'topOffset')

```

```

        'verticalPolicy' 'horizontalPolicy' 'proxy')
    PaintedDataSetView ('editMode')
    SequenceView ('visualBlock' 'selectedVisualBlock'
        'measureWidth' 'cachedWidth')
    ComboBoxListView ('buttonController' 'editorController')
    MultiSelectionSequenceView ('selections'
        'lastSelectionIndex')
    TabBarView ('visualBlock' 'leftSpine' 'tabScrollPosition'
        'tabScrollPositionKnown' 'localEnabled' 'hasFocus')
    HorizontalTabBarView ('tabWidth' 'cacheIndex'
        'cacheBox' 'cacheVisual')
    VerticalTabBarView ()
    SliderView ('axis' 'markerBlock' 'markerLength' 'marker'
        'color1' 'color2' 'markerBorder' 'rangeMap')
    MacSliderView ()
    Win3SliderView ()
    MotifMenuItemSeparatorComponent ('menuView')
    SimpleComponent ('state')
    GroupBox ('label' 'border' 'textStyle')
    PassiveLabel ('label' 'textStyle' 'margin')
    VisualBinderComponent ('binderPosition' 'binderThickness')
    VisualDivider ('orientation' 'lineWidth' 'etched')
    VisualRegion ('extent' 'lineWidth' 'isElliptical')
    Wrapper ('component')
    GeometricWrapper ()
    FillingWrapper ()
    StrokingWrapper ('lineWidth')
    GraphicsAttributesWrapper ('attributes')
    GridWrapper ('showGrid' 'griddedHorizontally' 'griddedVertically'
        'grid' 'gridPaint')
    PassivityWrapper ('controlActive' 'visuallyActive' 'visible')
    ReversingWrapper ('reverse' 'offset')
    StrikeOutWrapper ()
    ScalingWrapper ('scale')
    TranslatingWrapper ('origin')
    LayoutWrapper ('layout')
    BoundedWrapper ('extent')
    BorderedWrapper ('insetDisplayBox' 'border' 'inset'
        'insideColor')
    MenuBarWrapper ()
    BoundingWrapper ()
    LDMAbstractElementWrapper ('depth' 'pass' 'children' 'isRoot')
    LDMGraphElementWrapper ('builder')
    LDMListElementWrapper ('reference')
    ScrollWrapper ('dependents' 'preferredBoundsBlock')
    DataSetScrollWrapper ()
    SlaveScrollWrapper ('vertical' 'horizontal')
    WidgetStateWrapper ('widgetState')
    WidgetWrapper ('widget' 'widgetState' 'decorator' 'dependents'
        'dropTarget')
    SpecWrapper ('spec')

```

2.3 Controller

Le contrôleur est la seconde brique de base de l'interface graphique. Il est détaillé dans la section 5. Il sert à l'interaction avec l'utilisateur : souris et clavier principalement, mais il comprend aussi la gestion du contrôle entre les différentes fenêtres et vues de l'interface.

```

Object ()
  Controller ('model' 'view' 'sensor')
    ComboBoxButtonController ()
  ControllerWithMenu ('menuHolder' 'performer')
    BitEditor ('foreground' 'background' 'imageHasChanged')
      ColorBitEditor ('currentColor')
    DataSetController ('keyboardProcessor' 'outside' 'dispatcher'
      'callbackLock' 'rowSelect' 'clicked')
      PaintedDataSetController ()
    LDMCompositeViewController ()
    LDMElementViewController ('charged')
    ModalController ('currentMode')
      UIPainterController ('showGrid' 'gridStep' 'griddedHorizontally'
        'griddedVertically' 'fenced' 'primarySelection' 'selections'
        'handles' 'canvasHasChanged')
    ParagraphEditor ('beginTypeInIndex' 'emphasisHere' 'dispatchTable'
      'charComposer' 'textHasChanged')
    TextEditorController ('keyboardProcessor' 'keyboardHook' 'readOnly'
      'accepted' 'autoAccept' 'continuousAccept' 'tabMeansNextField'
      'dispatcher')
      InputBoxController ('tabBlock' 'crBlock' 'maxChars')
        ComboBoxInputBoxController ('buttonController')
    SequenceController ('keyboardProcessor' 'outside' 'searchString'
      'keyboardHook' 'doStringSearching' 'dispatcher'
      'dragDropCallbacks' 'selectOnDownWithDrag')
    ComboBoxListController ('continuousAccept' 'closeChannel'
      'multiCharSearch')
    EmulatedSequenceController ()
  DataSetControllerProxy ('keyboardProcessor' 'masterController' 'direction')
  LauncherController ()
  MenuBarButtonController ()
  MenuItemController ('controlType' 'dispatcher')
  MenuController ('scrollWrapper' 'state')
    MenuAsPopUpController ()
    MenuAsSubMenuController ()
    MenuFromMenuBarController ()
    MenuFromMenuItemController ()
  MenuItemController ()
  NoController ()
  ScrollbarController ('cursors' 'evenIfKeyPressed')
    EmulationScrollBarController ()
  SelectController ('keyboardProcessor' 'keyboardHook' 'dispatcher')
    BasicButtonController ()
      ToggleButtonController ()
      RadioButtonController ()
      TriggerButtonController ()
    ColoredAreaController ()
    SliderController ()
    TabBarController ('outside')
  StandardSystemController ('locked')
    ApplicationStandardSystemController ()
      ApplicationDialogController ()
        DropDownListController ('buttonController' 'mouseHasBeen
          Released' 'listHasTakenControl')
      UIPainterSystemController ()
  WidgetController ('cursor' 'activeAccessor' 'evenIfKeyPressed'
    'controlBlock' 'enableYellowButton')

```

2.4 Code

La hiérarchie de code correspond aux méthodes et blocs. Y sont décrits les descriptions et évaluation de blocs et d'envois de messages.

```
Object ()
  InstructionClient ()
    InstructionPrinter ('method' 'stream' 'inStream')
    InstructionStream ('method' 'pc')
      CodeRegenerator ('parent' 'codeGenerator' 'labels' 'lastPC' 'locals'
        'stack' 'copiedValues' 'canCopy' 'oldInstVarNames' 'pcStack'
        'pcLabels' 'repressNextPop')
      Context ('sender' 'receiver' 'stackp' 'stack')
        BlockContext ()
        MethodContext ()
      Decompiler ('builder' 'instVars' 'localVars' 'copiedVars' 'tempCount'
        'stack' 'statements' 'limit' 'primitive' 'primErrorCode'
        'lastPc' 'exit' 'lastJumpPc' 'hasValue' 'loopDepth'
        'lastAssignment' 'lastAssignVar' 'lastAssignPc')
%
```

2.5 Behavior

La hiérarchie de comportement explicite la description des classes et des méta-classes. Toutes les méta-classes héritent de la classe **Behavior**.

```
Object ()
  Behavior ('superclass' 'methodDict' 'format' 'subclasses')
  ClassDescription ('instanceVariables' 'organization')
  Class ('name' 'classPool' 'sharedPools')
  ... all the Metaclasses ...
  Metaclass ('thisClass')
```

2.6 Process

Les processus sont la réalisation des traitement en Smalltalk. Ils sont difficiles à gérer car font partie du noyau du système. Donc à manipuler avec d'extrêmes précautions.

```
Object ()
  Link ('nextLink')
  Process ('suspendedContext' 'priority' 'myList' 'interruptProtect')
```

2.7 Model

La hiérarchie des modèles est le troisième volet de la triade MVC. Un modèle contient le coeur de l'application *i.e.* sans l'interface. Ce coeur est enrichi (ou surchargé) de liens vers les éléments de l'interface.

```
Object ()
  Model ('dependents')
  ApplicationModel ('builder')
    AdHocQueryTool ('connection' 'session' 'answerStream' 'queryModel'
      'tableModel' 'whichDriverModel' 'connectedStatusHolder'
      'executeStatusHolder' 'moreAnswersStatusHolder' 'usernameModel'
      'passwordModel' 'environmentModel' 'blockFactorModel')
  Browser ('organization' 'category' 'className' 'meta' 'protocol'
    'selector' 'textMode' 'categoryList' 'classList' 'protocolList')
```

```

        'selectorList' 'textValue' 'metaHolder')
Debugger ('context' 'contextList' 'receiverInspector'
        'contextInspector' 'shortStack' 'sourceMap' 'sourceCode'
        'processHandle' 'mayProceed' 'breakPC' 'oldCursor' 'label')
HierarchyBrowser ('myClass')
MethodListBrowser ('methodList' 'methodName' 'selInMethodList'
        'initialSelection')
ParcelBrowser ('parcelFilterOn' 'parcelNameList' 'currentParcel'
        'parcelMenu' 'categoryMenu' 'classMenu' 'protocolMenu'
        'messageMenu' 'textMenu')
ChangeList ('value' 'listName' 'changes' 'selectionIndex' 'list'
        'filter' 'removed' 'filterList' 'filterKey' 'changeDict'
        'doItDict' 'checkSystem' 'fieldList' 'selectionInList')
CodingAssistant ('classNameChannel' 'instVarList' 'readAccessing'
        'writeAccessing' 'dependency')
ExamplesBrowser ('page' 'examples' 'code' 'title' 'exampleNumber')
FileBrowser ('fileName' 'list' 'myPattern' 'selectionState' 'autoRead'
        'lastModified' 'currentFileEncoding' 'defaultEncodings')
HelpBrowser ('pageNumber' 'text' 'element' 'selectionInList' 'history'
        'bookmarks' 'searchScope' 'ignoreCase' 'useWildcards'
        'searchString' 'finderBuilder' 'searchStatus' 'cancelSearch'
        'library' 'menuBar' 'aboutHelpText' 'helpDir' 'pathnames' 'book'
        'chapter' 'page' 'container' 'historyBuilder')
Inspector ('object' 'field' 'fieldList')
    ChangeSetInspector ()
    CompiledCodeInspector ()
    ContextInspector ('tempNames' 'tempIndex')
    DictionaryInspector ()
    SequenceableCollectionInspector ()
    OrderedCollectionInspector ()
LabelConstructor ('labeledValueHolder' 'suppliedByBuilder'
        'parentBuilderBlock' 'fieldMenuBlock')
LensApplicationModel ('parent' 'children' 'session')
    LensDataManager ('rows' 'row' 'trigger' 'rowCount' 'isEditing'
        'isCreating' 'isDirty' 'lockPolicy')
    LensMainApplication ()
    LensTemporaryMain ()
LensApplicationSpecEditor ('spec' 'graphView' 'dataModel' 'adtype'
        'typesMenu' 'fieldsDict' 'fields' 'there' 'className' 'generator'
        'ldmChanged' 'newFields' 'templatesMenuHolder' 'accept'
        'subBuilder')
LensApplicationStructureView ('graphView' 'focus' 'selection' 'finder')
LensBrowsingToolModel ()
    LensGraphView ('bm' 'focus' 'perspective')
LensEditor ('selectedDataModelHolder' 'selectedType' 'ldmView'
        'referenceButton' 'toReference' 'generator' 'valueClassesMenu'
        'mainMessage' 'mappingEditor')
LensMappingEditor ('graphView' 'toMap' 'mapButton' 'type' 'selVar'
        'selCol' 'columnTypeMenu' 'editor' 'lastType')
ParcelList ('parcelInfo' 'parcelList' 'menuBar' 'listMenu' 'selectedParcel')
QueryEditor ('ldm' 'tables' 'from' 'answer' 'where' 'orderBy' 'groupBy'
        'unique' 'distinct' 'lock' 'current' 'oldCurrent' 'tableView'
        'menusValue' 'selectedTables' 'changing' 'targetClass'
        'targetSelector' 'className' 'pattern' 'parameters' 'fullObjects'
        'instanceVariablesMenu' 'tmpCode' 'functionsMenu' 'mode' 'ownRow'
        'editorKind' 'menuAccessorName' 'saved' 'useStreaming')
SimpleDialog ('close' 'accept' 'cancel' 'preBuildBlock' 'postBuildBlock'
        'postOpenBlock' 'escapeIsCancel' 'parentView')

```

```

LensApplicationCreationDialog ('spec' 'dataModel' 'addtype' 'typesMenu'
    'generator' 'entities' 'subBuilder' 'lastAppType' 'fixedAppType')
LensDataModelGenerator ('lensDataModelHolder' 'className' 'super
    className' 'category' 'extractedColumnList' 'applicationNameAdaptor'
    'dialect' 'lensPolicy' 'selectorNameAdaptor' 'catalogLens'
    'defaultCategory' 'oldDataModel')
LensKeyEditor ('type' 'nonKeyVariables' 'keyVariables' 'message1'
    'message2')
LensReferenceNameDialog ('nameSelector' 'toEntity' 'name1'
    'replaced' 'name2' 'referenceEntities' 'name3' 'fromEntity')
LensTablesSelector ('editor' 'columnsList' 'columnType'
    'columnCanBeNull' 'tablesList' 'pattern')
SimpleHelp ('helpString')
SimpleListEditor ('list' 'currentName' 'validationBlock' 'changedBlock')
SpecModel ('specChannel')
IntegratedSpecModel ('propertiesTool' 'readMode' 'interruptedRead')
    ColorToolModel ('cubesBrightness' 'cubeColors' 'colorValue'
        'colorName' 'lookPreferences')
    DataSetCallbacksSpecModel ('focusInSelector' 'valueChangeSelector'
        'focusOutSelector' 'currentColumn' 'selectionChannel'
        'requestFocusInSelector' 'requestValueChangeSelector'
        'requestFocusOutSelector' 'trigger')
    DataSetSpecColumnDetailsModel ('selectionChannel' 'currentColumn'
        'modelGen' 'columnType' 'changedBlock' 'typeChoices' 'typeSize'
        'typePopup' 'typeAlign' 'typeMenu' 'typeFont' 'typeDataType'
        'typeDataFormat' 'defaultFormats' 'lock' 'readSelector'
        'printSelector')
    DataSetSpecColumnModel ('selectionChannel' 'currentColumn' 'label'
        'width' 'labelFont' 'modelGen' 'columnType' 'changedBlock'
        'labelIsImage' 'frozen' 'lock')
    PositionToolModel ('selectionType' 'alignmentX' 'topFraction'
        'bottomOffset' 'rightOffset' 'bottomFraction' 'alignmentY'
        'leftFraction' 'topOffset' 'leftOffset' 'rightFraction'
        'topSlide' 'leftSlide' 'bottomSlide' 'rightSlide'
        'alignmentXSlide' 'alignmentYSlide' 'tracking')
LensDFBasicsSliceModel ()
LensDFConnectionSliceModel ()
UIFinderVW2 ('classNameList' 'selectorList' 'iconList' 'filter' 'menuBar'
    'addWhat' 'partSortType' 'updateTrigger' 'lastCategory'
    'menuBarMenuWithSelection' 'menuBarMenuWithNoCanvasSelected'
    'menuBarMenuWithNoSelection' 'classListMenuWithSelection'
    'classListMenuForNoSelection')
UIPainter ('targetClass' 'targetSelector' 'acceptedState' 'windowSpec'
    'minWindowExtent' 'prefWindowExtent' 'maxWindowExtent'
    'currentLook' 'definer')
UIPainterWatcher ('selectionHolder')
MenuEditor ('targetClass' 'targetSelector' 'menu' 'menuBar'
    'menuBarView' 'menuList' 'currentItem' 'modified' 'properties')
UICanvasTool ('menuBar' 'controller' 'arrangeMode' 'constValue'
    'statusBarText')
UIMaskEditor ('magnifiedBitView' 'directBitView' 'acceptedState'
    'targetClass' 'targetSelector' 'modified' 'menuBar' 'storeMask'
    'useCachedImage' 'doTheCurrentColor')
UIMenuEditor ('targetClass' 'targetSelector' 'menu' 'menuBar'
    'menuString' 'modified')
UIPropertiesTool ('selection' 'selectionKind' 'controller'
    'currentSpecCopy' 'currentSpecBindings' 'specChannel' 'subBuilder'
    'lock' 'slice' 'lastSlice' 'slicesMenu' 'sliceInfo' 'client'

```

```

        'statusBarText' 'list' 'sliceChanging' 'sliceTabChanging')
UIPalette ('activeSpecs' 'toolName')
UISettings ('list' 'disturbed' 'subBuilder' 'memo')
VisualLauncher ('menuBar' 'oldHeight' 'textCollector')
ChangeSet ('classChanges' 'methodChanges' 'classRemoves'
           'reorganizeSystem' 'specialDoIts')
Explainer ('class' 'selector' 'instance' 'context' 'methodText')
LDMAbstractBody ('foa' 'locked' 'missedUpdate' 'viewModel')
  LDMRelationsBody ('composite' 'generator' 'graph' 'canOpen' 'visualBuilder')
LDMBrowserModel ('body' 'focusHolder' 'perspectiveHolder' 'attributes'
                'selectionService' 'application' 'visualBuilderClass')
LDMSelectionService ('viewsValue' 'performersValue')
LensApplicationSpec ('name' 'kind' 'dataModel' 'rowType' 'fields'
                    'editPolicy' 'lockPolicy' 'template' 'superclass' 'category'
                    'appType' 'selector')
LensSession ('serialNumberGeneratorBlock' 'dataModel' 'connection'
            'lensPolicy' 'containers' 'nonBaseContainers' 'isActive'
            'cascadedAdds' 'removedObjects' 'pendingOperations'
            'pendingOperationsIndex' 'keyChanges' 'delayingOperations'
            'savingLastCommittedVersions' 'lensTransactionPolicy'
            'pendingLocks' 'postponedOperations')
ScrollValueHolder ('offset' 'grid' 'extraSpace' 'permitDisplayOutput')
SelectionInList ('listHolder' 'selectionIndexHolder')
  MultiSelectionInList ()
SelectionInTable ('tableHolder' 'selectionIndexHolder')
SyntaxError ('class' 'badText' 'processHandle')
TableInterface ('columnWidths' 'columnFormats' 'columnLabelsHeight'
               'columnLabels' 'columnLabelsFormats' 'rowHeights' 'rowLabelsWidth'
               'rowLabels' 'rowLabelsFormats' 'selectionInTable'
               'columnLabelsAndSelection' 'rowLabelsAndSelection' 'tableView')
UIDataReference ('referred' 'name')
  LensGraphReference ('children' 'application' 'cascade' 'position')
  LensContainerReference ()
  LensVariableReference ()
ValueModel ()
  ComputedValue ('cachedValue' 'eagerEvaluation')
  BlockValue ('block' 'arguments' 'numArgs')
  PluggableAdaptor ('model' 'getBlock' 'putBlock' 'updateBlock')
  TypeConverter ()
  ProtocolAdaptor ('subject' 'subjectSendsUpdates' 'subjectChannel'
                  'accessPath')
  AspectAdaptor ('getSelector' 'putSelector' 'aspect')
  IndexedAdaptor ('index')
  SlotAdaptor ()
  RangeAdaptor ('subject' 'rangeStart' 'rangeStop' 'grid')
ValueHolder ('value')
  BufferedValueHolder ('subject' 'triggerChannel')
  HelpProxy ('displayString' 'fileIndex' 'position')
  LensRowHolder ('clientMap')
  Project ('projectWindows' 'projectChangeSet' 'projectTranscript'
          'projectHolder')
  TextCollector ('entryStream')

```

2.8 DisplaySurface

La hiérarchie des surfaces et des fenêtres fait le lien entre Smalltalk et l'interface hôte (Windows, W, Mac).

```

Object ()
  GraphicsMedium ()
    DisplaySurface ('handle' 'width' 'height' 'background')
    UnmappableSurface ()
    Mask ()
    Pixmap ()
    Window ('inputOrigin' 'creationOrigin' 'sensor' 'iconic' 'windowType'
             'effectiveWindowType' 'statusLineHeight' 'allowsStatusLine')
    ScheduledWindow ('label' 'icon' 'minimumSize' 'maximumSize'
                     'component' 'lookPreferences' 'paintPreferences'
                     'edgeDecorationPolicy' 'widgetPolicy' 'controller' 'model'
                     'damageRepairPolicy' 'masterWindow')
    ApplicationWindow ('keyboardProcessor' 'application'
                       'sendWindowEvents' 'receiveWindowEvents' 'windowEventBlock'
                       'damageRepairIsLazy' 'activationNotification' 'dropTarget')
    TransientWindow ('component' 'paintPreferences')

```

Ces différentes hiérarchies illustrent le rangement du code dans Smalltalk. Les hiérarchies sont fonctions de la version de l'environnement utilisé et elles évoluent avec la programmation utilisateur.

Chapitre 4

L'environnement de programmation de Smalltalk-80

Dans cette section, nous étudions l'environnement de programmation **VisualWorks** pour Smalltalk. **VisualWorks** est un environnement de programmation visuelle de ParcPlace Systems, Inc. Dans un rapport précédent [And96], nous avons mis l'accent sur l'utilisation de Smalltalk-80 avec **VisualWorks** sous Unix et X-Window. Nous avons présenté différentes versions de **VisualWorks** (d'**ObjectWorks** 4.0 à **VisualWorks** 1.0). Dans ce cours, nous utilisons **Window98** comme système hôte et **VisualWorks** 2.5 de 1995. Cette version est issue d'une fusion avec **Smalltalk-V** le langage et l'environnement de **Digitalk**. D'autres versions de Smalltalk et d'autres environnements utilisent des outils similaires. Le plus connu est celui d'IBM, appelé **VisualAge**, qui existe aussi pour Java.

Nous examinons les points suivants : lancement/arrêt de Smalltalk, fenêtres et outils de développement. Nous commençons, dans la section 1, par les consignes de mise en route, notamment des contraintes matérielles et logicielles d'utilisation d'applications sous Smalltalk-80 dans le système hôte. Nous poursuivons par la description des différents outils de développement de l'environnement et celle des fenêtres Smalltalk.

1 Mise en route

L'installation de l'application **VisualWorks** sous **Windows** entraîne la création d'un répertoire **VISUAL** contenant les fichiers nécessaires à l'environnement de développement (notons **RepVisual** le répertoire de **VisualWorks** (par exemple **RepVisual** = **C:\VISUAL**)).

1.1 L'image Smalltalk

L'élément clé de l'environnement de Smalltalk est l'image. L'image est un fichier qui contient tous les objets de Smalltalk. Sachant qu'une classe est un objet, on comprend vite que l'image contient "tout" Smalltalk. Au départ, l'image contient les classes et les objets de l'environnement, les actions du programmeur seront d'enrichir cet environnement (ajout de classes et d'objets). Pour conserver une certaine intégrité, il est bon de séparer l'image de base de l'image représentant un travail en cours. En effet, lorsqu'on met au point des programmes, il arrive que l'image soit polluée par des objets rémanents (c'est notamment le cas lorsqu'on développe des applications avec des processus) ou que les références d'objets soient incohérentes et rendent inexploitable l'image. On doit alors refaire une image propre. En particulier, on développera une image différente pour chaque application importante développée.

La première chose à faire est donc de copier l'image initiale de Smalltalk. Par défaut, cette image s'appelle **visual.im**. L'image est complétée par deux fichiers importants pour former l'environnement de développement. L'environnement est donc composée de trois fichiers :

- `visual.sou` : contient les fichiers sources des classes de base de l'environnement (toutes les classes du système Smalltalk-80). Il n'est pas nécessaire de dupliquer ce fichier car il n'évolue pas au cours du développement. Au contraire, sa copie risque d'occuper beaucoup de places sur le système de fichier. Ce constat est encore plus vrai dans un système multi-utilisateur. Ce fichier doit donc exister en un seul exemplaire : chaque image y fait référence.
- `visual.im` : contient tous les objets du système : les objets de base et les objets utilisateur.
- `visual.cha` : contient les modifications du système depuis la version de base (*changes*). Ce fichier qu'on appelle aussi le **journal** est créé automatiquement lorsqu'on sauve l'image. C'est le fichier compagnon de `visual.im`.

1.2 Création d'une image

Soit `RepST` le répertoire dans lequel nous allons travailler, *e.g.* `RepST = C:\SMALLTALK`. Pour créer une image Smalltalk-80, nous recopions le fichier de base `visual.im` du répertoire `RepVisual\IMAGE` dans le répertoire `RepST` sous le nom `ex.im`. La création du fichier `ex.cha` et la liaison avec fichier `visual.sou` se font dans l'environnement de développement.

1.3 Lancement de VisualWorks

Pour lancer Smalltalk, on ouvre le fichier `ex.im` (à partir d'une fenêtre Explorer on *double clique* sur l'icône du fichier ou on active le menu `Fichier>Ouvrir`). Des fenêtres s'affichent alors.

1.4 Environnement initial, vue et fenêtres

L'environnement comprend des fenêtres (`Windows` dans notre cas) pour l'interaction avec l'utilisateur. Ces fenêtres sont découpées en vues. Au démarrage de l'application, l'environnement initial se présente comme suit.

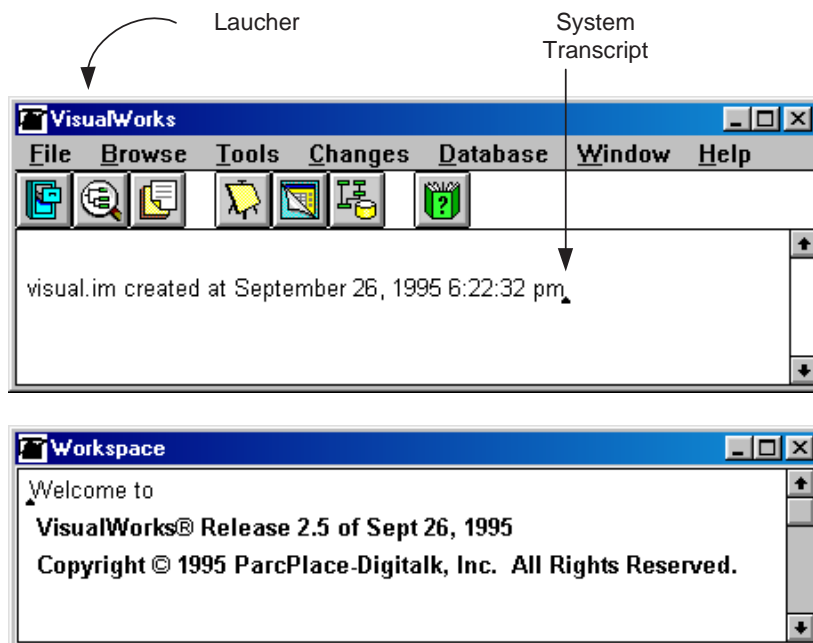


Figure 8 : *Environnement initial*

- La fenêtre **Launcher** rassemble les traitements relatifs à la session en cours : appel d’outils, sauvegarde et sortie de l’application avec en plus des outils de génération et d’utilisation d’interfaces. Dans l’ordre, de gauche à droite, les icônes représentent des raccourcis pour les fonctions suivantes :
 - **File list** : ouverture d’une fenêtre sur le système de fichier hôte,
 - **Class browser** : activation du navigateur de code des classes,
 - **Workspace** : création d’un espace de travail utilisateur (brouillon pour évaluer des expressions Smalltalk),
 - **New Canvas** : générateur d’interfaces homme-machine (à l’état de cannevas),
 - **Resource Finder** : ouvre un navigateur d’applications VisualWorks,
 - **Date modeler** : création de bases de données,
 - **Help** : activation de l’aide en ligne.

Toutes ces fonctions ouvrent de nouvelles fenêtres. La zone de texte s’appelle fenêtre **System Transcript**. C’est la console du système, elle affiche les messages du système.

- La fenêtre **Workspace** autrefois appelée **Installation Workspace** est une zone d’information pour la première installation de l’environnement.

Une seule fenêtre est active à un instant donné. La souris permet de passer d’une fenêtre à l’autre. Les fenêtres **Installation Workspace**, **System Workspace** et **System Transcript** sont des fenêtres contenant une vue de type texte. Le protocole d’utilisation des vues de textes est décrit dans la section 2.5.

1.5 La souris et les menus

En Smalltalk-80, la souris est un élément fondamental de l’utilisation de l’environnement. Son apparence change en fonction du contexte. Cette apparence est redéfinissable par le programmeur (voir la classe **Cursor**). Elle a trois boutons : gauche, central, droit (antérieurement les auteurs leur donnaient des couleurs).

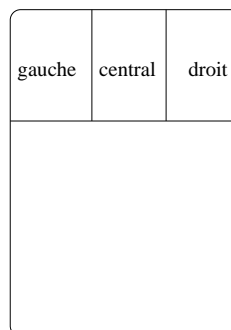


Figure 9 : *La souris Smalltalk-80*

La pression sur ces boutons fait apparaître des **menus** contextuels (**pop up menus**). La convention d’utilisation est la suivante :

- Le bouton *gauche* est utilisé pour pointer ou sélectionner une zone.
- Le bouton *droit* est utilisé pour les opérations de la fenêtre active. En cliquant sur ce bouton, le menu de la fenêtre apparaît avec les options suivantes :
 - **relabel as...** : donne un nouveau nom au bandeau de la fenêtre.
 - **refresh** : affiche à nouveau le contenu de la fenêtre.
 - **move** : déplace la fenêtre.
 - **resize** : change les dimensions de la fenêtre.
 - **front** : place la fenêtre au premier plan.
 - **back** : place la fenêtre en arrière plan.
 - **collapse** : donne à la fenêtre sa forme en icône.

- `close` : ferme définitivement la fenêtre.
- Le bouton *central* est associé à un menu contextuel. Par exemple deux contextes sont abordés dans ce document : la manipulation de textes est présentée dans la section 2.5) et la manipulation de listes est présentée dans la section 2.7).

Notez que ces menus sont redéfinissables par l'utilisateur dans ses propres applications. Lorsque la souris utilisée ne comporte que deux boutons, le bouton central est le bouton droit de la souris et le bouton droit s'obtient par la combinaison touche `Ctrl` et bouton droit de la souris.

1.6 Environnement initial

La première chose à faire est de positionner correctement les variables d'environnement dans l'image et sauvegarder cet environnement. Pour cela, nous effectuons quelques modifications de l'environnement.

1. Activer le menu `File>Settings` de la fenêtre `VisualWorks`.
2. Commençons par définir les fichiers de l'environnement. Dans la fenêtre `Settings` (appelée `Installation Workspace` avant), sélectionner l'onglet `Sources`.
 - Dans la zone `Sources`, indiquer le répertoire du code source soit `RepVisual\IMAGE\visual.sou`.
 - Dans la zone `Changes`, indiquer le répertoire du journal (le même que celui de l'image) `RepST\ex.cha`.
 Valider par le bouton `Accept`.
3. Poursuivons par le positionnement du répertoire de l'aide en ligne. Sélectionner l'onglet `Help`. Dans la zone `Documentation Directory`, indiquer le répertoire du `RepVisual\ONLINE`. Valider par le bouton `Accept`.
4. Enfin, on modifie le fuseau horaire. Sélectionner l'onglet `Time Zones`. Dans la fenêtre de texte, sélectionner avec le bouton gauche la zone correspondant au fuseau horaire choisi (le texte entre les quotes '). Par exemple, nous sélectionnons le méridien de Greenwich :

```
TimeZone setDefaultTimeZone:
(TimeZone timeDifference: 0
 DST: 1 at: 1
 from: 89 "on March 31"
 to: 273 "until September 30"
 startDay: #Sunday).
```

Une fois la zone noircie, nous l'évaluons par le menu `do it` du bouton central.

5. Les autres onglets contiennent différents paramètres de l'environnement. Leur paramétrage par défaut suffit pour débiter avec Smalltalk.
6. Quitter la fenêtre par le menu `close` du bouton droit. Sous `windows`, on peut aussi utiliser la fonction `Fermeture` (ou le raccourcis `Alt+F4`) accessible par le menu déroulant de la fenêtre (en haut à gauche) ou enfin l'icône (X) en haut à droite de la fenêtre.
7. Valider les changements dans l'image en utilisant le menu `File>SaveAs` de la fenêtre `VisualWorks`, une fenêtre d'enregistrement est ouverte, le nom de l'image doit être `ex`. Valider par le bouton `ok` ou par la touche `Entrée`. On peut aussi utiliser le menu `File>Exit VisualWorks`. Choisir le bouton `Save then Exit`, on retrouve la fenêtre d'enregistrement. Relancer l'application pour vous assurer que la modification est correcte. Vous retrouvez l'environnement tel que vous l'avez sauvé.


1.7 Sortir de Smalltalk

Utiliser le menu `File>Exit VisualWorks`. de la fenêtre `VisualWorks` (Laucher dans la figure 8). Sauver ou non l'image selon votre besoin.

2 Outils principaux et fenêtres de l'environnement

Nous abordons dans cette section quelques outils importants pour l'utilisation de l'environnement `VisualWorks`.

2.1 Lancer une application

Utiliser le bouton `Resource Finder`  (ou le menu `Browse>Resources`) de la fenêtre `VisualWorks` pour lancer une application. Un navigateur d'applications `VisualWorks` est lancé. Sélectionner la classe de l'application et lancer l'application par le `Start`. Par exemple, sélectionner la classe `Browser` et lancer le navigateur de classes.

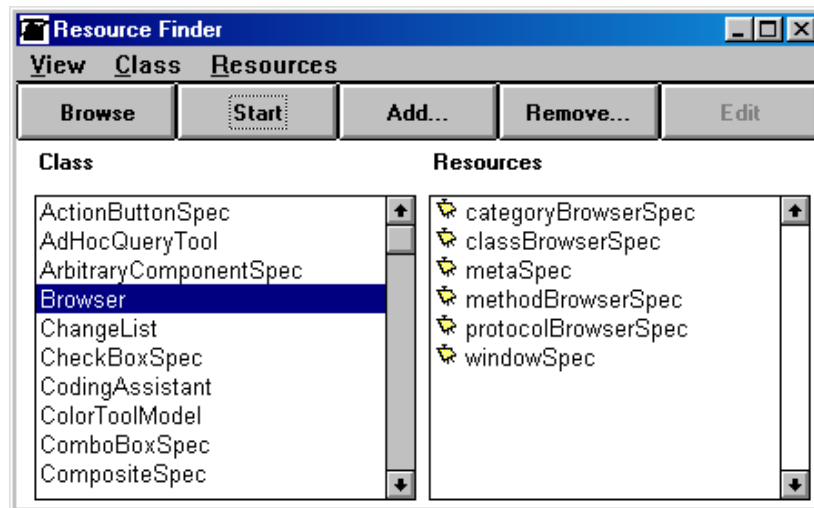


Figure 10 : Fenêtre de navigation de ressources

Lorsqu'on connaît la classe ou si on veut simplement évaluer une expression, on tape le texte de l'expression à évaluer dans une fenêtre `Workspace`. Par exemple, on tape l'expression `Browser open`, puis on sélectionne (on noircit) l'expression et on l'évalue par le menu `do it` du bouton central. Signalons, que la méthode de classe `open` correspond à l'ouverture (sans paramètres) d'une fenêtre. S'il n'existe pas de fenêtre `Workspace`, on peut en créer une nouvelle par le bouton `Workspace` (ou le menu `Tools>Workspace`) de la fenêtre `VisualWorks`.

2.2 Sortir d'une application

Utiliser le menu `close` de l'application, ou l'option `close` du menu correspondant au bouton droit de la souris, dans la fenêtre de l'application.

2.3 Lanceur

La fenêtre `VisualWorks` (`Launcher`) est le point d'entrée de la session de travail et de la manipulation de l'environnement de développement du système `Smalltalk` (depuis la version 4.0 d'`ObjectWorks` en fait). Elle comprend sept menus accessibles par le bouton gauche et des icônes (boutons) des fonctions principales des menus, que nous avons déjà détaillés dans la section 1.4 et la figure 8. Examinons les menus.

1. `File` : comprend les paramètres de l'environnement (sauvegarde, ramasse-miette -*garbage collect*, paramètres -*settings*, et sortie). Rappelons que le ramasse miette supprime de l'image tous les objets qui ne sont plus référencés par d'autres objets. La gestion de la mémoire est implicite comme en `Lisp`.

2. **Browse** : permet d'activer plusieurs navigateurs (voir section 2.8). Le flâneur de classe permet de consulter le code des classes de l'image. Il en existe deux versions : toutes les classes, une seule classe. Le flâneur de ressources permet de naviguer dans les applications et les outils d'interface. Le flâneur de références permet de naviguer dans les méthodes qui invoquent la méthode dont on précise le sélecteur. Le flâneur d'implanteurs permet de naviguer dans les méthodes dont on précise le sélecteur.
3. **Tools** : contient les appels aux outils utilisés dans le développement : explorateur du système de fichiers, éditeur de texte (d'un fichier), espace de travail temporaire (*workspace*), éditeurs d'interfaces, de menus ou d'images, console (*transcript*), etc. La console renseigne l'utilisateur sur ce qui se passe dans le système en affichant les messages du système. Elle fait partie intégrante de la fenêtre *Workspace*.
4. **Changes** : gestion du journal associé à l'image et gestion de projets.
5. **Database** : outils de manipulation d'une base de données : définition, accès, écrans de saisie, application, etc.
6. **Windows** : gestion des fenêtres.
7. **Help** : aide en ligne.

2.4 Dialogues

A chaque fois qu'une interaction a lieu avec l'utilisateur (ex : valider un choix, demander un nom de fichier, un nom de classe) une fenêtre de dialogue est ouverte. S'il s'agit d'un dialogue texte, alors ne rien mettre équivaut à annuler.

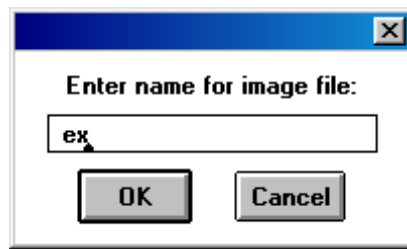


Figure 11 : Une fenêtre de dialogue texte

2.5 Textes

Dans une fenêtre contenant une vue texte, nous retrouvons les fonctions d'un éditeur de texte rudimentaire. Le bouton droit permet d'accéder au menu de manipulation de la fenêtre. Le bouton gauche sert à sélectionner une zone ou placer le curseur d'écriture dans une position donnée de la vue texte. Enfin le menu central (contextuel) est le suivant

- **find...** : recherche une chaîne de caractères dans le texte.
- **replace...** : recherche et remplace une chaîne de caractères dans le texte.
- **undo** : annule la dernière action.
- **copy** : copie la zone sélectionnée dans le presse-papier.
- **cut** : supprime la zone sélectionnée et la place dans le presse-papier.
- **paste** : recopie le contenu du presse-papier à l'endroit pointé par le curseur.
- **do it** : exécute le code contenu dans la zone sélectionnée. S'il ne s'agit pas d'une suite d'envois de message, rien ne se passe.
- **print it** : exécute le code contenu dans la zone sélectionnée et l'affiche à l'emplacement du curseur de la fenêtre texte. S'il ne s'agit pas d'une suite d'envois de message, affiche `nil`.
- **inspect** : exécute le code contenu dans la zone sélectionnée et lance l'outil inspecteur sur l'objet résultat de l'exécution (voir section 2.9).

- **accept** : sauvegarde le contenu de la fenêtre. Cette sauvegarde se fait dans l'objet auquel est associé le texte (voir section 5).
- **cancel** : restaure le contenu de la fenêtre depuis l'ancienne sauvegarde.
- **hardcopy** : fait une copie de la fenêtre sur imprimante.

D'autres fonctions sont ajoutées dans le contexte de "l'explorateur" de fichiers.

- Si un fichier est sélectionné, les fonctions suivantes sont ajoutées :
 - **file it in** : inclue le fichier dans l'environnement s'il s'agit d'un fichier source Small-talk (voir section 3).
 - **save** : enregistre les modifications dans le fichier.
 - **save as...** : enregistre les modifications dans le fichier dont on renomme le répertoire.
 - **spawn** : ouvre un éditeur de texte sur le fichier en cours.
- Si un répertoire est sélectionné. La fonction **cancel** n'apparaît pas mais la fonction suivante est ajoutée :
 - **spawn** : ouvre une nouvelle fenêtre liste sur le répertoire sélectionné.

2.6 Espace de travail

Un espace de travail (**workspace**) est une zone de texte contenant des expressions à évaluer. C'est une sorte de post-it. On peut en avoir autant qu'on veut. Utiliser l'opération **accept** du bouton central pour mémoriser le contenu de cette fenêtre. Les espaces de travail sont conservés tels quels lors de la sauvegarde de l'image.

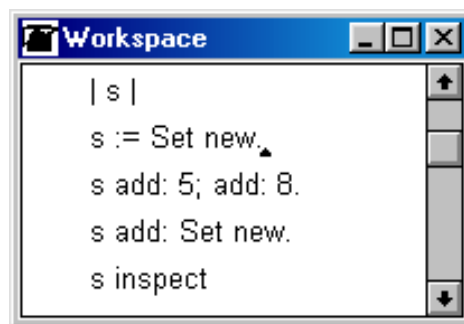


Figure 12 : *Un espace de travail*

2.7 Listes

Beaucoup d'opérations de choix se font par des listes. Par exemple, en cliquant sur le bouton **File list** de la fenêtre **Workspace** (ou le menu **Tools>File list**), on affiche le contenu d'un répertoire du système de fichier hôte.

Une fenêtre de dialogue de type liste comporte généralement deux vues : une **vue liste** et une **vue texte**. Dans notre exemple, une troisième vue est affichée, elle contient le motif (chemin d'accès) au répertoire courant. La vue texte est décrite dans la section 2.5. Dans notre exemple, le menu texte est légèrement enrichi.

La vue sur liste est régie de la façon suivante. Le bouton droit concerne (comme d'habitude) le menu de manipulation de la fenêtre. Le bouton gauche sert à sélectionner (ou désélectionner) un élément de la liste dans la vue liste. Si un élément est sélectionné alors il s'affiche alors en surlignement. Le bouton central sert à manipuler les éléments de la liste. Le menu associée varie en fonction de la sélection courante. Si un élément est sélectionné, alors le menu offre des opérations sur cet élément sinon le menu offre des opérations applicables à l'ensemble des éléments de la liste.

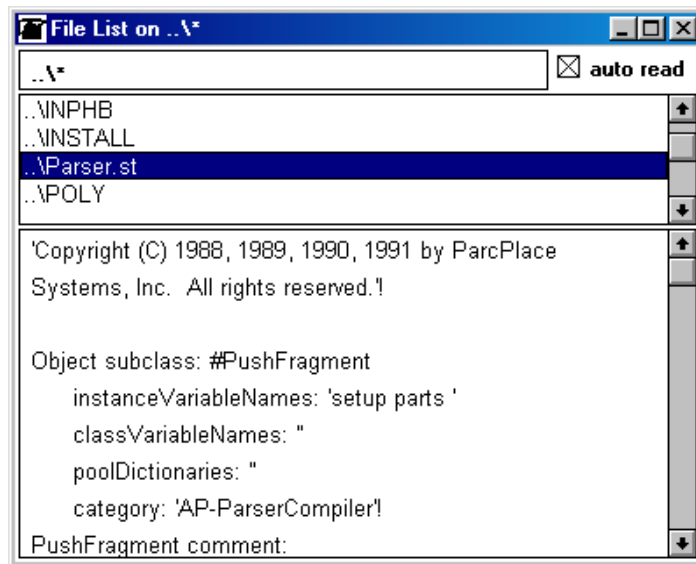


Figure 13 : Une fenêtre de dialogue de type liste

Dans notre exemple, si aucun fichier n'est sélectionné dans la liste, le menu central comprend une seule opération : **créer un répertoire**. Si un fichier est sélectionné, son contenu ou ses informations s'affichent au choix dans la vue texte. S'il s'agit d'un fichier ordinaire les opérations accessibles sont :

- **get info** : lit les informations du fichier
- **file in** : inclue le fichier dans l'environnement s'il s'agit d'un fichier source Smalltalk (voir section 3).
- **copy name** : copie le nom du fichier dans le presse-papier.
- **rename as...** : renomme le fichier.
- **copy file to...** : duplique le fichier.
- **remove...** : détruit le fichier.
- **spawn** : ouvre une fenêtre texte sur le contenu du fichier.

S'il s'agit d'un répertoire les opérations accessibles sont :

- **new pattern** : le répertoire sélectionné devient le nouveau répertoire courant.
- **add directory** : crée un répertoire dans le répertoire sélectionné.
- **add file** : crée un fichier ordinaire dans le répertoire sélectionné.
- **copy name** : copie le nom du répertoire dans le presse-papier.
- **rename as...** : renomme le répertoire.
- **remove...** : détruit le fichier.
- **spawn** : ouvre une nouvelle fenêtre liste sur ce répertoire.

2.8 Flâneurs, navigateurs et organisation du code

Dans cette section, nous passons en revue les outils de manipulation de classes et de hiérarchies de classes. Nous empruntons à Pierre Cointe la traduction gracieuse de **flâneur** pour le terme angliciste **browser**. Le terme **navigateur** semble être maintenant le plus approprié, car il est passé dans le langage courant des informaticiens depuis l'avènement d'internet. Un flâneur sert à fouiller dans les classes existantes pour trouver du code proche de ce que le programmeur veut faire. Code qui sera ensuite adapté par héritage.

Le **code** est rangé selon la hiérarchie suivante. Une **catégorie** contient plusieurs classes. Une **classe** contient plusieurs protocoles. Chaque **protocole** rassemble plusieurs méthodes. Une **méthode** est définie par un profil et un corps. Une classe dont une méthode fait appel à une autre méthode **expéditeur** (*sender*) de cette autre méthode. Une classe qui définit une méthode est dite **implanteur** (*implementor*) de la méthode.

Pour ouvrir un flâneur, nous utilisons l'option **Browsers** > de la fenêtre. Ces flâneurs sont généralement des fenêtres avec des vues listes comme définies dans la section 2.7.

- **system** : ouvre une fenêtre flâneur sur l'ensemble des classes du système. Ces classes apparaissent dans des vues listes selon la hiérarchie décrite ci-dessus. La vue principale est de type texte et son contenu évolue en fonction des éléments sélectionnés dans les listes.
- **class...** : ouvre une fenêtre flâneur sur la classe spécifiée par l'utilisateur (sous-ensemble du flâneur précédent).
- **sender of...** : ouvre une fenêtre liste dont les éléments sont les méthodes qui utilisent la méthode spécifiée par l'utilisateur (les expéditeurs). Lorsque le contenu de la méthode est affiché : la méthode utilisée apparaît en surligné.
- **implementors of...** : ouvre une fenêtre liste dont les éléments sont les méthodes qui implante la méthode spécifiée par l'utilisateur.
- **class examples** : ouvre une fenêtre flâneur sur les classes ayant dans leur protocole de classe un protocole d'exemple. C'est un bon moyen d'appréhender la connaissance des classes.

La figure 14 représente l'exécution d'un flâneur système. Il est décomposé en quatre vues de listes, deux boutons "radio" et une vue de texte.

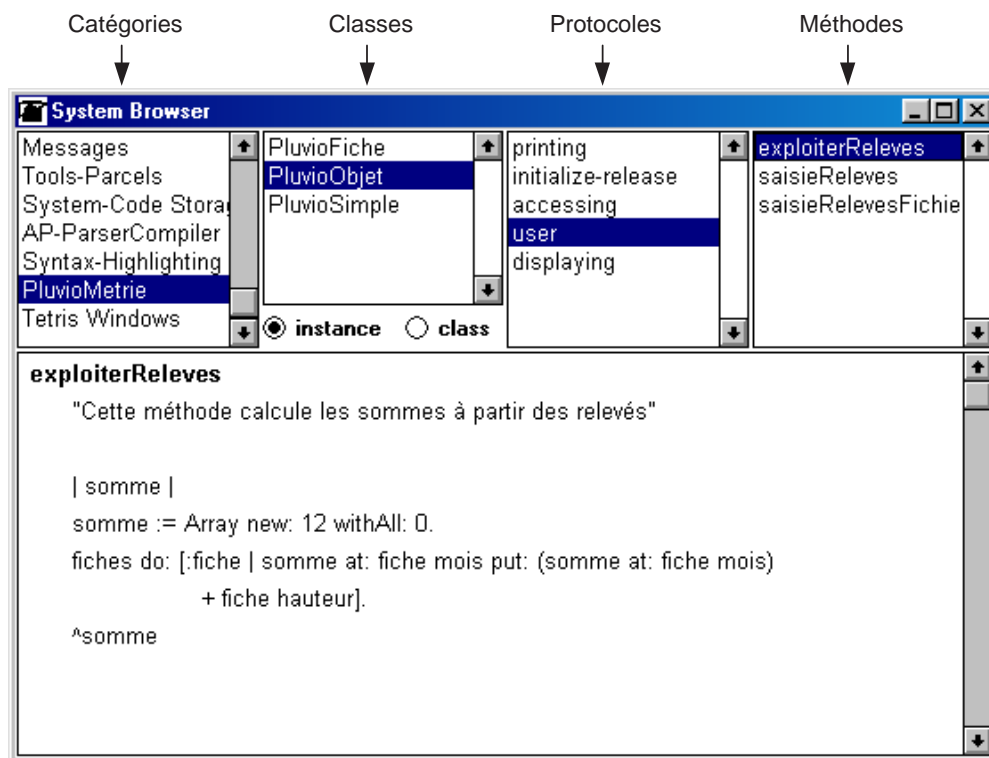


Figure 14 : Une fenêtre flâneur système

La navigation s'effectue de gauche à droite dans les vues de listes selon les relations de causalité suivantes.

1. Aucune catégorie n'est sélectionnée : toutes les fenêtres sont vides sauf celle des catégories.
2. Une catégorie est sélectionnée. La fenêtre des classes liste toutes les classes de la catégorie.
 - (a) Aucune classe n'est sélectionnée : les fenêtres de protocoles et de méthodes sont vides. La zone de texte contient un prototype de création d'une nouvelle classe dans cette catégorie.

- (b) Une classe est sélectionnée : la suite dépend de l'état du bouton radio.
 - i. Si `instance` est sélectionné, ce qui est dit plus loin concerne la classe (définition structurelle de la classe, méthodes d'instance, hiérarchie de la classe).
 - ii. Si `class` est sélectionné, ce qui est dit plus loin concerne la métaclasse (définition structurelle de la métaclasse, méthodes de classe, hiérarchie de la métaclasse).

La fenêtre des protocoles liste tous les protocoles de la classe (resp. de la métaclasse).

- i. Aucun protocole n'est sélectionné : la fenêtre de de méthodes est vide. La zone de texte contient la définition structurelle de la classe (resp. de la métaclasse).
- ii. Un protocole est sélectionné : la fenêtre des méthodes liste toutes les méthodes du protocole.
 - A. Aucune méthode n'est sélectionnée : la zone de texte contient un prototype de création d'une nouvelle méthode dans ce protocole.
 - B. Une méthode est sélectionnée : la zone de texte contient la définition de la méthode.

A chaque niveau de liste, par le menu du bouton central, lorsqu'un élément est sélectionné, on peut

- générer la description textuelle du code dans un fichier (catégorie, classe, protocole, méthode) par l'option `file out as...`,
- ouvrir un flâneur spécifique par l'option `spawn`,
- générer une copie papier par l'option `hardcopie`,
- renommer l'élément par l'option `rename as...`,
- supprimer l'élément par l'option `remove`.

Dans les listes de catégories ou de protocoles, on peut

- ajouter de nouvelles catégories ou de nouveaux protocoles par l'option `add`,
- rechercher une classe ou une méthode l'option `find`,
- afficher l'ensemble des catégories ou des protocoles par l'option `edit all` (affichage sous forme de tableaux de noms).

Dans les listes de catégories ou de protocoles, on peut

- ajouter de nouvelles classes ou méthodes par modification d'une classe ou méthode ou à partir d'un prototype de classe ou de méthode,
- déplacer une classe dans une autre catégorie ou une méthode dans un autre protocole par l'option `move to...`

Enfin, chaque fenêtre de liste contient des fonctions propres accessibles par le bouton central.

- Catégorie. L'option `update` est essentielle à la cohérence du code dans l'image. Si on a ajouté ou modifié des catégories sans passer par le navigateur (par exemple par un navigateur de fichier et la fonction `file in`) alors il faut mettre à jour le contenu du navigateur par la fonction `update`.
- Classe ou métaclasse.
 - L'option `hierarchy` permet d'afficher dans la zone de texte les superclasse de la classe sélectionnée.
 - L'option `comment` permet d'afficher dans la zone de texte le commentaire de la classe sélectionnée.
 - L'option `definition` permet d'afficher dans la zone de texte la définition structurelle de la classe sélectionnés.
 - L'option `spawn hierarchy` permet d'ouvrir un navigateur sur les superclasses et les sous-classes de la classe sélectionnée.

- Les options **refs** permettent d'afficher dans une nouvelle fenêtre l'utilisation de la classe, des variables d'instance ou des variables de classe.
- Méthode.
 - L'option **senders** permet d'ouvrir un navigateur sur les méthodes qui invoquent la méthode sélectionnée.
 - L'option **implementors** d'ouvrir un navigateur sur les méthodes qui implantent le sélecteur de la méthode sélectionnée.
 - L'option **message...** d'ouvrir un navigateur sur les méthodes qui implantent les sélecteurs invoqués dans le code de la méthode sélectionnée.

Ainsi que nous l'avons vu précédemment, la zone texte affiche selon son contexte un prototype de classe ou de méthode, la définition d'une classe, d'une méta-classe ou d'une méthode. Cette zone contient toutes les fonctions habituelles du menu central d'une fenêtre de texte. En particulier, l'option **accept** valide le texte par appel au compilateur (pseudo-code) et l'option **cancel** reprend le texte depuis la dernière validation. A ces fonctions s'ajoutent trois options

- **format** : active la fonction **pretty print** ou présentation normalisée du texte. Attention, cette fonction n'a d'intérêt que pour les méthodes.
- **spawn** : ouvre un navigateur sur la classe ou la méthode.
- **explain** : affiche des informations sur un élément sélectionné dans la zone de texte. C'est une fonction puissante de documentation automatique du code. Le résultat est une chaîne de caractère qui s'affiche dans le contexte d'évaluation.

2.9 Inspecteur

L'inspecteur est un outils d'examen du contenu d'un objet choisi. Cet objet est le résultat de l'évaluation d'une expression. L'inspection se fait soit dans l'expression par un envoi de message **expression inspect** ou par la sélection d'une expression dans une fenêtre et le menu central **inspect** de la souris.

Une fenêtre s'ouvre sur le contenu de l'objet : la valeur de ses variables d'instance. Par exemple, évaluons le code suivant dans l'espace de travail de la figure 12. Le résultat est l'ouverture d'une fenêtre inspecteur qui permet d'examiner récursivement le contenu des variables d'instance ou d'évaluer dans ce contexte une expression Smalltalk.

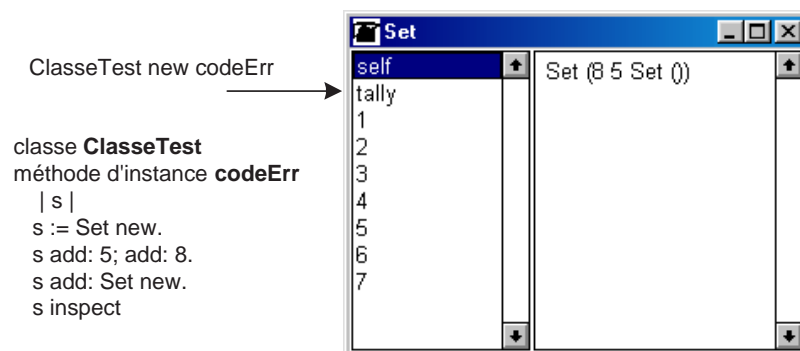


Figure 15 : Une fenêtre inspecteur

2.10 Débogueur

Le débogueur permet d'inspecter le code d'une méthode lors de l'exécution et de suivre pas à pas le déroulement de l'évaluation d'une expression. Une fenêtre débogueur est ouverte soit à l'initiative du programmeur par l'envoi de message **self halt**, soit lorsqu'une erreur est levée pour le **traitement d'exceptions**.

Par exemple, soit une méthode `codeErr` dans la classe `ClasseTest`, dont le code est donné en partie gauche de la figure 16. L'évaluation de l'envoi de message `ClasseTest new codeErr` génère une erreur car le message `addi :` est inconnu.

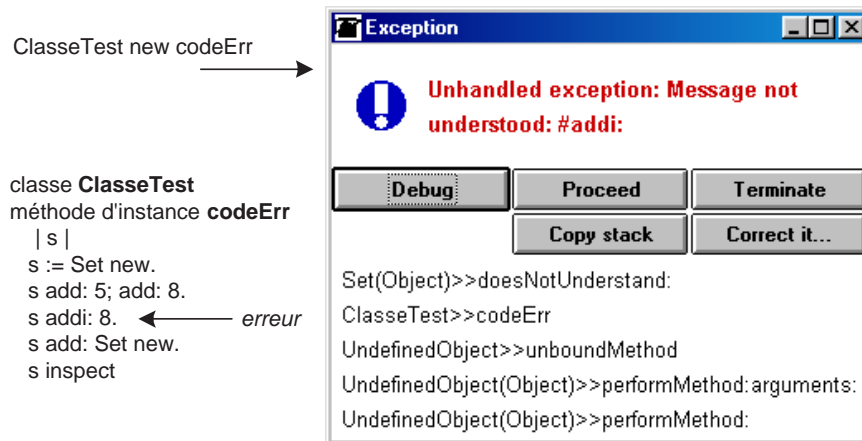


Figure 16 : Une fenêtre signalement d'erreur

L'option `debug` dans la fenêtre erreur permet d'inspecter le code défaillant comme le montre la figure 17.

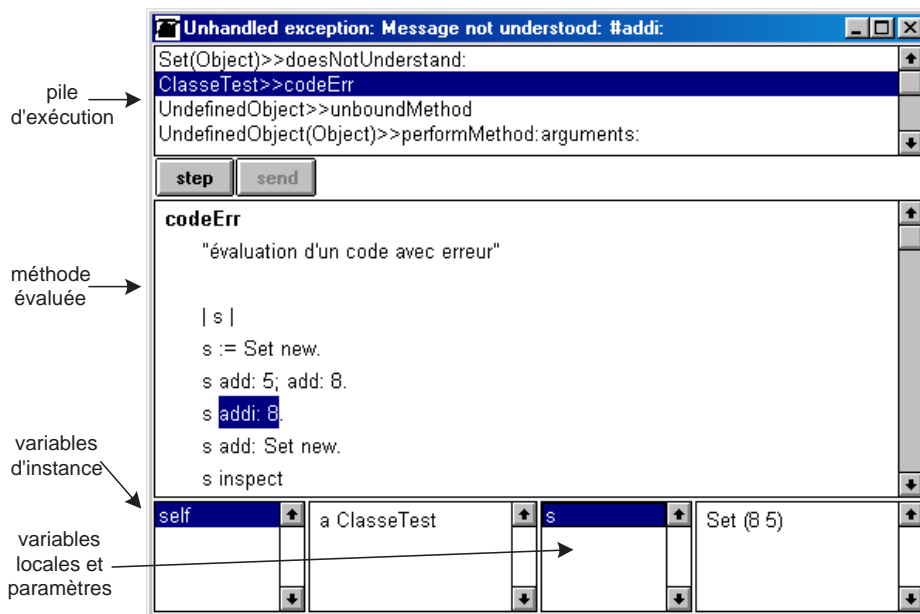


Figure 17 : Une fenêtre débogueur

Cette fenêtre débogueur est découpée en quatre zones : pile d'exécution, méthode en cours d'exécution, variables d'instances et variables temporaires (locales et paramètres). Les zones des variables d'instance ou temporaires sont des fenêtres d'inspection. La zone centrale est une zone de texte dans laquelle on peut modifier dynamiquement le code de la méthode (comme sous un browser de classe). On peut aussi y évaluer du code. Si on modifie le code alors on valide la modification par l'option `accept` du bouton central puis relance l'exécution par l'option `restart` du bouton central dans la zone de la pile d'exécution.

L'option `more stack` du bouton central dans la zone de la pile d'exécution permet d'avoir une pile d'exécution plus complète et remonter à l'envoi de message qui a causé l'erreur. Il est possible à ce niveau de modifier le code et de ré-exécuter dynamiquement la suite soit

pas à pas par la combinaison de l'option `step` (un pas de plus dans l'exécution = un envoi de message exécuté) et de l'option `send` (étude en profondeur de cet envoi de message) du bouton central ou soit normalement par l'option `proceed` du bouton central (le débogage est abandonné).

3 Représentation textuelle du code

Les programmes Smalltalk ont une description textuelle accessible par n'importe quel éditeur de texte. Ces fichiers sont suffixés par `.st`. Ils comprennent indifféremment des déclarations de méthodes, de protocoles et de classes. Les conventions d'écritures (générales ou personnelles) suivantes sont inspirées de la traduction des classes formelles [ACR94].

Les différentes déclarations Smalltalk sont séparées par le point d'exclamation (!). La première chose à faire est l'écriture d'une entête de la description textuelle. Nous l'obtenons en lisant un fichier quelconque suffixé par `.st`. Prenons par exemple le fichier généré pour la classe `PluvioFiche` de la section 1.1.

```
'From VisualWorks(R), Release 2.5 of September 26, 1995 on January 5, 1999
at 9:53:28 am'
```

Viennent ensuite les déclarations de structure des instances de la classe et des liens d'héritage.

Pour passer de l'héritage multiple d'une conception à objets à l'héritage simple de Smalltalk. Une politique simple est utilisée : un chemin d'héritage principal est choisi, puis les variables d'instance et les méthodes des chemins secondaires qui ne sont pas redéfinies dans la classe elle-même sont recopiées (règle de gestion des conflits des classes formelles).

Autre particularité, en smalltalk les variables d'instance héritées ne sont pas déclarées. Un petit contrôle permet de savoir lesquelles sont héritées, lesquelles sont définies, lesquelles sont redéfinies (la redéfinition correspond à un sous-typage du type de la variable d'instance). Ces informations figurent en commentaire de la classe. Les paramètres de type n'apparaissent pas explicitement dans la déclaration, du fait de l'absence de types statiques. Ils figurent aussi en commentaire.

```
Object subclass: #PluvioFiche
  instanceVariableNames: 'lieu mois hauteur '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PluvioMetrie'!
```

Noter que le premier nom correspond à la super-classe (unique), et que le message `variableSubclass: .. category: ..` est envoyé à la classe `Object`, supposée exister. Le second nom ne désigne pas la classe mais uniquement son nom (c'est un symbole Smalltalk). Il n'y a pas de variables de classes ou de pool dans cet exemple. La catégorie est un nom arbitraire. Les variables ne sont pas typées, pour ne pas perdre cette information, on la place en commentaire comme la contrainte ou la liste d'exportation.

PluvioFiche comment:

```
'Cette classe implante en objet le programme pluvio. Il s'agit d'une
écriture Smalltalk avec conception en classes d'un programme impératif.
```

```
Le programme est implanté par une classe PluvioObjet avec les méthodes de
saisie, addition, exploitation et affichage.
```

```
Des variantes sont possibles du point de vue des interfaces.
```

```
Les variables d'instance sont :
```

```
  lieu : String
```

```
mois : Integer [1..12]
hauteur : Integer
```

La principale difficulté est la saisie des entiers. Nous passons par le scanner Smalltalk qui rend un tableau d'arguments interprétés. Une autre possibilité consiste à utiliser une nouvelle interface avec des champs (InputField) dont les types sont précisés. »!

Les méthodes d'instance sont alors écrites. Dans la description textuelle, chaque déclaration de protocole ou de méthode se termine par un !. Nous avons pour cet exemple utilisé deux protocoles par défaut : **accessing** pour les méthodes d'accès aux instances en lecture et écriture, **printing** pour les méthodes de représentation textuelle de l'objet. Les méthodes privées sont mises dans le protocole **private**, il n'y en a pas ici.

```
!PluvioFiche methodsFor: 'accessing'!
```

```
hauteur
  ↑hauteur!
```

```
hauteur: aValue
  hauteur ←aValue!
```

```
lieu
  ↑lieu!
```

```
lieu: aValue
  lieu ←aValue!
```

```
mois
  ↑mois!
```

```
mois: aValue
  mois ←aValue! !
```

```
!PluvioFiche methodsFor: 'printing'!
```

```
printOn: aStream
```

```
"Append to the argument aStream a sequence of characters that
  identifies the receiver."
```

```
"Amplifies method in superclass to add name string."
```

```
"nextPutAll au lieu de print pour \ 'eviter les quotes"
```

```
aStream cr.
super printOn: aStream.
aStream cr; nextPut: $(.
aStream nextPutAll: lieu; tab; nextPutAll: mois printString; tab;
  nextPutAll: hauteur printString.
aStream nextPut: $)! !
```

Les méthodes de création sont mises dans le protocole de la métaclasse selon les mêmes normes que la traduction des méthodes d'instance. Dans notre exemple, il n'y a pas de variable d'instance de la métaclasse.

```
PluvioFiche class
  instanceVariableNames: ''!
```

```
!PluvioFiche class methodsFor: 'instance creation'!
```

```
newLieu: aString mois: anInteger hauteur: anInteger1
```

```
"crée une nouvelle fiche (sans contrôle)"
```

```
| fiche |
fiche ←self new.
fiche lieu: aString.
fiche mois: anInteger.
fiche hauteur: anInteger1.
↑fiche!
```

```
saisirFiche
```

```
"saisit une fiche au clavier"
```

```
| res val |
res ←Dialog
    request: 'Donnez votre fiche : lieu mois hauteur
            (annuler pour ne pas en saisir)'
    initialAnswer: 'z'
    onCancel: ['z']. "Le problème est la lecture des entiers"
val ←Scanner new scanTokens: res.
(val at: 1) asString == 'z'
ifTrue: ["tableau interprété des valeurs lues :
        l = val [1], m = val [2], h = val [3]"
    ↑self
        newLieu: 'z'
        mois: 0
        hauteur: 0]
ifFalse:
    [| m h |
    m ←val at: 2.
    [m < 1 | m > 12]
    whileTrue:
        [Dialog warn: 'Erreur mois (1..12)'.
        m ←(Scanner new scanTokens:
            (Dialog request: 'Donnez le mois' initialAnswer: '1'))
            at: 1].
    h ←val at: 3.
    [h < 0]
    whileTrue:
        [Dialog warn: 'Erreur hauteur négative '.
        h ←(Scanner new scanTokens: (
            Dialog request: 'Donnez la hauteur' initialAnswer: '1'))
            at: 1].
    ↑self
        newLieu: (val at: 1)
        mois: m
        hauteur: h]!
```

L'inclusion d'un fichier source Smalltalk dans l'image se fait en utilisant la fonction `file in` de la fenêtre `File List` (figure 13).

4 Programmation avec VisualWorks

Dans cette section, nous allons monter comment coder un programme en Smalltalk avec VisualWorks. L'exemple support est le jeu de Nim.

4.1 Le jeu de Nim

Le jeu de Nim se joue entre deux joueurs et avec un tas d'allumettes. Les joueurs enlèvent alternativement 1, 2 ou 3 allumettes. Le perdant est celui qui épuise le tas.

4.2 Conception abstraite à objets

Le diagramme de classes de la figure 18 représente une conception abstraite du programme dans la notation uml [Gro03, RJB99]. Une conception abstraite à objets plus formelle est présentée dans [AR98], qui explicite la spécification des opérations. Quatre classes composent l'application : `Personne`, `Personne`, `Joueur`, `Arbitre` et `JoueurIntelligent`.

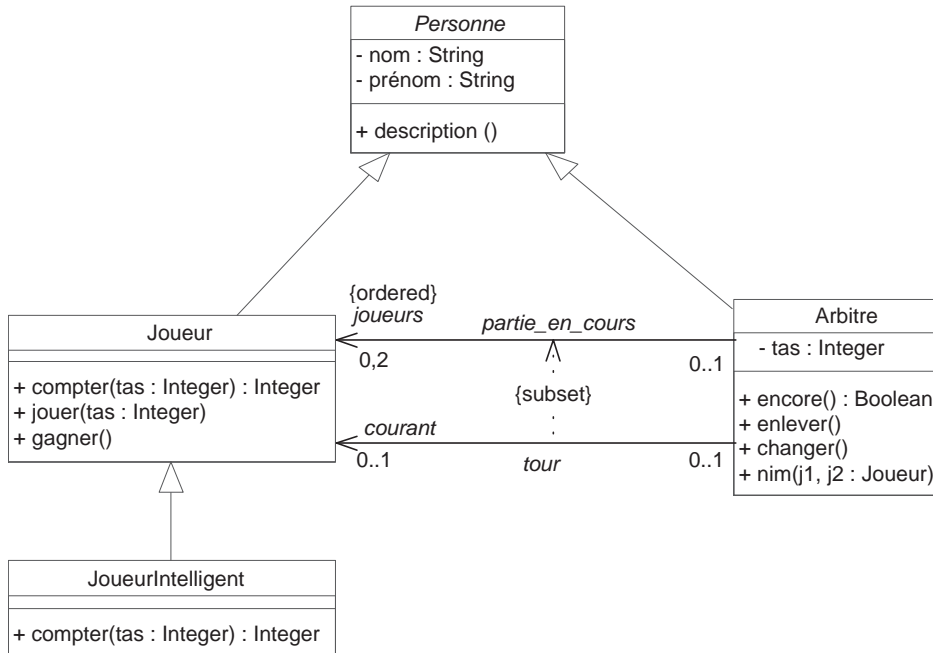


Figure 18 : *Environnement initial*

La classe abstraite `Personne` définit les participants au jeu. Les participants sont caractérisés par deux attributs : le nom et le prénom. Nous avons défini une opération `description()` qui rend une description textuelle de la personne. On distingue les deux joueurs de l'arbitre en utilisant la relation de spécialisation.

La classe `Joueur` définit les opérations (les responsabilités) suivantes :

- `compter` : le joueur détermine le nombre d'allumettes à ôter,
- `jouer` : le joueur joue un coup,
- `gagner` : le joueur se proclame gagnant.

La sous-classe `JoueurIntelligent` décrit des joueurs qui savent "mieux compter". Au lieu de tirer au hasard, le joueur choisi en fonction du nombre restant. L'opération `compter` est redéfinie.

L'arbitre a la maîtrise du jeu, c'est lui qui détient le tas d'allumettes et organise les actions des joueurs. Il définit les opérations (les responsabilités) suivantes :

- `encore` : indique si la partie n'est pas finie,
- `enlever` : modifie le tas d'allumettes après qu'un joueur ait joué un coup,
- `changer` : l'arbitre passe la main à un autre joueur,
- `nim` : l'arbitre démarre une nouvelle partie. Si une partie était en cours, elle est annulée.

La partie n'est possible que lorsque les joueurs sont mis en relation avec l'arbitre. Les deux associations `partie_en_cours` et `tour` symbolisent les liens entre joueurs et l'arbitre.

L'association `partie_en_cours` indique les deux joueurs de la partie en cours (ou aucun s'il n'y a pas de partie en cours). La contrainte `{ordered}` signifie qu'ils sont distingués selon un ordre donné. L'association `tour` indique le joueur qui doit jouer un coup. La contrainte `{subset}` met en évidence le fait que le joueur joue un coup dans la partie en cours. Autrement dit, le joueur qui a la main est un des participants du jeu. La navigation est unidirectionnelle pour les deux associations, cela signifie que les joueurs n'ont pas connaissance de l'arbitre ou encore que les communications se font dans le sens arbitre vers joueurs. Ce choix dans la navigation a une conséquence en conception détaillée (ou concrète). Il n'y a pas non plus de communication directe entre deux joueurs.

Noter qu'on respecte le principe d'encapsulation : les attributs sont privés et les opérations sont publiques. Ainsi seul l'arbitre peut modifier le tas d'allumettes (par l'opération `enlever`), les noms et prénoms ne peuvent être changés car il n'y a pas d'opération associées. Les opérations d'accès en lecture et de création d'instance n'ont pas été explicitement modélisées. Les types des attributs sont ceux du langage Object Constraint Language (OCL), qui fait partie de la notation UML.

4.3 Conception détaillée

Ce que nous appelons conception détaillée (ou concrète) est la mise en place de la conception dans l'environnement de développement cible. Il ne s'agit pas encore de coder le programme mais de faire l'adéquation entre les concepts de la conception abstraite et ceux de la conception détaillée. En principe, les concepts d'objets et de classes se retrouvent dans l'environnement cible.

Principes généraux

En supposant, un seul langage de programmation cible¹, voici les grandes lignes d'une telle transition :

- **Vocabulaire** : on adapte le vocabulaire à celui du langage cible. Par exemple, un attribut (resp. une opération) d'instance sera appelée variable (resp. méthode) d'instance en Smalltalk ou donnée (resp. fonction) membre en C++.
- **Typage** : on adapte les règles de typage et les conventions associées à celles du langage cible. Par exemple, en Smalltalk, le typage est dynamique alors qu'en C++ il est statique. Le polymorphisme des méthodes est implicite en Smalltalk alors que celui des fonctions C++ est explicite par le qualificatif `virtual`.
- **Association** : ce point est détaillé dans la section suivante.
- **Héritage multiple** :
 - Si le langage cible n'autorise pas l'héritage multiple (Smalltalk par exemple), il faut mettre en œuvre une politique de traduction adaptée. Par exemple, on choisit un chemin d'héritage principal et on duplique les caractéristiques issues des chemins secondaires.
 - Si le langage cible n'autorise pas l'héritage multiple, il faut adapter la stratégie de gestion des conflits à celle du langage cible (renommage et redéfinition en Eiffel, tri topologique en CLOS).
- **Contrôle des objets** : La plupart des langages à objets manipulent des objets séquentiels passifs. Lorsque la conception définit des objets actifs, il faut mettre en place un environnement d'exécution distribué simulé (processus en Smalltalk) ou intégré (*threads* Java). Ce point, qui inclue les variantes d'envois de messages et d'événements, constitue une étape essentielle de la conception pour les systèmes temps-réels.

1. Avec CORBA, des objets issus de différents langages peuvent coopérer.

- **Méta-objets** : certains langages autorisent des protocoles pour méta-objets (Smalltalk), pour les autres, il faut simuler en fonction des possibilités du langage (routines en Eiffel).
- **Assertions** : si le langage autorise les assertions (Eiffel par exemple), la traduction de contraintes OCL est simplifiée, sinon il faut programmer explicitement les contraintes.
- **Réutilisation** : une partie de la conception est réécrite en fonction des éléments qui existent dans l'environnement cible. C'est le cas par exemple des collections OCL.

Conception des associations

En programmation à objets, une association s'implante habituellement avec des pointeurs. On peut s'inspirer des règles de transformation du schéma E-A-P dans le modèle relationnel. Examinons les alternatives de modélisation.

1. L'association ne possède pas de propriétés. Les liens sont représentés par des attributs de navigation, un par classes de la relation. Par exemple, *arbitre.courant* donne le joueur qui a la main. Ces attributs prennent en général les noms des rôles. Il y a quelques cas particuliers :
 - (a) Navigation unidirectionnelle : un seul attribut de lien, dans la classe origine de la navigation. C'est le cas des deux associations ici.
 - (b) Cardinalité maximale de 1 dans un sens : on peut choisir une navigation unidirectionnelle.
2. L'association possède des propriétés et
 - (a) une cardinalité égale à 1. Les propriétés migrent dans la classe correspondant à la cardinalité. Après migration, on se retrouve dans le cas 1.
 - (b) aucune cardinalité maximale de 1. L'association donne lieu à une nouvelle classe. Elle est reliée aux classes sous-jacentes par des associations binaires sans propriétés (cas 1).
 - (c) une cardinalité dans 0..1. Les deux cas précédents sont applicables. Si les propriétés migrent dans la classe, il se pose le problème des valeurs partiellement définies.

Application à l'exemple

Les attributs sont représentés par des variables d'instances. Le type des attributs est noté dans le commentaire de la classe. Les types `Integer`, `String` et `Boolean` existent tels quels en Smalltalk. L'héritage ne pose pas de problèmes puisqu'il n'y a pas de cas d'héritage multiple.

Dans la classe `Personne`, on définit deux variables d'instance : `nom` et `prenom`. Le commentaire de la classe indique que leur type est `String`. Il n'y a pas de définition de variables d'instance dans les classes `Joueur` et `JoueurIntelligent` car les variables sont implicitement héritées en Smalltalk.

Dans la classe `Arbitre`, la situation est plus compliquée pour les raisons suivantes :

1. Le tas d'allumettes est représenté par un entier qui donne le nombre d'allumettes dans le tas. C'est une variable d'instance "conditionnelle". Le nombre d'allumettes est fixé aléatoirement au départ du jeu et non à la création de l'arbitre. Le nombre initial d'allumettes varie entre 10 et `NbMaxAllu` où la constante `NbMaxAllu` sera représenté par une variable de classe.
2. Les associations sont représentées sous forme de variables d'instances. Le nom de la variable est le rôle associé s'il existe. Il faut modéliser les contraintes portant sur les associations.
 - (a) La contrainte d'ordre se modélise en choisissant une collection ordonnée pour ranger les joueurs. La taille de cette collection est au maximum de deux, plus précisément

elle contient deux ou aucun objet de la classe `Joueur`. **Toute mise à jour de cette variable devra vérifier cette condition.**

- (b) Le joueur qui a la main est représenté par une variable d'instance. En principe, nous devrions utiliser un objet de la classe `Joueur`. Attention, ici aussi, il peut ne pas y avoir de joueur courant (cardinalité 0..1).

Ces deux variables sont aussi une variable d'instance "conditionnelle" dont l'existence est conditionnée au fait qu'une partie est en cours.

Pour prendre en compte les variables conditionnelles, nous ajoutons une nouvelle variable d'instance booléenne `partieEnCours` dont la valeur conditionnera les autres variables d'instance. Ainsi tout accès en lecture ou écriture des variables conditionnées implique de vérifier leur existence. En cas d'échec, une erreur est levée.

Cette traduction "systématique" peut être améliorée. En effet, nous n'avons pas représenté la contrainte d'inclusion du joueur courant dans les joueurs de la partie en cours. Nous n'avons pas non plus représenté le mécanisme de changement de joueur. La structure la plus pratique en termes de manipulation est en fait le tableau. Le joueur courant sera représenté par un indice dans le tableau à deux dimensions. Sachant que les tableaux ont des indices de 1 à n en Smalltalk, le changement de joueur s'écrit arithmétiquement `courant := ((courant mod 2)+1`.

Les opérations sont traduites en méthodes. Le nommage des méthodes varie par rapport à celui des opérations. Nous utilisons la convention suivante :

1. Opération sans arguments `op():TypeRes` : le sélecteur de la méthode associée à `op` est `op`.
2. Opération avec arguments `op(arg1 : T1, ..., argn : Tn) : TypeRes` : le sélecteur de la méthode associée à `op` est `op + maj1(arg1) + ': ' article(T1) ... maj1(argn) + ': ' article(Tn)`, où `maj1` est une fonction qui met en majuscule la première lettre de son argument et `article` est une fonction qui préfixe par 'a' ou 'an' son argument selon qu'il s'agit d'une consonne ou d'une voyelle. Par exemple, la méthode associée à l'opération `compter(tas : Integer) : Integer` est `compterTas: anInteger`.

Les méthodes d'instances d'accès en lecture sont générées de manière automatique.

La méthode de description par défaut des objets Smalltalk est `printOn: aStream()`. Nous remplaçons donc la méthode `description()` de la classe `Personne` par cette méthode.

La méthode d'instance `compter()` est redéfinie dans la classe `JoueurIntelligent` et masque de ce fait la définition héritée de la classe `Joueur`.

Les méthodes de classes, implicite dans la modélisation abstraite, sont définies, qui tient compte les variables d'instances des classes correspondantes et des contraintes du schéma. La classe `Personne` est abstraite et n'a donc pas de méthode d'instanciation. Dans la classe `Joueur`, la méthode d'instanciation sera `newNom: aString prenom: aString`. Dans la classe `Arbitre`, la définition tient compte de la représentation choisie : il n'y a pas d'arguments car au départ il n'y a pas de partie en cours. Par contre, le tableau des joueurs est créé avec deux positions et la variable `partieEnCours` est initialisée à `false`. C'est la méthode d'instance `nim`, qui initialise les autres variables pour la partie qui démarre.

4.4 Codage

Cette section illustre étape par étape l'écriture du code dans l'environnement `VisualWorks`.

Création d'une classe

Ouvrir un flâneur système depuis la fenêtre `Launcher` par le menu `Browse>All Classes`

ou le bouton .

Ajouter une nouvelle catégorie par le menu central `add...`. Une fenêtre de dialogue s'ouvre. Taper 'Nim' dans cette fenêtre et valider. Si aucune catégorie n'était sélectionnée, la nouvelle catégorie se place à la fin de la liste, sinon elle se place juste au dessus de la catégorie sélectionnée. En principe on range les catégories utilisateur vers la fin de la liste pour laisser les catégories de base de Smalltalk en début de liste.

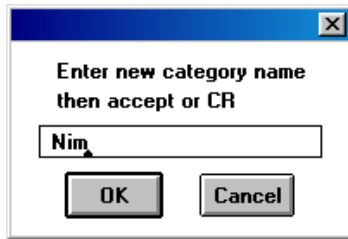


Figure 19 : *Création de la catégorie, jeu de Nim*

Sélectionner si nécessaire la catégorie Nim. La liste des classes de cette catégorie est vide. Dans la fenêtre de texte remplacer le prototype de création de classe

```
NameOfSuperclass subclass: #NameOfClass
instanceVariableNames: 'instVarName1 instVarName2'
classVariableNames: 'ClassVarName1 ClassVarName2'
poolDictionaries: ''
category: 'Jeux'
```

par

```
Object subclass: #Personne
instanceVariableNames: 'nom prenom'
classVariableNames: ''
poolDictionaries: ''
category: 'Nim'
```

Valider par la fonction `accept` du menu central. La liste des classes de cette catégorie contient maintenant une nouvelle classe `Personne`.

Dans la liste des classes, activer la fonction `comment` du menu central. Dans la fenêtre de texte remplacer le prototype de création de commentaire par

Cette classe implante la classe abstraite `Personne` du jeu de Nim.

Les variables d'instance sont :

```
nom : String
prenom : String
```

La méthode de classe `new` aurait du être redéfinie pour interdire la création d'instances. Cependant pour pouvoir appeler la méthode d'instanciation `new` de la classe `Object`, sans passer par un artifice supplémentaire, nous n'avons pas redéfini la méthode `new`. Par défaut, la méthode `new` est donc celle de la classe `Object`.

Valider par la fonction `accept` du menu central.

Création d'une méthode d'instance

Nous allons maintenant créer les méthodes d'accès en lecture et écriture des variables d'instance et la méthode de description.

Dans la liste des protocoles, activer la fonction `add...` du menu central. Une fenêtre de dialogue s'ouvre. Taper 'accessing' dans cette fenêtre et valider. Un nouveau protocole est créé.

Astuce : le nom affiché dans la fenêtre est celui du dernier protocole utilisé. Pour éviter de taper ce nom, passer dans une autre classe, sélectionner le protocole `accessing`, puis revenir dans la classe `Personne`, activer la fonction `add...` du menu central de la fenêtre des protocoles puis valider.

Dans la fenêtre de texte remplacer le prototype de création de méthode Dans la fenêtre de texte remplacer le prototype de création de méthode

```
message selector and argument names
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

par

```
nom
    "rend le nom de la personne"

    ↑nom
```

Formater par la fonction `format` du menu central et valider par la fonction `accept` du menu central. Le formatage et l'acceptation effectuent des contrôles sur l'existence des variables et des méthodes employées. Modifier ensuite le texte précédent par

```
nom: aString
    "affecte le nom de la personne"

    nom ← aString
```

La valeur rendue par défaut est la variable `self` (`^self`). Formater par la fonction `format` du menu central et valider par la fonction `accept` du menu central. Faire de même pour les méthodes `prenom`, `prenom:`.

Pour la méthode de description, on redéfinit la méthode `printOn:`. Pour gagner du temps, sélectionner cette méthode dans le protocole `printing` de la classe `Object` (ou d'une autre classe). Puis copier le contenu de la méthode dans la fenêtre de texte par la fonction `copy` du menu central. Revenir dans la classe `Personne`, activer la fonction `add...` du menu central de la fenêtre des protocoles puis valider. Dans la fenêtre de texte, remplacer le prototype par le texte copié fonction `paste` du menu central et compléter le texte de la méthode par :

```
printOn: aStream
    "Ecrit le nom et le prénom dans le flot."

    aStream cr ; nextPutAll: 'Je m''appelle ', nom, ' ', prenom
```

Formater valider. Les doubles quotes servent à placer une quote. La méthode `cr` place un retour chariot dans le flot de sortie. La méthode `nextPutAll:` place son argument (une collection) dans le flot de sortie. Dans notre exemple, la collection est une chaîne de caractères, obtenue par concaténation (opérateur `,`) d'autres chaînes de caractères.

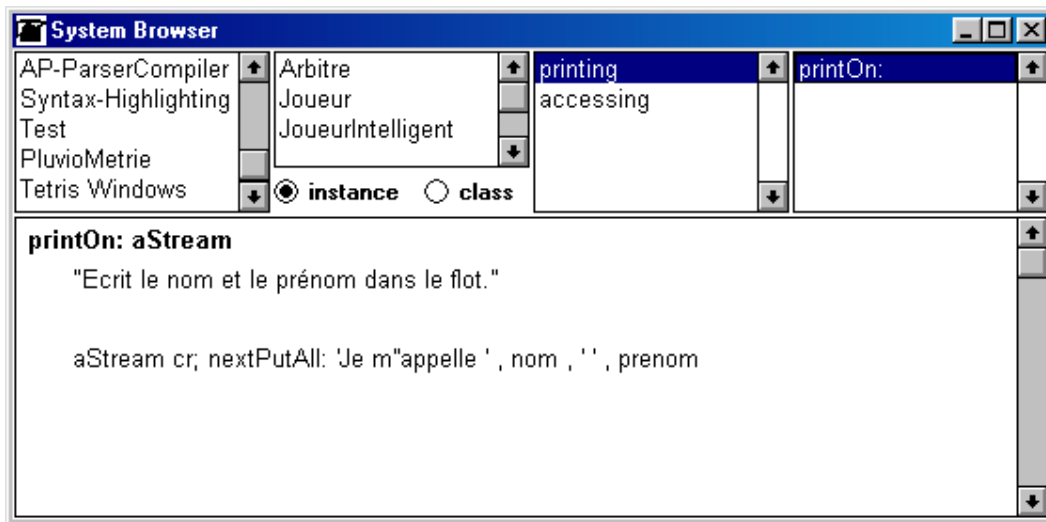


Figure 20 : Une fenêtre flâneur système

Création d'une méthode de classe

Bien que la classe soit abstraite, nous décidons, dans un but pédagogique, de créer une personne par une méthode de classe. Dans le flâneur système, sélectionner le protocole de classe (par le bouton radio **class**. Créer un nouveau protocole (de classe) appelé **examples**. Créer une nouvelle méthode **albert** avec le code suivant :

albert


"Cette méthode crée une personne et affiche ses caractéristiques sur la console. Il s'agit d'un exemple jouet, car en principe la classe Personne est abstraite "

Transcript show: ((Personne new) nom: 'Albert'; prenom: 'Michel'; printString)

La méthode **printString** fait appel à la méthode **printOn:** en redirigeant le flot **aStream** vers une chaîne de caractères.

Evaluer un envoi de message

Ouvrir une fenêtre de travail depuis la fenêtre **Launcher** par le menu **Tools>Workspace**

ou le bouton . Puis taper l'envoi de message **Personne albert** dans la zone de texte. Sélectionner ce texte et évaluer la fonction **do it** du menu central. Le texte **Je m'appelle Albert Michel** apparaît dans la console de la fenêtre **Launcher**.

Sauvegarde des travaux

La sauvegarde du code se fait soit en sauvegardant l'environnement (l'image) soit en générant une version textuelle du code écrit.

Pour enregistrer l'image, activer la fonction **File>Save As...** de la fenêtre **Launcher**.

Pour enregistrer le code dans un fichier texte, activer la fonction **file out as...** du menu central d'une des fenêtres de listes dans un flâneur système. Par exemple, sélectionner la catégorie **Nim**, activer la fonction **file out as...** pour créer un fichier contenant l'ensemble du code des classes de cette catégorie.

Autres classes

Les classes `Joueur`, `Arbitre` et `JoueurIntelligent` sont définies dans la même catégorie dans un flâneur de classes, en respectant les relations d'héritage. En Smalltalk, une classe ne peut être créée que si sa superclasse existe. Ce n'est pas le cas en C++ car les classes sont dans des fichiers séparés. Les classes `Joueur` et `Arbitre` sont créées après la classe `Personne`. La classe `JoueurIntelligent` est créée après la classe `Joueur`. Les méthodes correspondant au jeu sont placées dans un protocole `jeu`. Les méthodes d'instanciation sont placées dans le protocole de classe `instance creation`.

Pour l'arbitre, on considère que les variables d'instances sont privées (pas de méthodes d'accès). L'initialisation de la variable de classe `NbMaxAllu` (nombre maximal d'allumettes dans le tas) est réalisée par une méthode `initialize` placées dans le protocole de classe `initialize release`.

```
initialize
  "Arbitre initialize "
```

```
NbMaxAllu ← 30
```

Attention Pour que que l'initialisation soit effective, il faut évaluer le message `Arbitre initialize`. Par convention, on place ce message dans un commentaire de la méthode et on l'évalue par la fonction `do it` du menu central. Noter que lors d'une inclusion d'une classe dans l'image depuis un fichier texte (voir section 3) la méthode `initialize`, lorsqu'elle est définie, est exécutée.

Voici le code de ces différentes classes.

```
Personne subclass: #Joueur
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Nim'!
```

```
!Joueur methodsFor: 'jeu'!
```

```
compterTas: anInteger
```

```
"le joueur détermine combien il enlève d'allumettes du tas"
"Le calcul aléatoire est possible par la classe Random (résultat dans 0..1)
ou l'utilisation d'une horloge Time now réduite dans l'intervalle 1..3"

| combien |
anInteger = 1 | (anInteger = 2)
  ifTrue: [combien ← 1]
  ifFalse: [anInteger = 3
    ifTrue: [combien ← 2]
    ifFalse: [combien ← (Random new next * 1000 rem: 3) truncated + 1]].
↑combien!
```

```
gagner
```

```
"le joueur se proclame gagnant"

Transcript cr; show: '--> Moi, ', self nom, self prenom, ' j''ai gagné'.
↑self!
```

```
jouerTas: anInteger
```

```
"le joueur joue un coup"

| combien |
```

```

combien ←self compterTas: anInteger.
Transcript cr; show: 'Il y a ', anInteger printString , ' allumette(s)'.
Transcript show: ' moi, ', self nom, self prenom, ' j''enlève ', combien printString , ' allumette(s)'.
↑combien!!
"-----"!

Joueur class
instanceVariableNames: ''!

!Joueur class methodsFor: 'instance creation'!

newNom: aString prenom: aString1
↑(super new) nom: aString; prenom: aString1!

```

Figure 21 : *Implantation Smalltalk de la classe Joueur*

```

Joueur subclass: #JoueurIntelligent
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Nim'!

!JoueurIntelligent methodsFor: 'jeu'!

compterTas: anInteger
"le joueur intelligent détermine combien il enlève d'allumettes du tas"
"Le calcul n'est plus aléatoire "

| combien reste |
reste ←anInteger rem: 4.
reste = 0
ifTrue: [combien ←3]
ifFalse: [reste = 1 | (reste = 2)
ifTrue: [combien ←1]
ifFalse: [combien ←2]].
↑combien!!

```

Figure 22 : *Implantation Smalltalk de la classe JoueurIntelligent*

```

Personne subclass: #Arbitre
instanceVariableNames: 'partieEnCours tas courant joueurs '
classVariableNames: 'NbMaxAllu '
poolDictionaries: ''
category: 'Nim'!
Arbitre comment:

```

'Cette classe implante la classe Arbitre du jeu de Nim.

Les variables d'instance sont :

```

partieEnCours : Boolean
tas : Integer [0..NbMaxAllu]
courant : Integer 1..2
joueurs : Array [1..2] of Joueur

```

La variable partieEnCours indique si une partie est démarrée, elle conditionne les trois variables suivantes.

- tas indique le nombre d'allumettes restant dans le tas


```

- courant donne l'indice du joueur courant dans le tableau joueurs
- joueurs de la partie en cours '!

!Arbitre methodsFor: 'jeu'!

changer
  "l'arbitre donne la main à l'autre joueur"

  partieEnCours
    ifTrue: [courant ← (courant rem: 2) + 1]
    ifFalse: [self error: 'Pas de partie en cours']!

encore
  "rend vrai si la partie n'est pas terminée"

  partieEnCours
    ifTrue: [↑tas > 0]
    ifFalse: [self error: 'Pas de partie en cours']!

enlever
  "l'arbitre demande au joueur courant le nombre d'allumettes à enlever
  et il enlève ces allumettes."

  partieEnCours
    ifTrue: [tas ← tas - ((joueurs at: courant) jouerTas: tas)]
    ifFalse: [self error: 'Pas de partie en cours']!

nimJ1: aJoueur1 j2: aJoueur2
  "l'arbitre crée un nouveau jeu avec un nombre quelconque d'allumettes
  compris dans 10 .. NbMaxAllu. Si une partie était en cours, elle est
  implicitement annulée."

  partieEnCours ← true.
  joueurs ← Array with: aJoueur1 with: aJoueur2.
  courant ← 1. "on aurait pu choisir au hasard"
  tas ← 10 + (Random new next * 10000 rem: NbMaxAllu - 9) truncated + 1.
  [self encore]
  whileTrue:
    [self enlever.
     self changer].
  (joueurs at: courant) gagner.
  ↑self !!

!Arbitre methodsFor: 'initialize-release'!

initialize
  "Au départ, il n'y a aucune partie en cours"

  partieEnCours ← false! !
"-----"!

Arbitre class
  instanceVariableNames: ''!

!Arbitre class methodsFor: 'instance creation'!

newNom: aString prenom: aString1
  ↑(super new) nom: aString; prenom: aString1; initialize !!

```

```

!Arbitre class methodsFor: 'initialize-release'!

initialize
  "Arbitre initialize "

  NbMaxAllu ← 30!

!Arbitre class methodsFor: 'examples'!

unePartie
  "Cette méthode crée un arbitre et deux joueurs, lance le jeu et affiche les résultats sur la console."
  "Arbitre unePartie"

  | j1 j2 arbitre |
  j1 ←Joueur newNom: 'Alain' prenom: 'Térier'.
  j2 ←Joueur newNom: 'Alex' prenom: 'Andrin'.
  arbitre ←self newNom: 'Sémoi' prenom: 'Lechef'.
  arbitre nimJ1: j1 j2: j2!

Arbitre initialize !

```

Figure 23 : *Implantation Smalltalk de la classe Arbitre*

L'évaluation du message `Arbitre unePartie` affiche le résultat suivant :

```

Il y a 20 allumette(s)   moi, AlainTérier j'enlève 3 allumette(s)
Il y a 17 allumette(s)  moi, AlexAndrin j'enlève 3 allumette(s)
Il y a 14 allumette(s)  moi, AlainTérier j'enlève 2 allumette(s)
Il y a 12 allumette(s)  moi, AlexAndrin j'enlève 1 allumette(s)
Il y a 11 allumette(s)  moi, AlainTérier j'enlève 2 allumette(s)
Il y a 9 allumette(s)   moi, AlexAndrin j'enlève 1 allumette(s)
Il y a 8 allumette(s)   moi, AlainTérier j'enlève 1 allumette(s)
Il y a 7 allumette(s)   moi, AlexAndrin j'enlève 3 allumette(s)
Il y a 4 allumette(s)   moi, AlainTérier j'enlève 3 allumette(s)
Il y a 1 allumette(s)   moi, AlexAndrin j'enlève 1 allumette(s)
--> Moi, AlainTérier j'ai gagné

```

L'évaluation du message `Arbitre unePartieInt` affiche le résultat suivant :

```

Il y a 26 allumette(s)  moi, AlainTérier j'enlève 2 allumette(s)
Il y a 24 allumette(s)  moi, AlexLefort j'enlève 3 allumette(s)
Il y a 21 allumette(s)  moi, AlainTérier j'enlève 2 allumette(s)
Il y a 19 allumette(s)  moi, AlexLefort j'enlève 2 allumette(s)
Il y a 17 allumette(s)  moi, AlainTérier j'enlève 1 allumette(s)
Il y a 16 allumette(s)  moi, AlexLefort j'enlève 3 allumette(s)
Il y a 13 allumette(s)  moi, AlainTérier j'enlève 2 allumette(s)
Il y a 11 allumette(s)  moi, AlexLefort j'enlève 2 allumette(s)
Il y a 9 allumette(s)   moi, AlainTérier j'enlève 3 allumette(s)
Il y a 6 allumette(s)   moi, AlexLefort j'enlève 1 allumette(s)
Il y a 5 allumette(s)   moi, AlainTérier j'enlève 3 allumette(s)
Il y a 2 allumette(s)   moi, AlexLefort j'enlève 1 allumette(s)
Il y a 1 allumette(s)   moi, AlainTérier j'enlève 1 allumette(s)
--> Moi, AlexLefort j'ai gagné

```

Faire de même pour les classes `Joueur`, `Joueur`, `Joueur`,

4.5 Conclusion

Dans cette section, nous avons mis en œuvre un programme Smalltalk à partir d'une description abstraite pour illustrer l'utilisation des outils de Smalltalk pour le codage. Nous n'avons pas présenté les outils de mise au point (inspecteurs et débogueurs) bien que nous les ayons utilisés pour tester le programme.

Cet exemple montre qu'il est rapide de programmer en Smalltalk, c'est pourquoi le langage est utilisé pour le prototypage rapide. Il montre aussi que la connaissance des classes du système influence fortement le résultat.

Chapitre 5

Le modèle MVC

Le modèle MVC ou `Model/View/Controller` est la brique de base de construction d'interfaces graphiques en Smalltalk-80.

1 Principes

Le principe est le suivant : le modèle contient les données, la vue les affiche et le contrôleur s'occupe des interactions avec l'utilisateur (clavier, souris). Le principe est simple, mais son application n'est pas triviale. En fait, dans une interface, il y a plusieurs modèles, plusieurs contrôleurs et plusieurs vues qui sont en interaction. Une version très schématique est donnée dans figure 24.

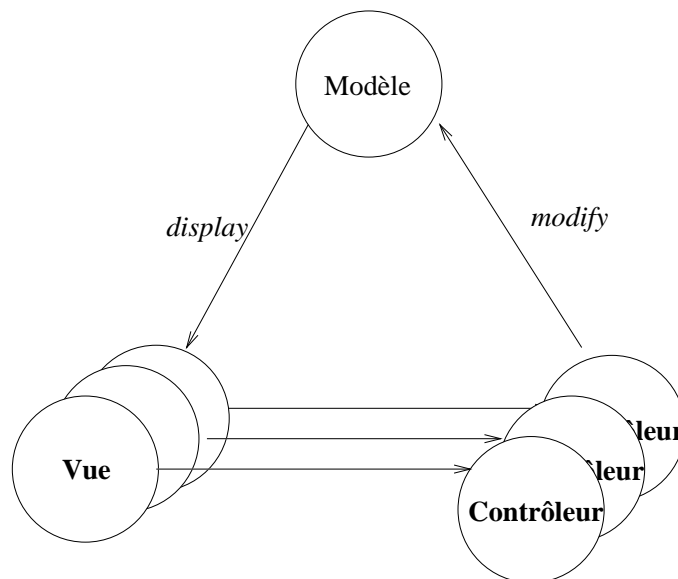


Figure 24 : *Version schématique du modèle MVC*

Contrairement à la programmation sous Windows (Microsoft), le contrôle est réparti dans plusieurs contrôleurs, programmables et personnalisables. Dans Windows, tous les événements sont captés par un contrôleur unique qui associe à chaque événement une action.

A partir de la version 4.0, les auteurs du langage ont pris le principe des dépendants pour alléger les liens entre modèles, vues et contrôleurs. Un modèle est à priori une instance d'une sous-classe de la classe `Model`, qui implémente les relations de dépendance plus efficacement que la classe `Object`. Il est illustré par la figure 25. Son principe est le suivant :

1. L'utilisateur agit sur l'interface par la souris et le clavier.
2. Le **contrôleur** capte cette modification avec une opération.

3. Le **contrôleur** demande éventuellement des informations à l'utilisateur.
4. Muni de ces informations, il informe le **modèle** d'une modification.

```
self model aMethodWith: aParameter
```

5. Le **modèle** effectue la modification sur ses données.

```
aMethodWith: aParameter
```

```
"Exemple fictif : modification du modèle et information du changement"
```

```
self changed: anAspectSymbol with: aParameter
```

6. Le **modèle** informe ses dépendants qu'il a changé (**self changed**). C'est à ce niveau que des distinctions sont faites entre les vues d'un modèle par l'utilisation de mots-clés **self changed: anAspectSymbol**.

```
!Object methodsFor: 'changing'!
```

```
changed: anAspectSymbol with: aParameter
```

```
"The receiver changed. The change is denoted by the argument anAspectSymbol.
```

```
Usually the argument is a Symbol that is part of the dependent's change protocol, that is, some aspect of the object's behavior, and aParameter is additional information. Inform all of the dependents."
```

```
self myDependents update: anAspectSymbol with: aParameter from: self !
```

La variable d'instance **myDependents** est de type **DependentCollection**, s'il s'agit d'un modèle. S'il s'agit au contraire d'un objet normal, un pseudo-codage des dépendants est accessible par la méthode **myDependents** qui lit une variable de classe. C'est le cas de l'exemple de la figure 30. Ce n'est pas une bonne programmation.

7. La **vue** modifie (éventuellement) ses informations par la méthode **update:with:** ou l'une de ses variantes. Puis, elle signale à son container, un cadre instance de **Wrapper** par exemple, qu'elle est modifiée (*damaged*) par **self invalidate**. Il s'en suit une remontée de la hiérarchie des composants visuels qui met à jour l'affichage par **displayOn:**.
8. La **vue** informe le contrôleur de sa modification (**ctrl updateFromView**).

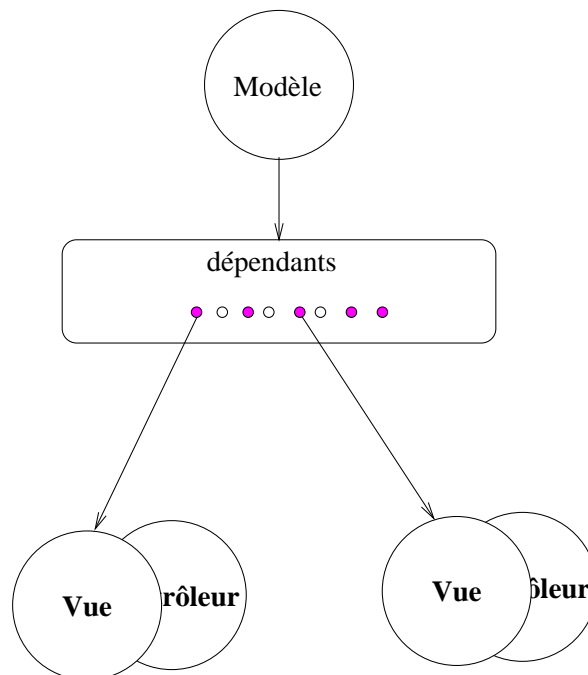


Figure 25 : Le modèle MVC avec les dépendants

Le modèle ne connaît pas ses dépendants. Chaque vue a un contrôleur. La vue connaît son contrôleur et le modèle. Le contrôleur connaît le modèle. Il peut a priori contrôler plusieurs vues. On évite ainsi trop de dépendances circulaires, source de programmes infinis.

La construction se fait sur ce modèle mais aussi sur un modèle simplifié : les **vues enfichables** ou *pluggable views*.

La liaison avec le système de fenêtrage hôte est transparente par la classe `ScheduledWindow` qui va de pair avec la classe `StandardSystemController`. Pour qu'une vue soit affichée dans une fenêtre, il faut qu'elle soit placée dans un **cadre** ou *wrapper* (emballage). Il existe quatre cadres principaux dans le système :

- `TranslatingWrapper` : cadre placé relativement à l'origine de la fenêtre.
- `BoundedWrapper` : cadre délimité par un rectangle extensible.
- `BorderedWrapper` : cadre muni d'un bord.
- `ScrollWrapper` : cadre muni d'ascenseurs.

Lorsque plusieurs vues sur un même modèle ou des modèles différents sont affichées dans une fenêtre, nous utilisons une **vue composite** ou `CompositePart`. Cette vue composite comprend des cadres placés de façon fixe ou proportionnelle dans la vue. Il faut ensuite associer un contrôleur, lorsqu'il ne s'agit pas d'une vue enfichable. Pour cela, on utilise habituellement un **contrôleur avec menu** ou `ControllerWithMenu` ou des boutons (vue enfichable). Le contrôleur par défaut de la fenêtre (vue système) est une instance de la classe `StandardSystemController`. Chaque vue a son contrôleur et réagit donc indépendamment des autres vues. Cependant, lorsque les vues sont organisées hiérarchiquement, le contrôle est lui aussi remonté. Un contrôleur a la structure générique suivante :

```
startUp
```

```
  self controlInitialize .
  self controlLoop.
  self controlTerminate
```

```
avec
```

```
controlInitialize
  self hasControl.
  ↑self
```

```
controlLoop
```

```
  [ self poll .
  self isActive]
  whileTrue:
    [ self controlActivity]
```

```
controlTerminate
```

```
  ↑self
```

```
poll
```

```
  ScheduledControllers checkForEvents.
  self view isOpen
  ifFalse : ["if the view was closed or the top component is closed "
    ScheduledControllers class closedWindowSignal
    raiseRequestFrom: self view topComponent controller].
  self sensor pollForActivity
```

```
checkForEvents
```

```
  view == nil ifTrue: [↑self ].
  view topComponent checkForEvents
```

Cette structure est redéfinie et spécialisée dans les sous-classes. Le lien entre la vue et le contrôleur est souvent décrit dans le cadre `Wrapper`, qui englobe la vue, et plus généralement dans la classe `VisualPart` (et ses sous-classes `DependentPart` (les vues), `CompositePart` et `Wrapper`) qui intercepte les messages vers la vue et le contrôleur.

2 Architecture MVC

L'architecture signifie ici la hiérarchie d'inclusion des composants entre eux. En général, cette hiérarchie est implantée dans les variables d'instance. Il s'agit d'une (ré)utilisation horizontale des composants en opposition avec la (ré)utilisation verticale induite par l'héritage. L'architecture est en générale très complexe et souvent symétrique (le modèle connaît la vue qui connaît le modèle). En l'absence de règles strictes, le programmeur peut construire son application n'importe comment. Rien n'empêche d'utiliser un composant plutôt qu'un autre (exemple : vue à la place de cadre). Pour chacun des exemples suivants, nous donnons l'architecture MVC associée. Rien ne nous certifie que ce soit la meilleure architecture possible. C'est ce qui rend difficile la conception d'une application avec interface graphique.

La figure 26 donne l'inclusion des composants et la figure 33 en présente un exemple. La classe `ScheduledWindow` est une sous classe de `Window` et plus généralement de `DisplaySurface`.

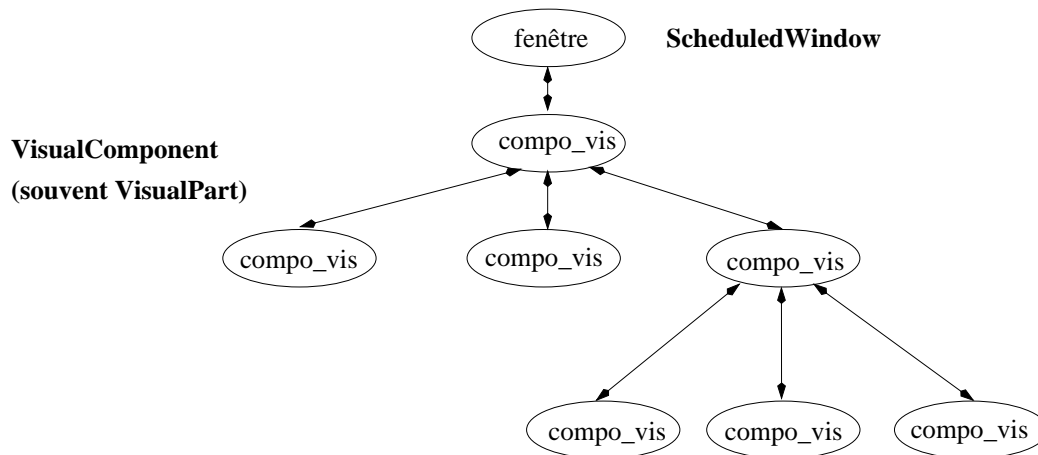


Figure 26 : Architecture générale des composants visuels selon le MVC

3 Exemples d'utilisation du MVC

Les exemples présentés sont des applications simples programmées avec une version "traditionnelle" du MVC dans la version 4.0 d'`ObjectWorks` sous Unix.

3.1 Editeur de trajet

Voici un exemple de vue graphique.

```
TrajetView openOn: Trajet unExemple
```

Il s'agit clairement d'une mauvaise utilisation de la méthode `myDependents` de la classe `Object`. En effet, si un objet est un modèle, alors sa classe doit hériter de `Model`.

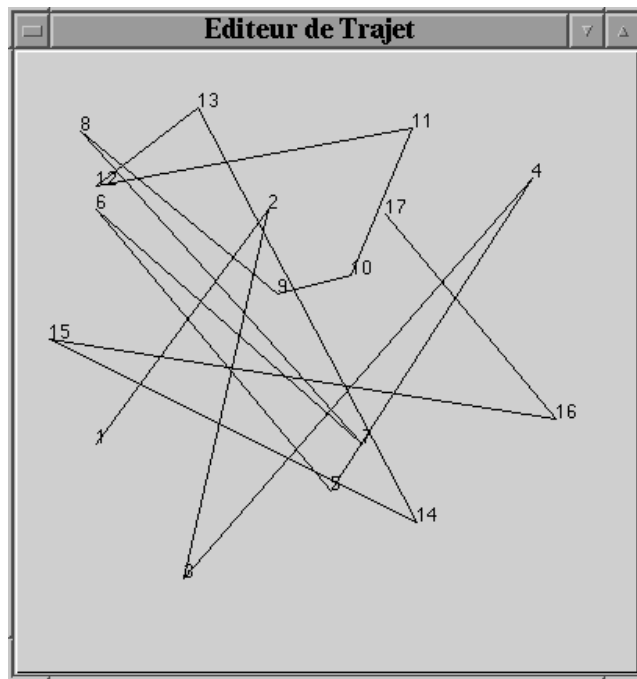


Figure 27 : Un exemple de MVC : l'éditeur de trajets

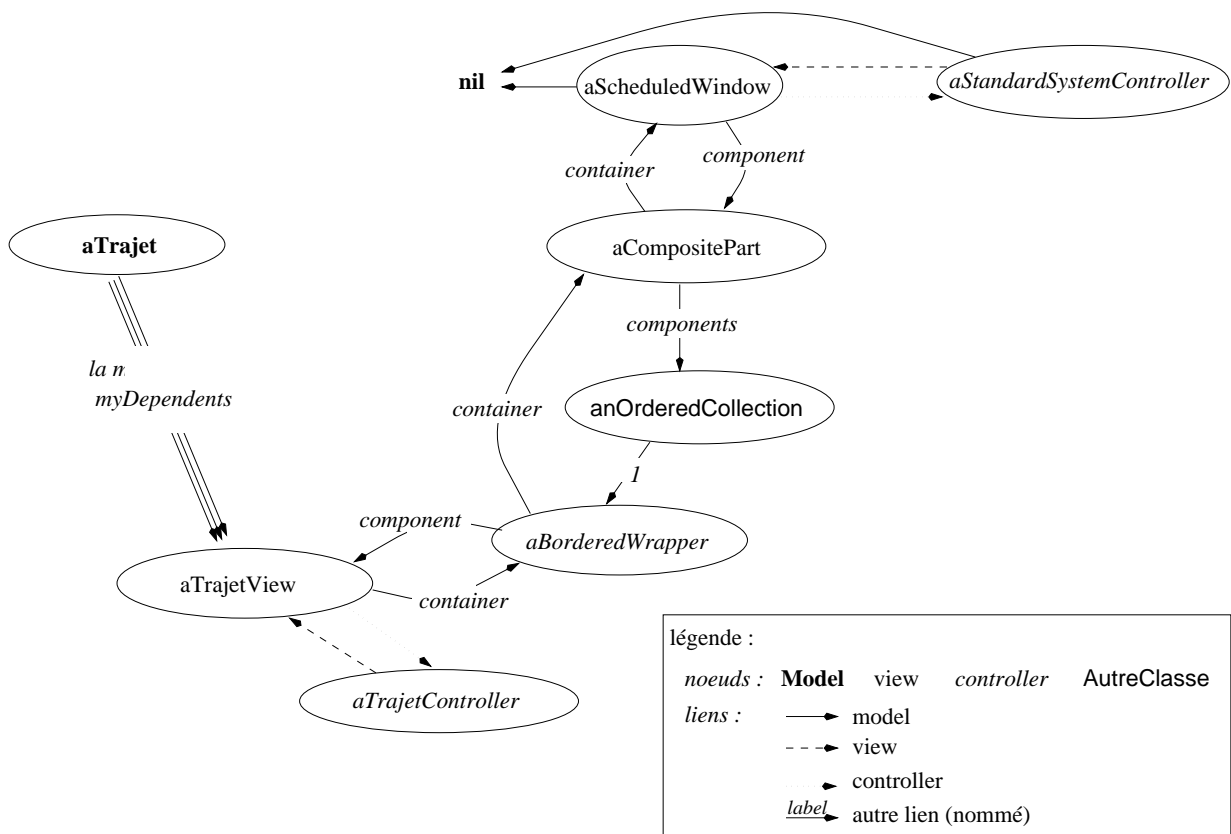


Figure 28 : Un exemple d'architecture MVC : l'éditeur de trajets

3.2 Compteur

Voici un exemple de vue multiple. Le compteur possède deux vues et des boutons.
 CountHolderGlobalView open



Figure 29 : Un exemple de MVC : le compteur

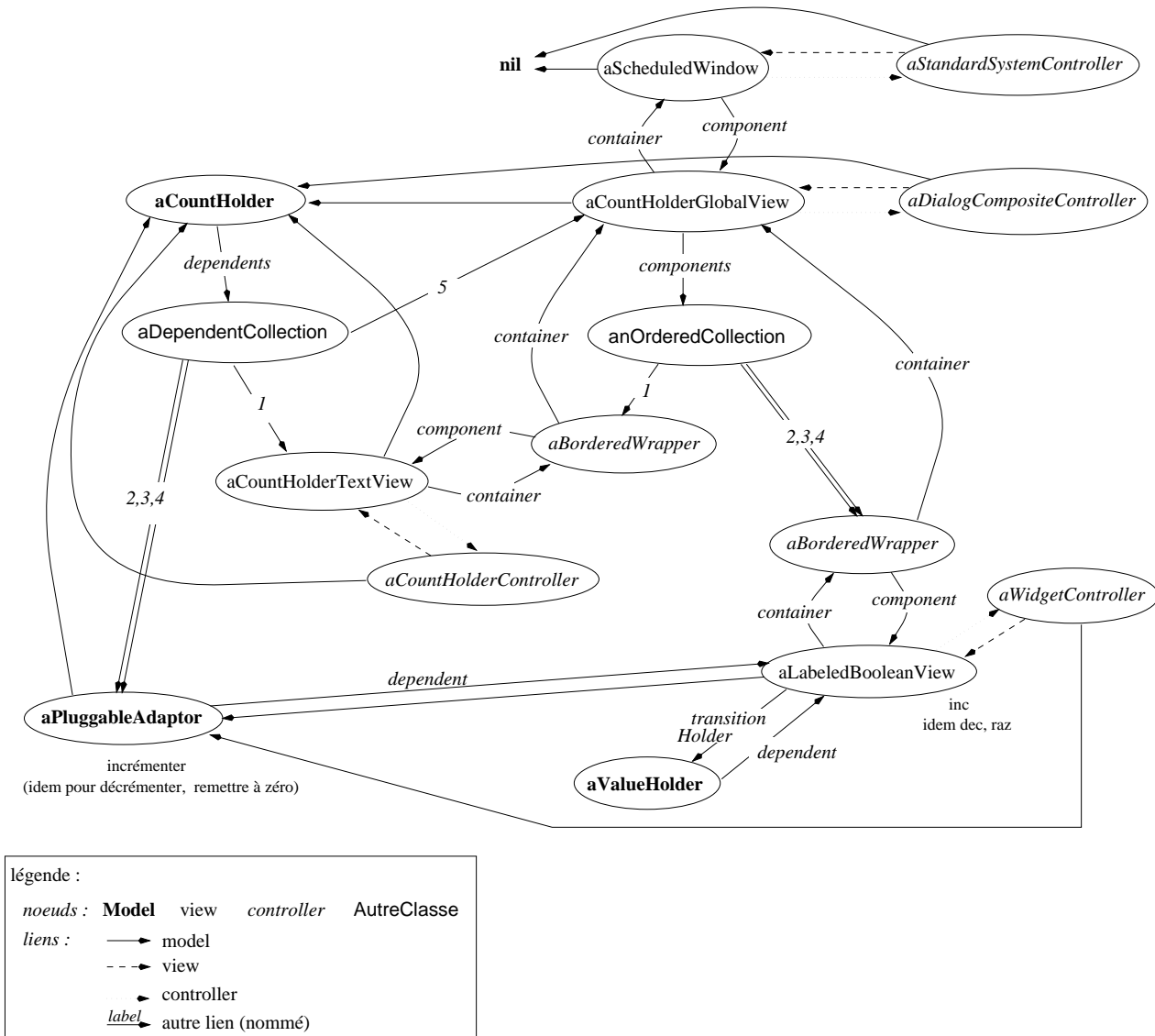


Figure 30 : Un exemple d'architecture MVC : le compteur

3.3 Histogrammes

Voici un exemple de vue avec liste de sélection.

DataView openOn: Data initialize

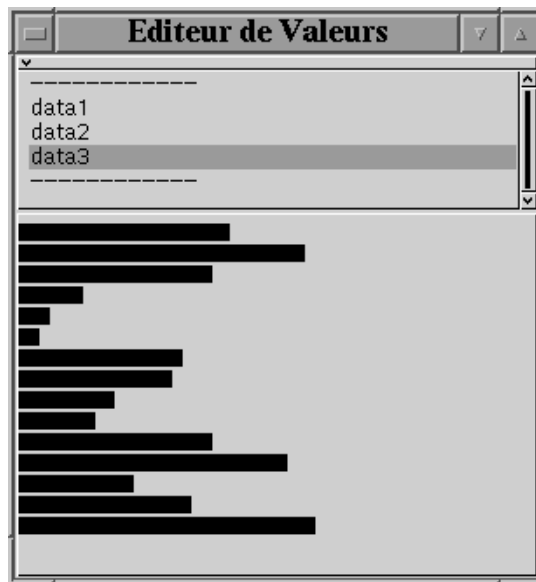


Figure 31 : Un exemple de MVC : les histogrammes

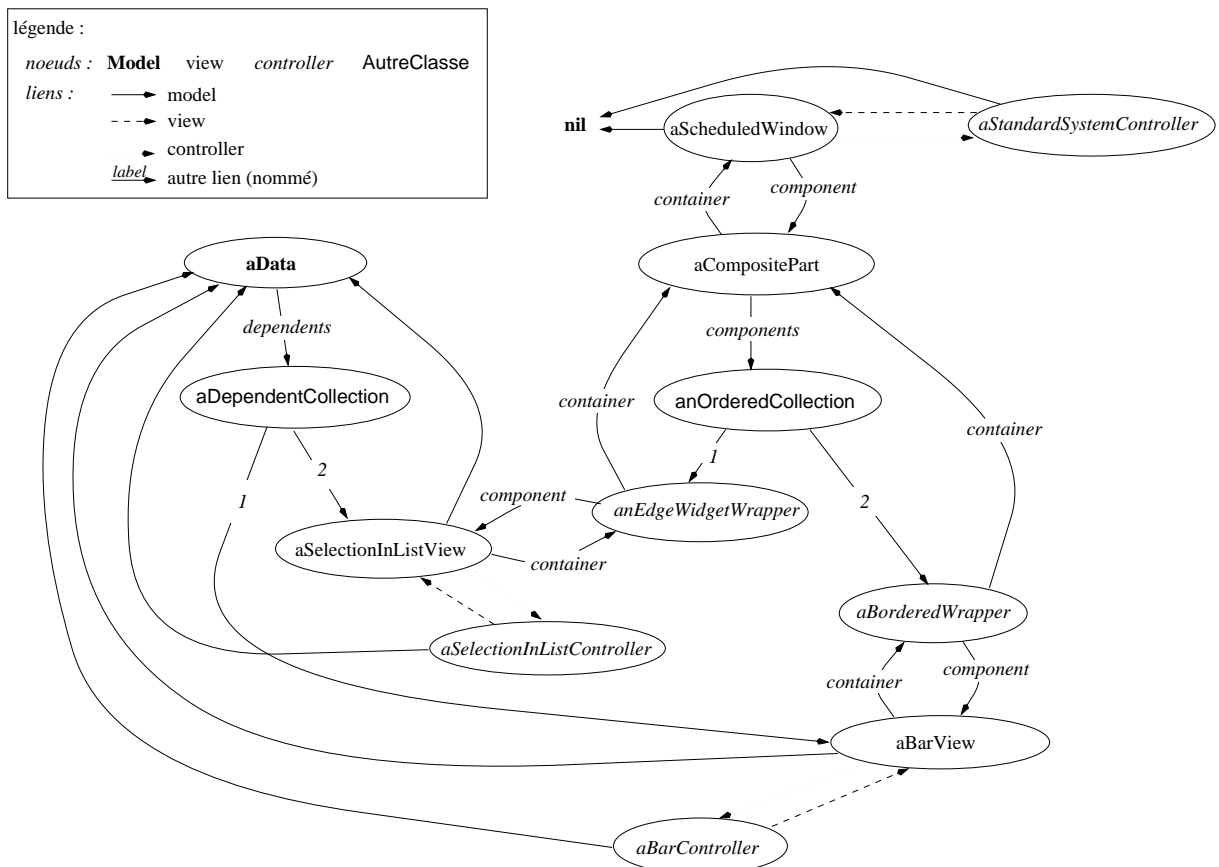


Figure 32 : Un exemple d'architecture MVC : les histogrammes

Attention, une programmation de ces exemples varie avec la version 2.5 de VisualWorks.

4 Précisions sur le MVC

4.1 Exemple d'architecture conseillée

Au travers des exemples ci-dessus et de l'implantation même des composants visuels du système d'interface de Smalltalk, il apparaît que que l'architecture suivante est souhaitable.

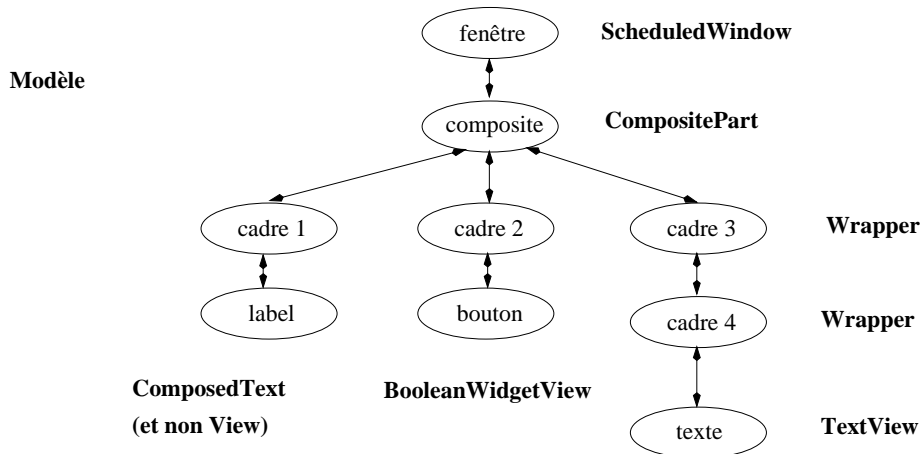


Figure 33 : Un exemple d'architecture générale d'application selon le MVC

Remarque : En Smalltalk, le style de programmation induit est la programmation par l'exemple : le programmeur cherche un exemple correspondant à son besoin et l'adapte. Ce style est difficile car il nécessite une grande connaissance de l'environnement et de la bibliothèque ou des outils de conception de haut niveau tels que les *patterns*¹ décrits dans [GHJV95]. Malheureusement, elle ne marche que si les programmeurs précédents ont fait de la programmation correcte.

4.2 Un peu plus sur les adaptateurs

Les adaptateurs, sont des modèles enfichables sur d'autres modèles.

Les instances de `ValueHolder` sont des modèles génériques pour des objets de base tels que `Boolean`, `Float`, `String`. Le principe des modèles est alors applicable à ces objets de base notamment le protocole d'avertissement `change/update`. Pour créer un modèle booléen pour les boutons par exemple, nous évaluons l'envoi de message suivant : `ValueHolder newBoolean`, qui initialise la valeur par défaut à `false`.

Les instances de `PluggableAdaptor` sont des modèles intermédiaires entre un modèle général et une vue partielle de ce modèle général. Habituellement, un adaptateur se plaque sur un aspect particulier du modèle tel qu'une variable d'instance. La vue partielle a alors pour modèle l'adaptateur. Les adaptateurs(enfichable) permettent d'uniformiser, du point de vue d'une vue, l'accès au modèle. Ainsi, une vue bouton, telle que les instances de `LabeledBooleanView`, peut communiquer avec n'importe quel modèle selon un protocole unique. La vue accède alors à son modèle (intentionnel) via l'interface homogène que lui fournit son modèle (réel). Chaque adaptateur possède trois variables d'instances `getBlock`, `putBlock`, `updateBlock`, qui sont des blocs et assure l'accès à l'aspect d'intérêt du modèle, la mise à jour de cet aspect et le test de la condition de réaffichage de la vue. La vue envoie les messages `value`, `value:`, `update:with:from:` à l'adaptateur qui les réoriente via les blocs vers le modèle. L'interface est identique pour la vue quelque soit le modèle réel auquel elle est attachée via l'adaptateur.

4.3 Lacunes du MVC

Les problèmes habituellement rencontrés avec l'approche traditionnelle du développement traditionnel d'applications peuvent être classés en trois catégories [How95] :

1. construction de l'interface

1. la conception par motif ou pattern a été reprise pour le développement objet par des développeurs Smalltalk.

- La construction est faite entièrement par l'utilisateur. Il y a un manque d'assistance. De plus, il tâtonne jusqu'à trouver la bonne visualisation. Il manque d'outils de prototypage des vues.
2. architecture de l'application
 - En général, un seul modèle pilote l'application et les vues. Il est donc surchargé. La présence d'adaptateurs ne résoud pas ce problème.
 - Un tel modèle diffuse ses modifications aux (nombreux) dépendants. Beaucoup de ces dépendants ne sont pas concernés par telle et telle modification. Il y a perte d'efficacité et contrôle systématique du paramètre de modification et donc perte d'efficacité.
 - La vue doit connaître des informations spécifiques du modèle (méthodes de lecture et écriture de l'aspect). Il y a un manque d'uniformité qui oblige à enrichir inutilement le comportement des méthodes.
 - Certaines vues comme les boutons nécessitent obligatoirement des adaptateurs.
 - L'ouverture de l'interface est en général une méthode volumineuse et procédurale, souvent redondante.
 - Cette ouverture (création de vue) est souvent invoquée dans le modèle, ce qui n'est pas son rôle.
 - Le modèle contient ainsi des informations du domaine (ce qu'on modélise) et de l'application (interface, ...).
 - Les objets intéressés par un aspect d'un modèle doivent se connecter comme dépendant du modèle et donc surcharger le protocole `change/update`.
 - Chaque modification du modèle donne lieu à un cas dans la méthode `change`. L'alternative multiple est rarement un code efficace.
 3. manque de variété et de flexibilité
 - Il manque de support d'édition de valeurs autres que du texte (date, entiers...).
 - Il n'y a pas de support de communication entre vues (batterie de boutons, séquence de tables...).
 - Il n'y a pas de support pour la gestion d'une barre de menu, habituelle dans la plupart des interfaces utilisateur natives (Mac, Dos).
 - Il n'y a pas de support pour gérer dynamiquement les interfaces (ex : options inaccessible (grisées)...).
 - Il n'y a pas de support pour les applications en multi-fenêtrage.
 - Il n'y a pas de support de réutilisation d'interfaces (autre que couper/coller).
 - La personnalisation des dialogues est difficile.
 - Il manque des outils habituels d'interfaces tels que les champs de saisie, les saisie avec validation, les boutons menus, ...).

VisualWorks vise à améliorer les points ci-dessus.

5 Interfaces MVC avec VisualWorks

Dans cette section nous développons l'essentiel du MVC avec **VisualWorks**. Nous insistons sur les outils de construction d'interface MVC, le support à la génération d'application. De ce fait, Smalltalk est un bon outil de prototypage rapide. Des compléments seront trouvés dans [HH95, How95, Sha97].

5.1 Préambule : qu'est-ce qui a changé ?

L'apport principal de **VisualWorks** concerne le modèle MVC, dont nous avons présenté les principes dans la section 5. Citons, entre autres nouveautés, la séparation du modèle de l'application qui le concerne, l'introduction d'une meilleure gestion des Entrées/Sorties dans les

contrôleurs avec une prise en compte plus poussée des événements claviers, la gestion d'applications visuelle (construction et génération d'interface, installation d'une interface dans un "look" donné (Mac, Windows, X) pour une application donnée, etc.). Les nouvelles façons de faire sont à la fois plus compliquées à comprendre mais plus simples à mettre en oeuvre. En effet, le MVC traditionnel cohabite avec le nouveau MVC mais la compatibilité ascendante n'est pas entièrement assurée comme le montre le chargement et l'application dans VisualWorks des exemples des sections 3.2 3.1 3.3. Dans ce cas, les fenêtres s'affichent correctement, mais les contrôleurs sont inopérants. Si la notion de composant s'est renforcée dans VisualWorks, elle apparaissait déjà avec le concept de vue enfichable de la classe `PluggableAdaptor` et les modèles atomiques de la classe `ValueHolder` sous-classes de la classe `ValueModel`. Précisons quelques points sur le modèle MVC.

5.2 Concepts de base

Comme pour le modèle MVC, nous ferons abstraction ici des opérations graphiques et de l'affichage réel bien qu'elles soient utilisées par les nouveaux outils de VisualWorks. Consulter pour cela les classes `GraphicContext`, `Mask`, `Pixmap`, `Paint` ou encore `Image`.

VisualWorks a fait table rase du passé avec de nouvelles classes pour les composants visuels et les contrôleurs, qui réutilisent peu de choses des interfaces précédentes, selon [How95]. Ce qui a permis à *VisualWorks* de standardiser et simplifier la construction des interfaces. Maintenant, toutes les vues et modèles utilisent le même moyen de communication appelé, *value model protocol*.

Composants, cannevas et *widgets*

Une fenêtre est un assemblage de **composants visuels**. Chacun de ces composants visuel constitue un bloc d'interface.

Définition 5.1 (composant visuel) *Un composant visuel est un élément d'une fenêtre (Window) ou d'une partie visuelle (VisualPart) qui se dessine dans une surface d'affichage de fenêtre.*

Un composant VisualWorks est une spécification de cadre (spec wrapper), c'est-à-dire une forme particulière de cadre (classe `SpecWrapper`) développée pour VisualWorks. Tous les composants visuels ont donc un "truc" ou widget (variable d'instance de `SpecWrapper`)

Définition 5.2 (cannevas) *Un cannevas (canvas) est un composite d'une fenêtre dont les composants sont des composants VisualWorks liés à une même application [How95]. Le cannevas est soit en mode édition (paint), soit en mode exécution (runtime).*

Définition 5.3 (widget) *Un widget (gadget ou "truc") est une partie visuelle (visual part) responsable de la représentation visuelle d'un composant. Par exemple, l'interface comprend des labels, des boutons, des vues, des zones de texte... Le protocole d'utilisation comprend l'affichage, la visibilité, les préférences visuelles et la diffusion de messages vers les composants. L'état widget du précise ces attributs.*

Les *widgets* caractérisent les composants. Les *widgets* sont classés en quatre catégories : les *widgets* passifs (labels, les séparateurs, groupes) les boutons et potentiomètres, les textes et les sélections. La hiérarchie des *widgets* montre ces catégories. Elle varie avec les versions de *VisualWorks* comme le montrent les figures 34 et 35.

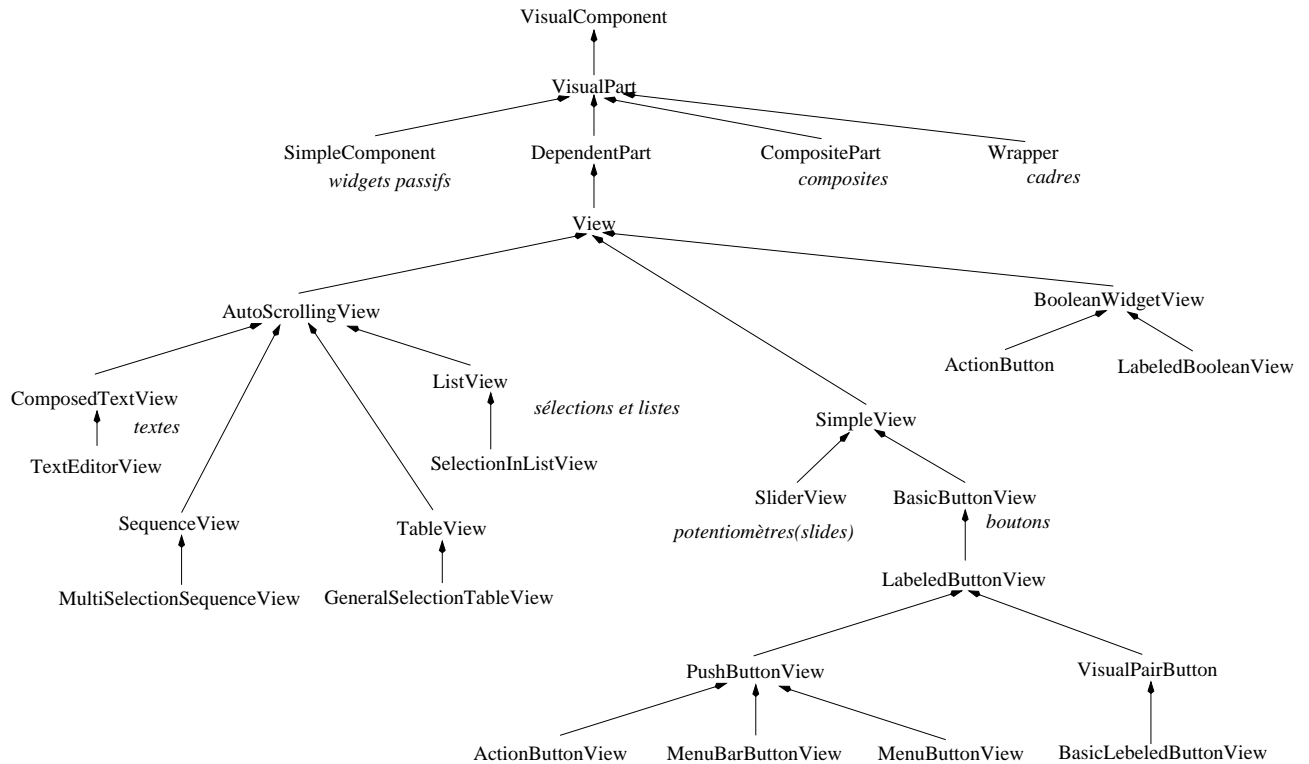


Figure 34 : La hiérarchie des widgets sous VisualWorks version 1

Cinq catégories de composants sont distinguées :

- les composants **passifs** dont le *widget* n'a ni contrôleur ni modèle (*widgets* passifs).
- les composants **actifs** dont le *widget* est une vue,
- les composants **arbitraire** dont le *widget* est une partie visuelle quelconque (i.e. pas forcément un composant *VisualWorks*),
- les composants **composites**, dont le *widget* est une partie (composite) composée de plusieurs composant *VisualWorks*,
- les **sous-cannevas** (*subcanvas*) qui sont des cannevas inclus dans d'autres cannevas.

Les composants *VisualWorks* sont définis par un **cadre de spécification** (*spec wrapper*), comprenant le *widget*, sa décoration (Mac, Windows, Motif...), une copie de son état (visible, ...) et une description textuelle appelée **spécification de composant** (*component spec*) instance de **ComponentSpec**. Cette spécification est indépendante de l'interface de fenêtre utilisée. Elle indique l'organisation logique des composants et *widgets*. Lors de la phase de construction (passage de l'édition du cannevas à son exécution), une présentation particulière est alors donnée (Mac, Windows, Motif...). La spécification de composant est représentée par des collections de symboles. C'est la représentation la plus abstraite, il n'y a pas d'objets mis en cause mais uniquement des tableaux de littéraux (chaînes de caractères, symboles, entiers, réels....). Par exemple, dans l'exemple de la section 5.4, l'interface est définie par la méthode

Listing 5.1 – Un exemple de spécification de composant

```
!CalculatorExample class methodsFor: 'interface specs'!
```

```
windowSpec
```

```
"UIPainter new openOnClass: self andSelector: #windowSpec"
```

```
↑#(#FullSpec
```

```
  #window: #(#WindowSpec #label: ' ' #min: #(#Point 225 218 )
```

```
  #max: #(#Point 581 218 )
```

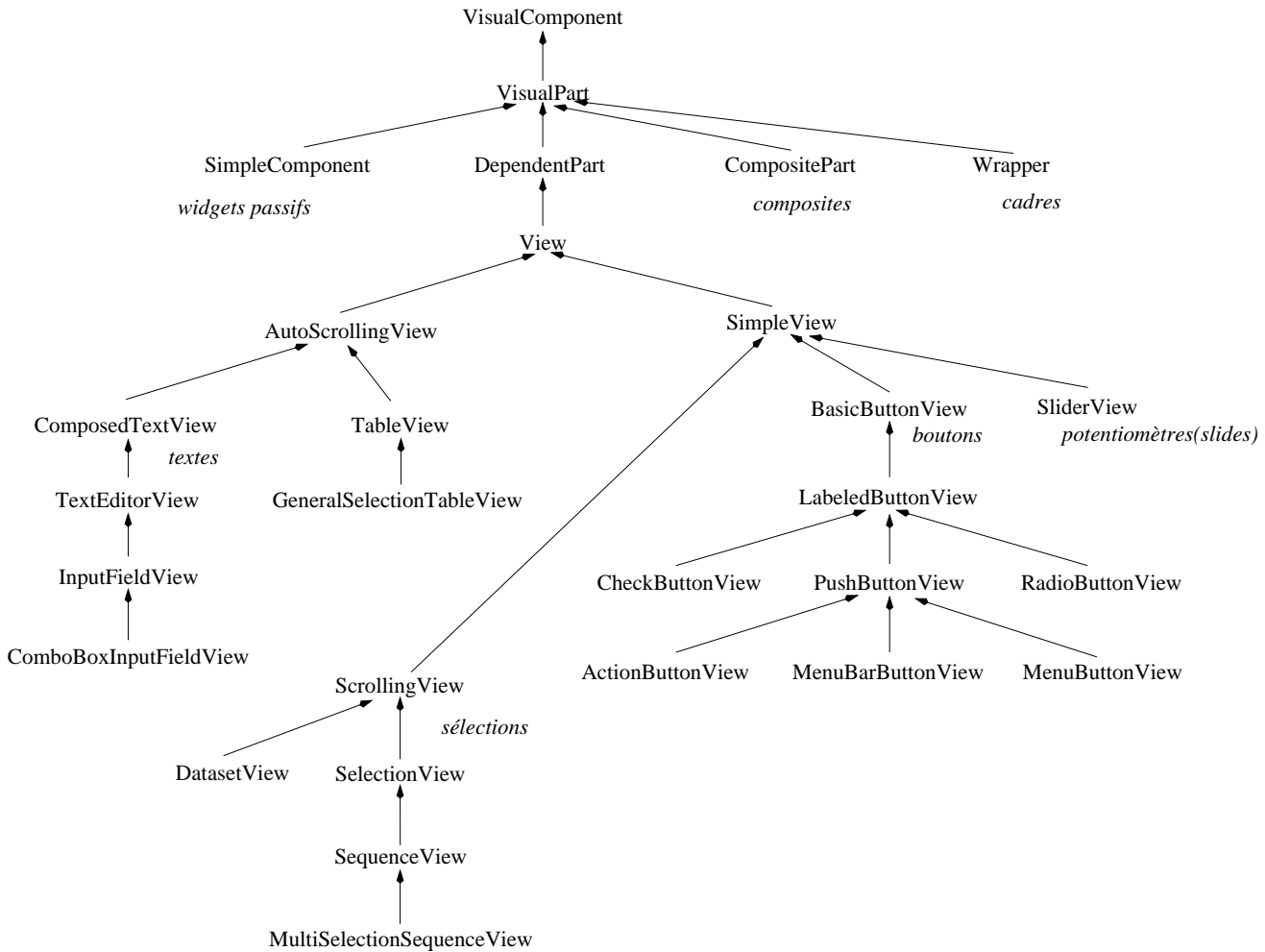


Figure 35 : La hiérarchie des widgets sous VisualWorks version 2

```

#bounds: #(#Rectangle 115 101 340 319 )
#component: #(#SpecCollection
#collection: #(
  #(#ActionButtonSpec #layout: #(#Rectangle 98 43 138 67 )
    #name: #cos #model: #cos #tabable: false #label: 'cos' )
  #(#ActionButtonSpec #layout: #(#Rectangle 50 43 90 67 )
    #name: #sin #model: #sin #tabable: false #label: 'sin' )
  #(#ActionButtonSpec #layout: #(#Rectangle 146 43 186 67 )
    #name: #tan #model: #tan #tabable: false #label: 'tan' )
  #(#ActionButtonSpec #layout: #(#Rectangle 64 80 96 104 )
    #name: #eight #model: #eight #tabable: false #label: '8' )
  #(#ActionButtonSpec #layout: #(#Rectangle 104 80 136 104 )
    #name: #nine #model: #nine #tabable: false #label: '9' )
  #(#ActionButtonSpec #layout: #(#Rectangle 24 112 56 136 )
    #name: #four #model: #four #tabable: false #label: '4' )
  #(#ActionButtonSpec #layout: #(#Rectangle 64 112 96 136 )
    #name: #five #model: #five #tabable: false #label: '5' )
  #(#ActionButtonSpec #layout: #(#Rectangle 104 112 136 136 )
    #name: #six #model: #six #tabable: false #label: '6' )
  #(#ActionButtonSpec #layout: #(#Rectangle 24 144 56 168 )
    #name: #one #model: #one #tabable: false #label: '1' )
  #(#ActionButtonSpec #layout: #(#Rectangle 64 144 96 168 )
    #name: #two #model: #two #tabable: false #label: '2' )
)

```



```

#(#ActionButtonSpec #layout: #(#Rectangle 104 144 136 168 )
  #name: #three #model: #three #tabable: false #label: '3' )
#(#ActionButtonSpec #layout: #(#Rectangle 24 176 56 200 )
  #name: #zero #model: #zero #tabable: false #label: '0' )
#(#ActionButtonSpec #layout: #(#Rectangle 64 176 96 200 )
  #name: #dot #model: #dot #tabable: false #label: '.' )
#(#ActionButtonSpec #layout: #(#Rectangle 104 176 136 200 )
  #name: #invert #model: #invert #tabable: false #label: '+-' )
#(#ActionButtonSpec #layout: #(#Rectangle 24 80 56 104 )
  #name: #seven #model: #seven #tabable: false #label: '7' )
#(#ActionButtonSpec #layout: #(#LayoutFrame -48 1 8 0 -10 1 32 0 )
  #name: #clear #model: #clear #tabable: false #label: 'C' )
#(#ActionButtonSpec #layout: #(#Rectangle 181 112 213 136 )
  #name: #divide #model: #divideOperator #tabable: false #label: '/' )
#(#ActionButtonSpec #layout: #(#Rectangle 181 80 213 104 )
  #name: #times #model: #multiplyOperator #tabable: false #label: 'x' )
#(#ActionButtonSpec #layout: #(#Rectangle 149 112 181 136 )
  #name: #plus #model: #plusOperator #tabable: false #label: '+' )
#(#ActionButtonSpec #layout: #(#Rectangle 149 80 181 104 )
  #name: #minus #model: #subtractOperator #tabable: false #label: '-' )
#(#InputFieldSpec #layout: #(#LayoutFrame 8 0 8 0 -62 1 32 0 )
  #name: #accumulator #model: #accumulator #alignment: #right
  #isReadOnly: true #type: #number )
  #(#GroupBoxSpec #layout: #(#Rectangle 14 75 146 205 ) )
#(#ActionButtonSpec #layout: #(#Rectangle 149 144 181 168 )
  #name: #enter #model: #enter #tabable: false #label: '=' ) ) ) )

```

Comme tous les cadres, le cadre de spécification a un composant et un contenu. Le contenu est un autre cadre, le composant est par exemple la fenêtre de l'application. Des exemples de cadres de spécification sont présentés dans les figures 36, 40 et 41.

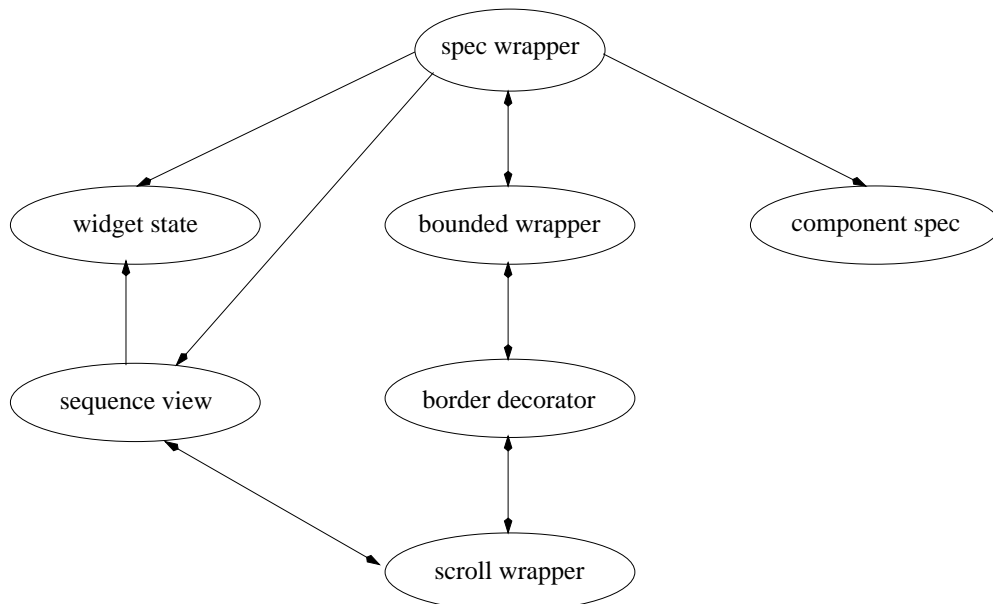


Figure 36 : Un exemple d'architecture MVC VisualWorks

Modèles

Les **modèles** (*model*) prennent des rôles spécifiques en *VisualWorks*. On distingue deux grandes catégories de modèles : les **modèles d'application** et les **modèles d'aspect**. Un modèle d'application gère une partie de l'interface de l'application (voir section 5.2). Un

modèles d'aspect gère un aspect de l'information pour un seul composant *VisualWorks*. Un modèle d'application contient plusieurs modèles d'aspects.

Une classification orthogonale à la précédente est qualifiée de modèles de valeurs. Un **modèle de valeurs** (*value model*) détient un aspect primaire d'information : sa valeur. Nous avons ainsi un mode de communication simplifié pour la consultation de valeur et la propagation des changements. Plusieurs variantes existent :

- Une instance de `ValueHolder` pointe sur un objet quelconque.
- Une instance de `AspectAdaptor` référence une valeur d'un objet (son sujet) avec des méthodes d'accès.
- Une instance de `PluggableAdaptor`, qui précédemment servait à tout, sert maintenant à établir une communication entre deux objets qui ne se connaissent pas (convertisseur de type, interpréteur de message, ...).
- D'autres catégories existent dans les versions récentes de *VisualWorks*.

Contrôleurs

Nous n'allons pas nous étendre sur les contrôleurs. Notons seulement une meilleure prise en compte des événements clavier et la prise en compte de barres de menu.

Constructeur d'interfaces

Le **constructeur** (*builder*) permet de séparer l'interface du modèle (inconvenient du MVC traditionnel). Le rôle du constructeur est de construire une interface homme/machine selon des spécifications de composants ou d'objets. Il sert aussi à la création de l'interface par les outils d'édition de cannavas et à son accès à l'exécution.

L'édition de cannavas produit une spécification de composant indépendante de l'application et du système de fenêtrage. Par exemple, la spécification d'une fenêtre d'une application est instance de `FullSpec` et comprend une spécification de fenêtre (*window spec*), et la collection des spécifications des composants de la fenêtre. L'exemple de la figure 5.1 illustre une telle spécification.

La construction des composants se fait en ajoutant la spécification au constructeur par la méthode `add`. L'ouverture de l'interface se fait par le méthode `open`. La construction de l'interface nécessite des ressources en plus des spécifications.

Définition 5.4 (ressource) Une *ressource* (*resource*) est un objet dont le constructeur a besoin pour construire l'interface selon les spécifications. Les ressources courantes sont les menus, tables, images et modèles.

Deux classes de ressources sont distinguées : celles qui évoluent, appelées **ressources dynamiques** et celles qui n'évoluent pas pendant l'exécution, appelées **ressources statiques**. Les premières correspondent souvent à des modèles de valeurs. La **source** (*source* d'un constructeur est un modèle d'application (instance d'une sous-classe de `ApplicationModel` (voir section 5.2). Le **client** (*client*) d'un constructeur est l'objet qui a créé le constructeur, c'est souvent la source elle-même.

Modèle d'application

L'**architecture en modèles d'application** (*application model architecture*) est l'architecture pour concevoir, construire et exécuter des applications en *VisualWorks*.

Définition 5.5 (Modèle d'application) Un *modèle d'application* (*application model*) est un modèle spécifiquement chargé de gérer une application².

2. Concept proche de la procédure principale, *main* en C.

Un modèle d'application est en principe responsable d'une seule fenêtre tout en déléguant une partie de l'interface à d'autres objets comme le montre la figure 37. L'application, au sens habituel du terme, est donc un ensemble de modèles d'application.

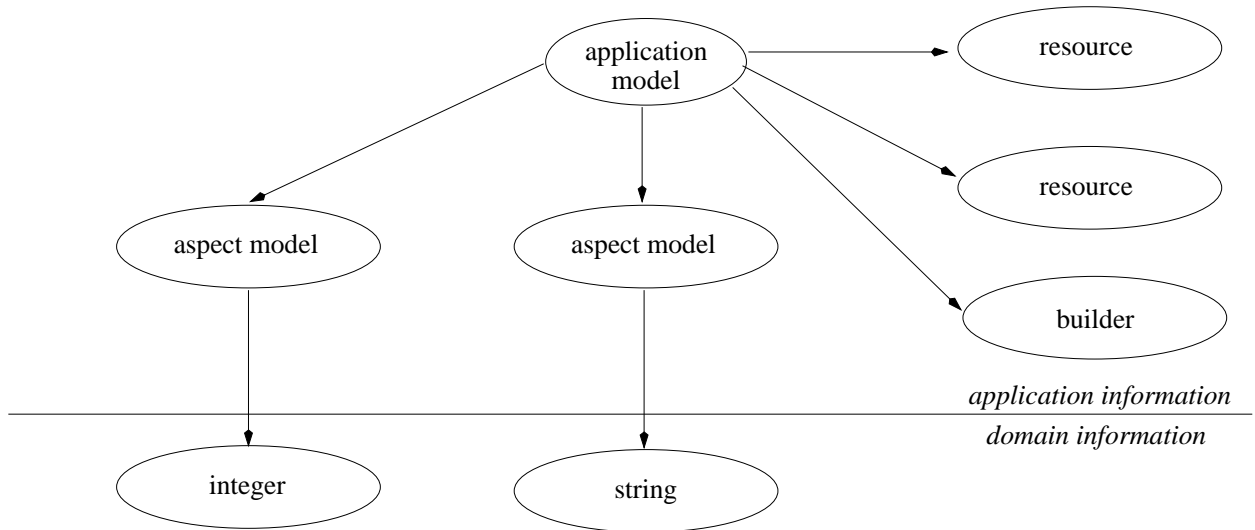


Figure 37 : Diagramme des objets du modèle d'application

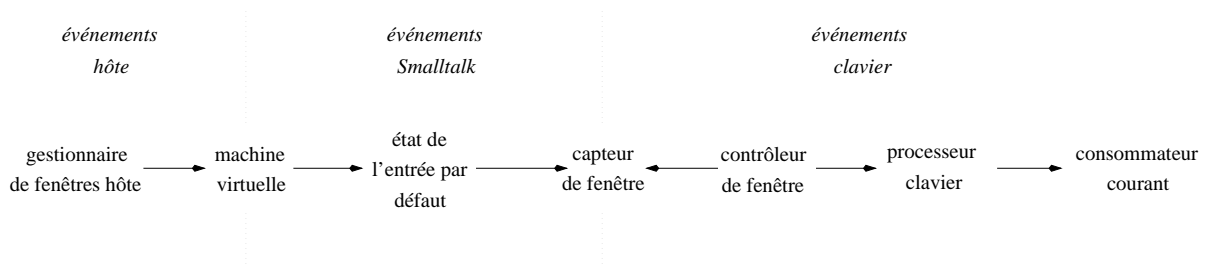
Le modèle d'application délègue l'aspect MVC aux modèles d'aspect. Le domaine des informations est ainsi indépendant de l'application elle-même et donc plus stable. Le seul dépendant, en général, de l'application est la fenêtre. Les autres relations sont réparties indépendamment dans les modèles d'aspect ; ce qui évite des diffusions inutiles de messages de mise-à-jour.

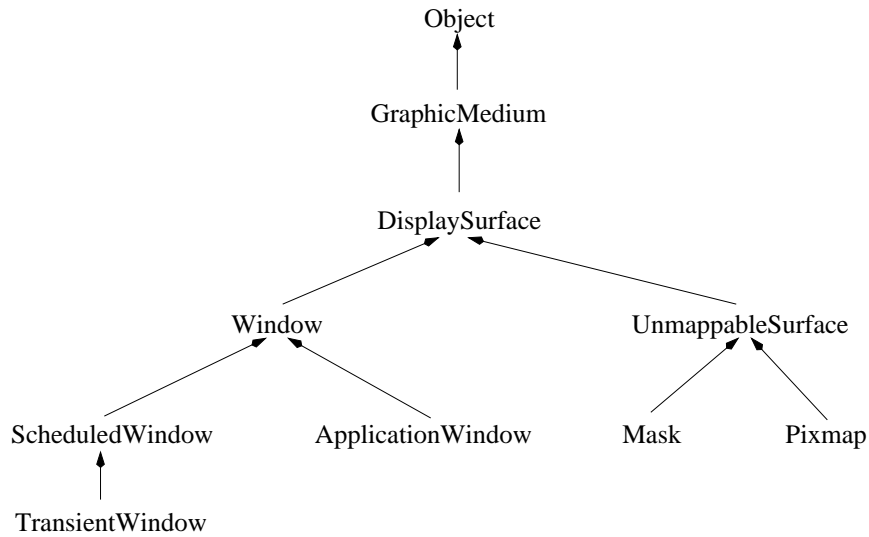
Le modèle d'application délègue l'aspect interface au constructeur d'interface. Si plusieurs fenêtres sont gérées, il faut alors autant de constructeurs. Le modèle fournit les ressources au constructeur (modèles d'aspects, labels, menus...). Il contient aussi les méthodes à exécuter dans les menus.

Fenêtres

Les fenêtres sont des surfaces graphiques modulaires. Depuis la version 4 de Smalltalk-80, elles ne sont plus liées à une fenêtre Smalltalk mais interfacées directement avec l'environnement de fenêtrage hôte. En *VisualWorks*, la classe `ApplicationWindow` est dédiée aux **fenêtres** liées à une application. .

La fenêtre est responsable des interactions souris (*event-handling mechanism*) et clavier (*keyboard-handling mechanism*), de l'affichage des informations sur la surface d'affichage et de l'ordonnancement et la manipulation de la fenêtre. Nous retrouvons là, les mécanismes habituels du MVC (vues et contrôleurs). La prise en compte des événements est la suivante :



Figure 38 : *Hiérarchie des fenêtres*

5.3 Méthode de construction d'une interface

Voici les étapes de la construction d'une application avec *VisualWorks*.

1. Décrire le problème.
2. En donner une spécification.
3. Créer les fenêtres du problème avec l'outil de dessin (**Painter**).
 - Dessiner le canevas.
 - Editer les propriétés des différents éléments (labels, boutons, menus).
 - Installer le canevas
 - Créer les barres de menus.
 - Créer les fenêtres de dialogue.
4. Créer les classes du modèle du domaine.
 - Structure et commentaires.
 - Protocole de classe et méthodes.
 - Protocole d'accès.
 - Protocole d'actions.
5. Lier le code Smalltalk aux composants de l'interface.
6. Propriétés des aspects et des actions.
7. Compléter les éléments.
8. Tester l'application.

5.4 Exemples

Les exemples présentés dans cette section sont livrés avec l'environnement *VisualWorks 2.5*. Intégrer les fichiers sources dans votre environnement depuis le répertoire `RepVisual\TUTORIAL\BASIC`.

Calcullette

Voici un exemple de vue multiple. La calcullette possède une vue d'affichage du résultat et des boutons pour chaque digit et chaque opération. Nous présentons la calcullette de *VisualWorks 1.0* sous Unix puis celle *VisualWorks 2.5* sous Windows, à titre de comparaison.

```
CalculatorExample open
```



Figure 39 : Un exemple de MVC sous VisualWorks 1.0 : la calculette

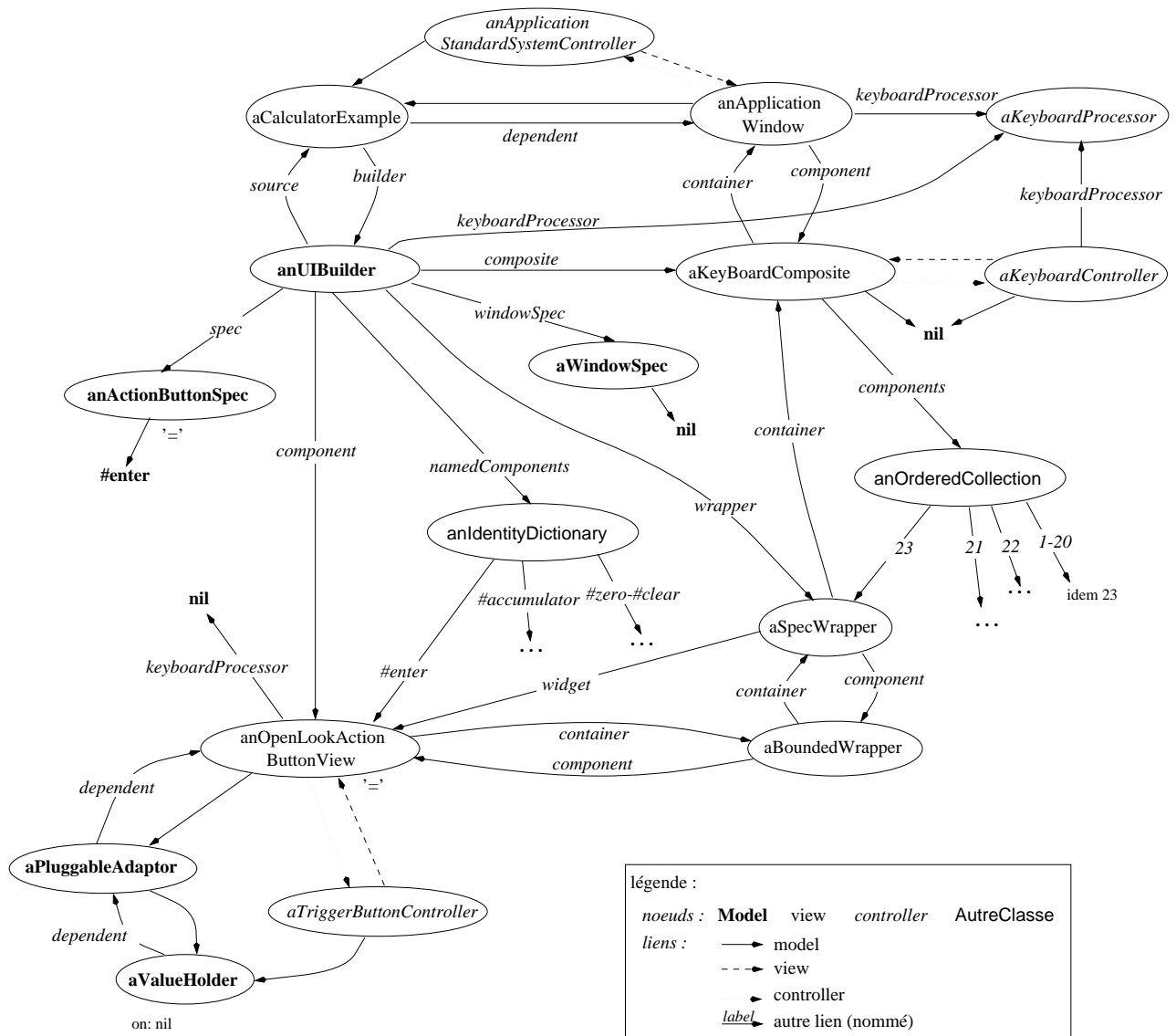


Figure 40 : Un exemple d'architecture MVC VisualWorks : la calculette 1/2

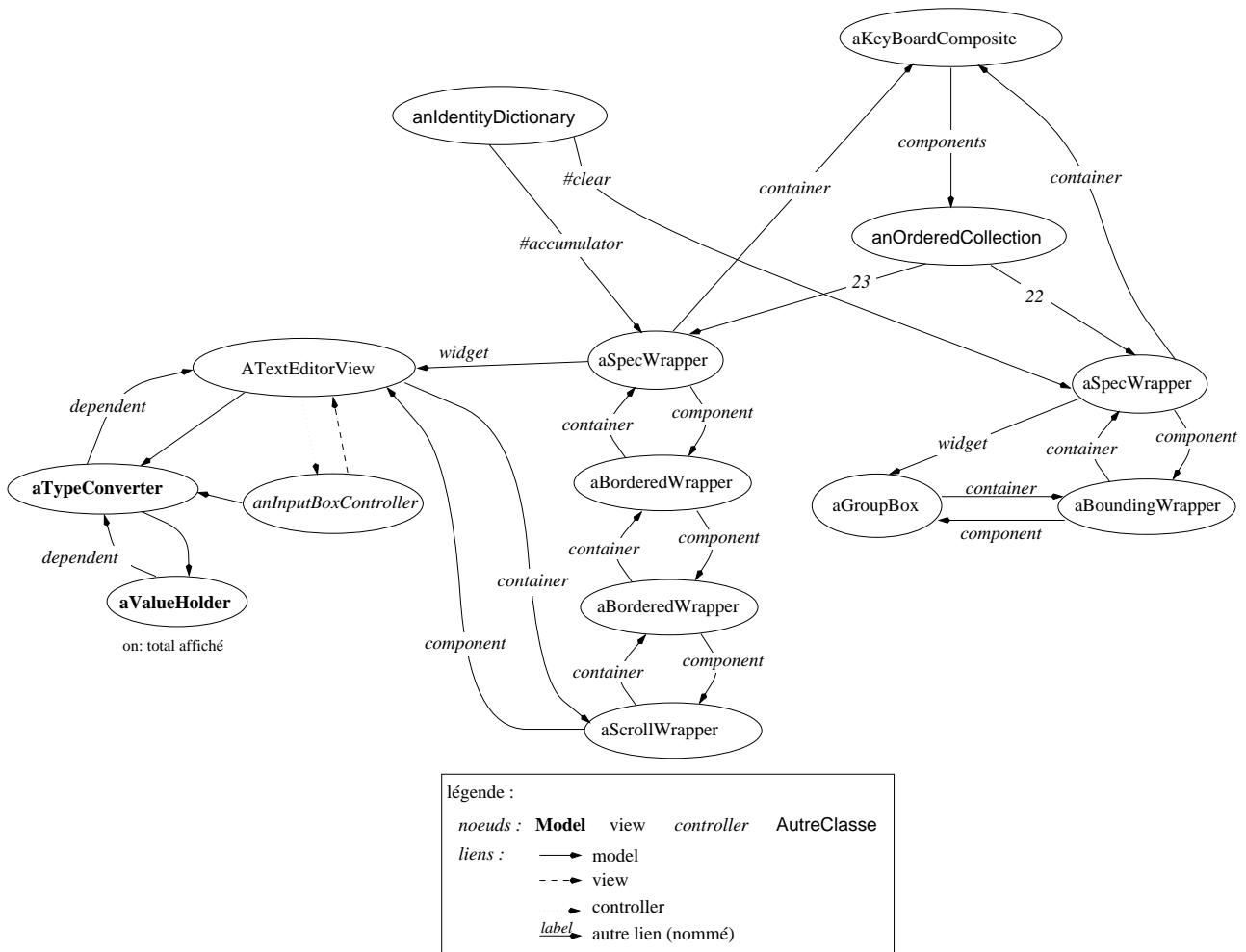


Figure 41 : Un exemple d'architecture MVC VisualWorks : la calculette 2/2

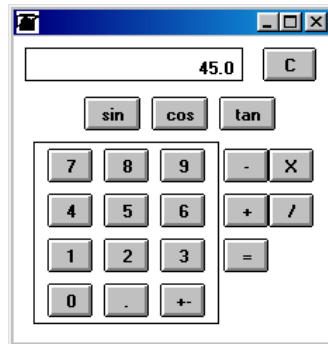


Figure 42 : Un exemple de MVC sous VisualWorks 2.5 : la calculette

Exercice Construire l'architecture des composants de l'exemple de la figure 42 en vous inspirant de celle des figures 40 et 41. Comparer.

Gestionnaire de compte

Le second exemple proposé dans l'environnement VisualWorks 2.5 est un gestionnaire compte financier.

Lancer l'application par : `T_CheckbookInterface open`

Exercice Construire l'architecture des composants de cet exemple.

Chapitre 6

Le développement d'applications avec VisualWorks


Ce chapitre est un exemple guidé de construction d'une application sous **VisualWorks**. Cette partie est fortement inspiré du tutorial de la section 2.4. L'exemple choisi est un **additionneur** d'entiers. Les éléments d'interface utilisés sont les zones de texte, les boutons et les menus.

Nous détaillons les différentes étapes de la construction de l'application : de l'interface au cœur de l'application, le modèle. En fait, pour une application réelle, on commence par une analyse du problème, puis une conception à objets. Ces deux étapes préliminaires mettent en évidence et structurent les objets de l'application.

L'application est un additionneur simple entre deux opérandes. Trois actions sont possibles : **additionner**, **soustraire**, **remettre à zéro**. L'opération **additionner** met la somme des deux opérandes dans le résultat. L'opération **soustraire** met la différence des deux opérandes dans le résultat. L'opération **remettre à zéro** met les deux opérandes et le résultat à zéro. Le résultat de l'opération est affiché après chaque opération. .

Pour contruire l'application, procéder dans l'ordre des étapes suivantes.

1 Dessin de l'interface

Nous commençons l'application par le dessin de l'interface graphique. Pour ouvrir une fenêtre sur un canevas d'interface, sélectionner la fonction **New Canvas** menu **Tools** de la fenêtre **Launcher** ou cliquer sur le bouton **New Canvas** . Trois fenêtres s'ouvrent : la palette, la boîte à outils et le canevas de l'interface.

La fenêtre supérieure est la boîte à outils (figure 43). Les fonctions s'appliquent aux éléments sélectionnés dans le canevas et par défaut au canevas lui-même si aucun élément n'est sélectionné. Les boutons à icônes représentent des raccourcis du menu **Arrange** : alignement, distribution et égalisation des éléments sélectionnés dans le canevas (de l'interface). L'égalisation se fait sur la hauteur ou la largeur des éléments. Le menu **Edit** définit des fonctions d'édition classiques (couper, copier, coller, valider, annuler, nouvelle fenêtre sur le canevas). Le menu **Tools** invoque des outils supplémentaires : palette, éditeur d'images bitmap, éditeur de menus (fixes ou déroulants), réutilisation d'autres composants). Le menu **Layout** définit les tailles des éléments sélectionnés dans le canevas ou de la fenêtre canevas elle-même : fixe ou variable. Le menu **Grid** définit une aimantation pour placer les éléments dans le canevas. Le menu **Look** définit la forme des fenêtres de l'interface (**Windows**, **OSF/Motif**, **MacIntosh**, etc.). Le menu **Special** contient des fonctions supplémentaires non-activables par défaut.

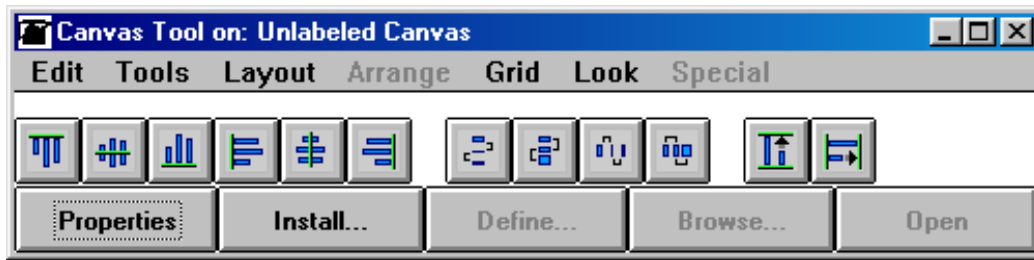


Figure 43 : Les outils de construction d'interface

La fenêtre gauche est la palette. Elle contient des éléments d'interface comme le montre la figure 44.

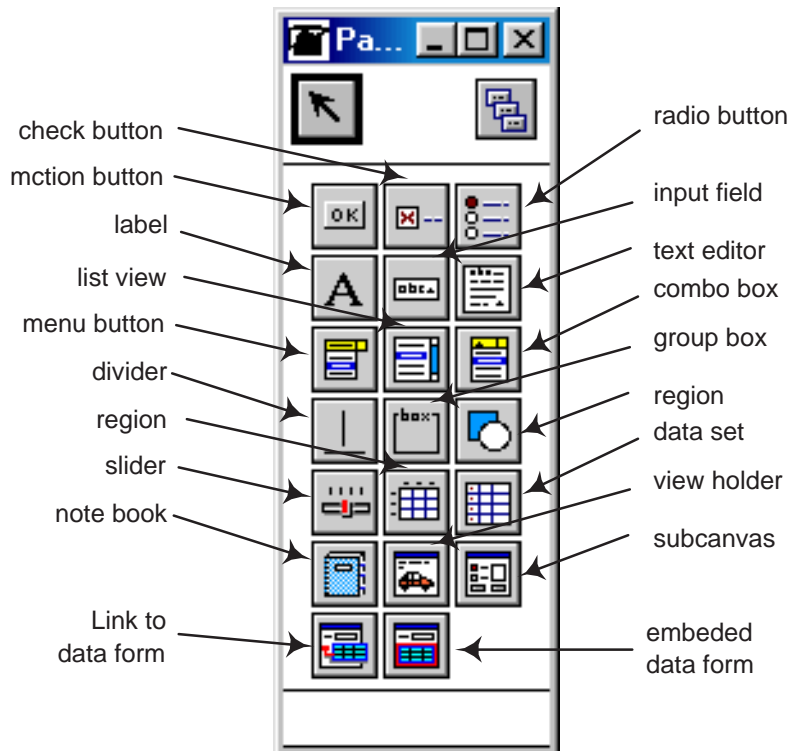


Figure 44 : Les outils de construction d'interface

La fenêtre centrale est le cannevas de l'interface, sur lequel nous allons travailler dans la figure 45.

Pour placer les éléments, procéder comme suit :

1. Choisir/Placer/Dimensionner trois *widgets* de type `InputField`.
2. Choisir/Placer/Dimensionner un *widget* délimiteur horizontal (`divider`).
3. Choisir/Placer/Dimensionner trois *widgets* de type `ActionButton`.
4. Aligner les éléments :
 - (a) sélectionner les champs de texte et le délimiteur, soit en cliquant successivement sur chaque élément avec le bouton gauche de la souris, soit en traçant avec le bouton gauche enfoncé une zone qui les contient. Ensuite, sélectionner la fonction d'alignement vertical :
 - par le bouton d'alignement vertical,
 - par la fonction `Align...` du menu `Arrange` de la fenêtre .
 - par la fonction `arrange>align...` du menu déroulant de la fenêtre cannevas accessible par le bouton central.

Dans les deux derniers cas, une fenêtre de dialogue s'ouvre, qui permet de choisir l'alignement des zones sélectionnées.

(b) Faire de même avec les boutons.

Nous pouvons aussi utiliser la grille et le couper/coller. En choisissant le look *Windows*, nous obtenons le canevas de la figure 45.

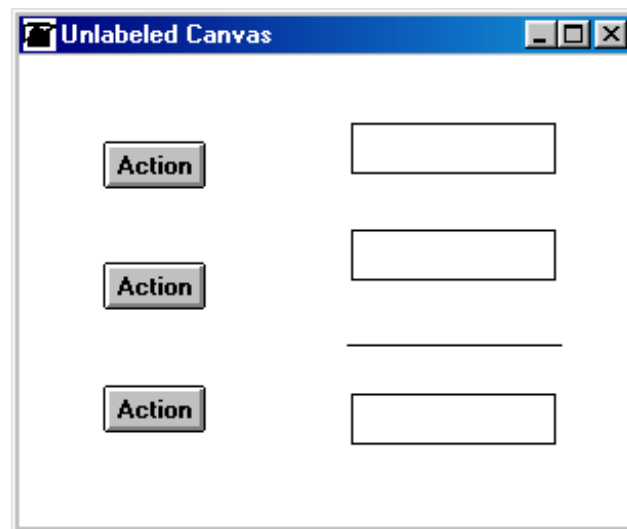


Figure 45 : *Exemple guidé : le canevas de l'additionneur*

5. Une fois les *widgets* placés, nous donnons les propriétés des éléments. Par exemple, le titre du canevas, les noms des boutons, etc. Nous sélectionnons chaque élément avec le bouton gauche. Puis nous utilisons le bouton **properties** ou le menu déroulant du bouton central pour ouvrir un dialogue sur les propriétés de l'élément. Noter que la fenêtre **Properties** est mise-à-jour en fonction de l'élément sélectionné. Il est donc inutile de fermer cette fenêtre pour passer aux propriétés d'un autre élément, il suffit de sélectionner cet autre élément dans le canevas en cours. Par exemple, sélectionnons le bouton d'action en haut à gauche. Ses propriétés sont positionnées comme le montre la fenêtre de dialogue de la figure 46.

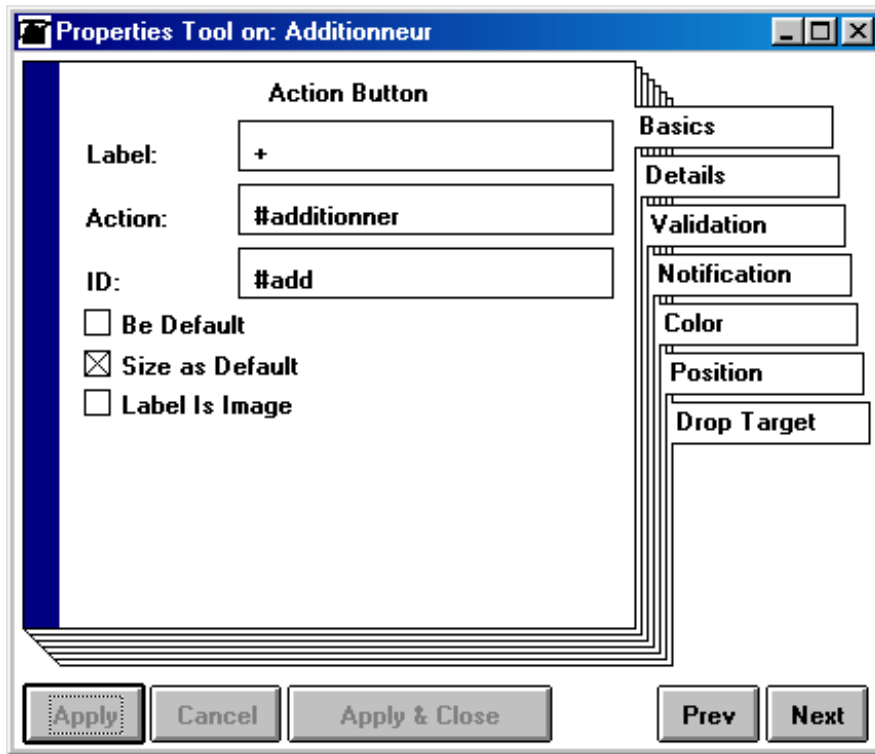


Figure 46 : Exemple guidé : les propriétés du bouton +

Les propriétés apparaissent sous forme d'une liste à onglets (NoteBook). Dans l'onglet **Basics**, les propriétés suivantes sont placées. Le label est '+'. L'action à exécuter en cas d'appui sur le bouton est la méthode '#additionner'. L'identifiant du bouton dans le canevas est 'add'. Valider par le bouton **Apply** avant de passer à l'onglet suivant. Dans l'onglet **Details**, choisir la taille actuelle comme taille par défaut, un bord pour le bouton, et la possibilité de passer à un autre élément graphique par le caractère tabulation (Can Tab). Faire de même avec les autres boutons selon les spécifications suivantes.

Elément	Label	Action	ID
ActionButton	+	#additionner	#add
ActionButton	-	#soustraire	#sous
ActionButton	Raz	#raz	#raz

Sélectionner maintenant la zone de saisie du haut (InputField).

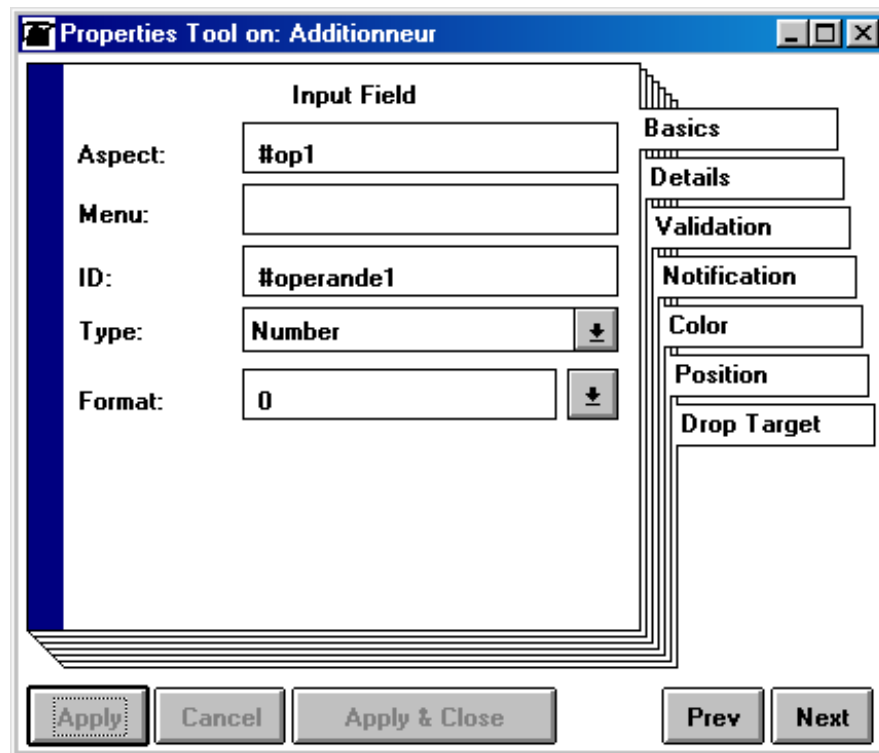


Figure 47 : Exemple guidé : les propriétés de l'opérande 1

Dans l'onglet **Basics**, les propriétés suivantes sont placées. L'aspect est le nom de la méthode d'accès du modèle qui délivre la valeur à afficher (*op1*). L'identifiant du champ dans le canevas est '*operande1*'. Il n'y a pas de menu associé au texte de ce champ. Le type du champ est **Number** affiché sur un format d'entiers. Valider par le bouton **Apply** avant de passer à l'onglet suivant. Dans l'onglet **Details**, choisir un alignement à droite, un bord pour le bouton, et la possibilité de navigation (**Can Tab**). Faire de même avec les autres boutons selon les spécifications suivantes avec la particularité de ne pas pouvoir modifier le résultat (**ReadOnly**) ni naviguer.

Elément	Aspect	ID	Font	Align	Type
InputField	#op1	#operande1	Default	Right	Number
InputField	#op2	#operande1	Default	Right	Number
InputField	#res	#resultat	Default	Right Can Tab OFF	Number ReadOnly ON

Si aucun élément n'est sélectionné, les propriétés manipulées sont celles de la fenêtre elle-même (le canevas). Nous pouvons alors changer son label et introduire une barre de menu appelée **menuBar** dans l'onglet **Basics** (figure 48).

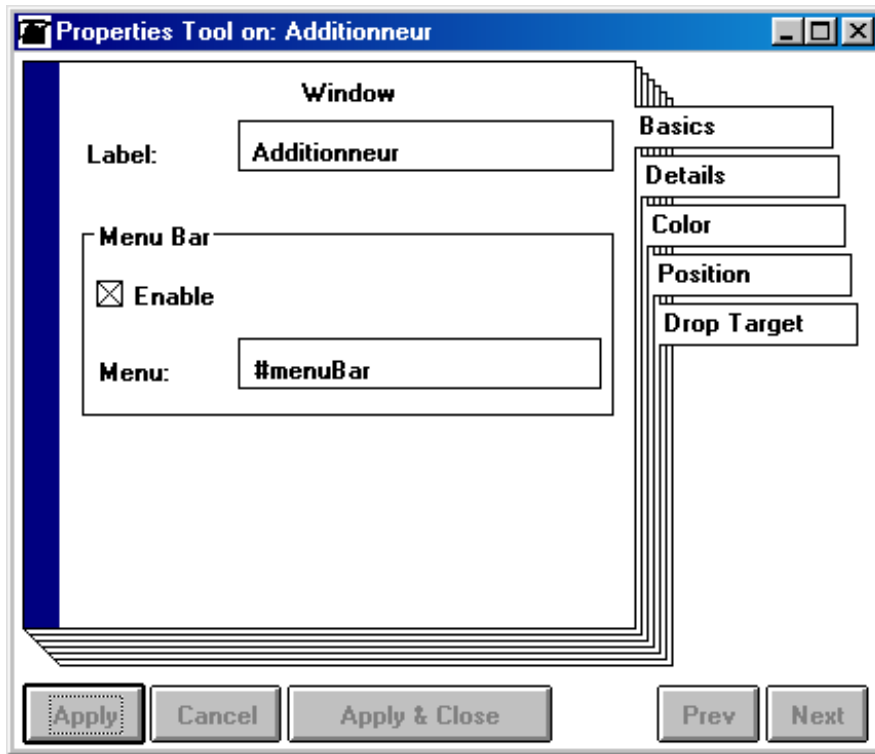


Figure 48 : Exemple guidé : les propriétés de la fenêtre

6. Nous installons ensuite le canevas dans la classe `AdditionneurInterface` en utilisant le bouton `install` ou le menu déroulant du bouton central pour ouvrir un dialogue sur les propriétés de l'application.

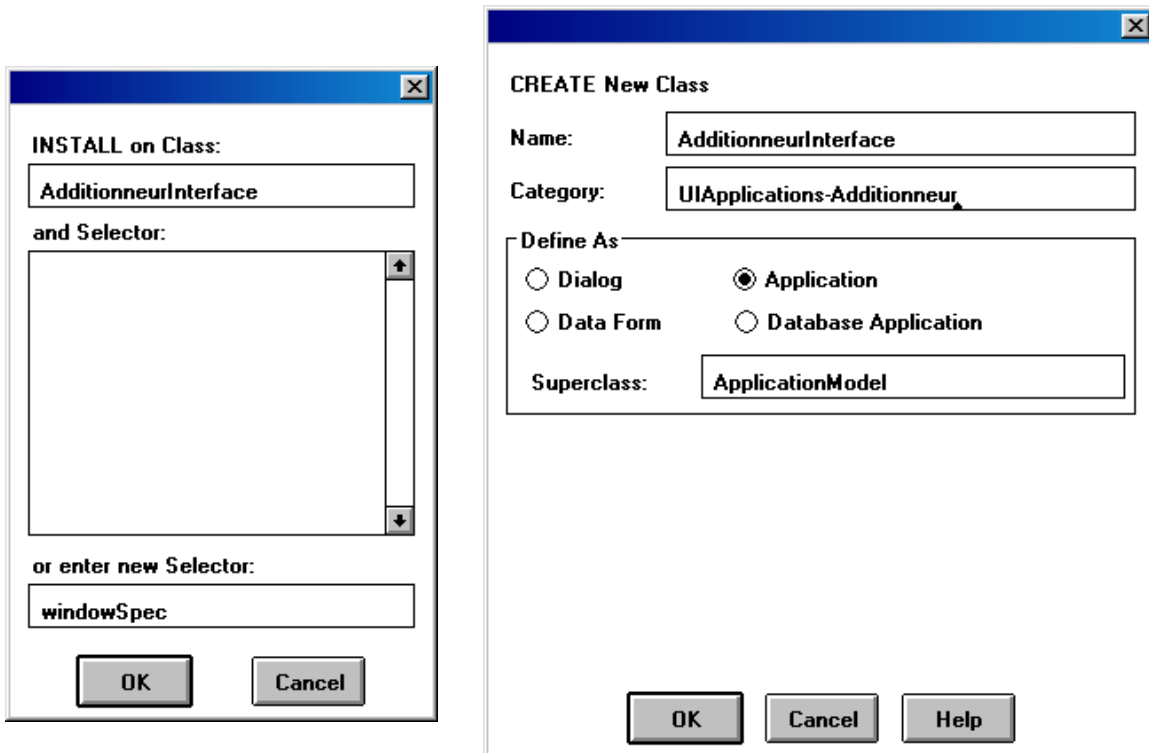


Figure 49 : Exemple guidé : les fenêtres d'installation

La méthode par défaut de spécification d'une interface est `windowSpec`. Procéder ensuite exactement comme suit. Dans la zone `Install on Class`, taper `AdditionneurInterface`

puis sur la touche **Entrée** pour ouvrir une fenêtre de création de l'application. La fenêtre est celle d'une application, on précise la catégorie de rangement dans le flâneur de classes. Rentrer les caractéristiques comme dans la (figure 49).

A partir de ce moment, l'interface est enregistrée comme une classe *Smalltalk*, dont la méthode de classe `windowSpec` définit entièrement la structure de la fenêtre principale de l'interface (exemple : voir figure 5.1). Vérifier dans un flâneur de classe que la classe `AdditionneurInterface` a bien été créée. Si le flâneur était déjà ouvert, il faut le mettre à jour par la fonction `update` du menu central de la liste des catégories.

7. La dernière chose à faire pour la fenêtre principale est de configurer la barre de menu. Pour cela, nous sélectionnons l'éditeur de menus par la fonction `Tools` de la boîte à outils ou la fonction `Tools` de la fenêtre `Launcher` ou enfin le menu déroulant du bouton central de la fenêtre du cannevas. Le menu apparaît comme une zone de texte dans laquelle on écrit les menus et sous menus. Le menu est construit comme un arbre, avec des actions vides au départ comme le montre la figure 50 en haut. Le séparateur est le caractère tabulation. En cliquant sur le bouton `Build`, un code est généré pour la barre de menus et un affichage en test apparaît sur la fenêtre de l'éditeur (figure 50 en bas).

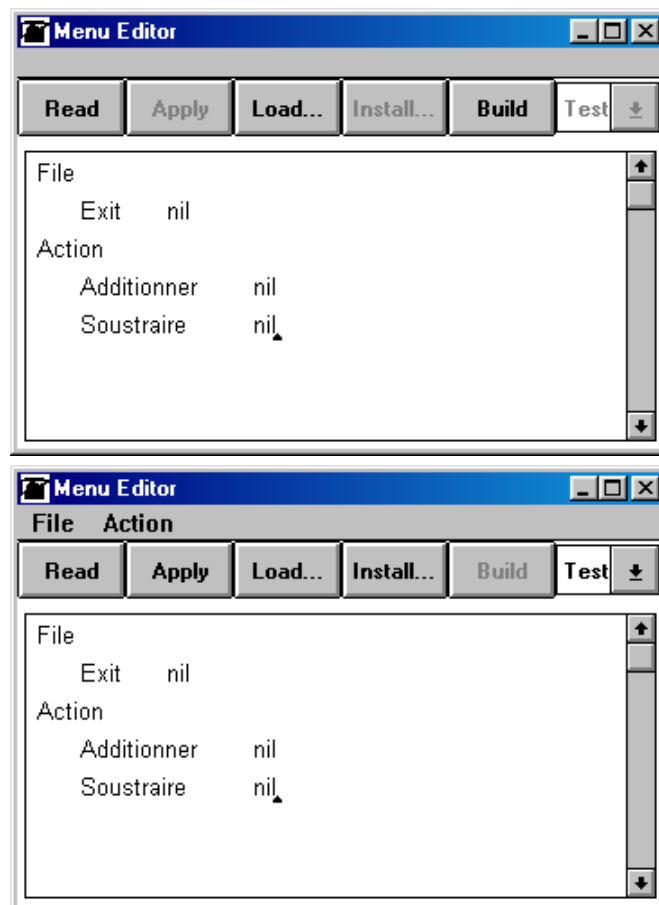


Figure 50 : Exemple guidé : l'éditeur de menus

Cliquer ensuite sur le bouton `Apply` pour installer le menu comme spécification de l'application additionneur. Une fenêtre similaire à celle de l'installation de l'interface apparaît. Taper `AdditionneurInterface` pour la classe et `menuBar` pour la méthode. Valider par `OK` puis fermer l'éditeur de menu par le menu ou le bouton de la fenêtre `windows` ou par le menu `close` du bouton droit dans la zone de texte.

8. Créer ensuite une fenêtre de dialogue `Validation` pour valider la remise à zéro. Refaites comme précédemment : création de cannevas.... Utilisez un label et deux boutons. Le

bouton `Oui` correspond à l'action `nil`, c'est le bouton par défaut. Le bouton `Non` correspond à l'action `nil`. Les véritables actions seront données plus tard. Il n'y a pas de menus pour la fenêtre. Installer ensuite le dialogue dans l'application `AdditionneurInterface` avec le sélecteur `dialogSpec`.

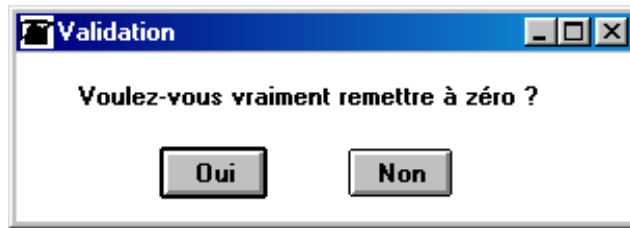


Figure 51 : Exemple guidé : le dialogue de validation

2 Création des classes du modèle

Nous allons maintenant procéder à la création du modèle (le domaine du problème, les classes métiers...). La création des classes du modèle est réalisée manuellement dans un navigateur de classe ou système.

- Nous commençons par créer une classe `Additionneur` contenant les informations du modèle en séparant ainsi application et domaine. Cette classe est placée dans la même catégorie que l'interface. Sa déclaration respecte les noms donnés dans l'interface.

```
Object subclass: #Additionneur
  instanceVariableNames: 'oper1 oper2 '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'UIApplications-Additionneur'
```

- Ajouter les commentaires nécessaires à la compréhension des opérandes.
- Ajouter les méthodes d'accès aux opérandes dans le protocole `accessing`.

```
oper1
  ^oper1!

op1: aNumber
  oper1 := aNumber !
```

```
oper2
  ^oper2!
```

```
oper2: aNumber
  oper2 := aNumber ! !
```

- Ajouter les méthodes de calcul dans le protocole `action`.

```
!Additionneur methodsFor: 'action'!
```

```
add
  ^oper1 + oper2!
```

```
sous
  ^oper1 - oper2! !
```

3 Liaison entre le modèle et l'interface

3.1 Liaison des données

Lier le code Smalltalk aux composants de l'interface.

- Une manière pratique de le faire dans le cas présent est de placer une variable d'instance `additionneur` dans la classe `AdditionneurInterface`.


```
ApplicationModel subclass: #AdditionneurInterface
  instanceVariableNames: 'additionneur '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'UIApplications-Additionneur'!
AdditionneurInterface comment:
'Cette classe implante l''interface d''un additionneur de nombres.
Les variables d''instance sont :
  additionneur : Additionneur'!
```

- Ajouter ensuite la méthode d'instance `initialize` dans le protocole `initialize`.
`!AdditionneurInterface methodsFor: 'initialize'!`

```
initialize
  additionneur := Additionneur new! !
```

3.2 Liaison du code

Nous donnons des propriétés des aspects et des actions pour lier code et données. Une propriété d'aspect définit une variable d'instance dont la valeur est associée à un *widget* de donnée (opérandes et résultat). Une propriété d'action définit une méthode d'instance invoquée à la sélection du *widget* (additionner, soustraire, remettre à zéro). Ces propriétés sont définies comme suit.

- Ouvrir le navigateur de ressources par le bouton `Resource Finder`  (ou le menu `Browse>Resources`) de la fenêtre `VisualWorks`. Puis cliquer sur le bouton `Edit` pour ouvrir le cannevas.

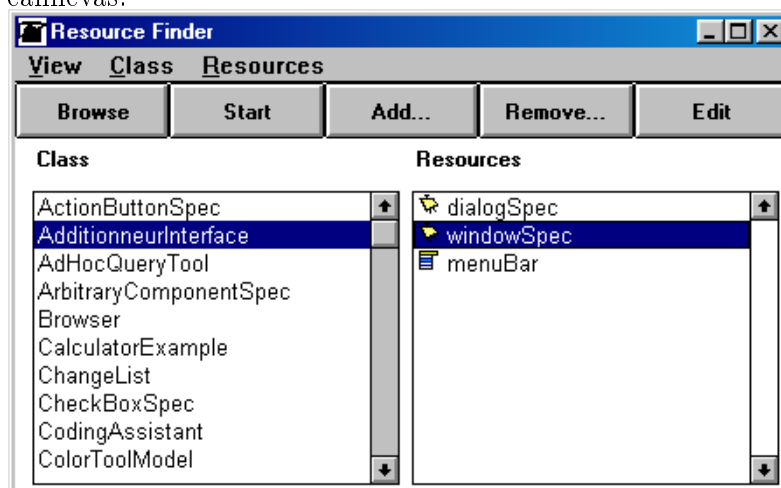


Figure 52 : Exemple guidé : l'installation des ressources

- Sélectionner le champ de l'opérande 1 puis affichez ses propriétés. L'aspect doit être `#op1`. Sélectionner ensuite le menu `method>define` pour définir les accesseurs correspondant dans le protocole `aspect` de la classe `AdditionneurInterface`. Faire de même avec `#op2` et `#res`.

- Sélectionner le bouton + puis affichez ses propriétés. L'action doit être #additionner. Sélectionner ensuite le menu `method>define` pour définir automatiquement les accesseurs correspondant dans le protocole `action` de la classe `AdditionneurInterface`. Faire de même avec #soustraire. Vérifier les ajouts avec un navigateur de classe. Modifier ensuite le code de ces deux actions avec le navigateur.

additionner

"This method was generated by UIDefiner and modified after "

```
additionneur oper1: op1 value.
additionneur oper2: op2 value.
res value: additionneur add!
```

soustraire

"This method was generated by UIDefiner and modified after "

```
additionneur oper1: op1 value.
additionneur oper2: op2 value.
res value: additionneur sous! !
```

- Pour le bouton #raz, nous devons d'abord associer les boutons Oui et Non de la fenêtre de dialogue. Editer la spécification `dialogSpec` dans la fenêtre `Resource Finder`. Commencer par donner une taille fixe à la fenêtre en positionnant les propriétés de la fenêtre par le menu `Layout>Window`. Puis lier respectivement le bouton Oui à la méthode prédéfinie `accept` et le bouton Non à la méthode prédéfinie `cancel`. Ces deux méthodes rendent vrai et faux. Réinstaller alors la spécification par le menu `install` et fermer le canevas. Modifier le code de la méthode `raz` de la classe `AdditionneurInterface` comme suit.

raz

"This method was generated by UIDefiner and modified after "

```
| accepte |
accepte ← self openDialogInterface: #dialogSpec.
accepte
    ifTrue:
        [op1 value: 0.
         op2 value: 0.
         res value: 0]!
```

- Compléter la barre de menu en associant une action à chaque option. Rappeler l'éditeur de menus pour l'application `AdditionneurInterface`.

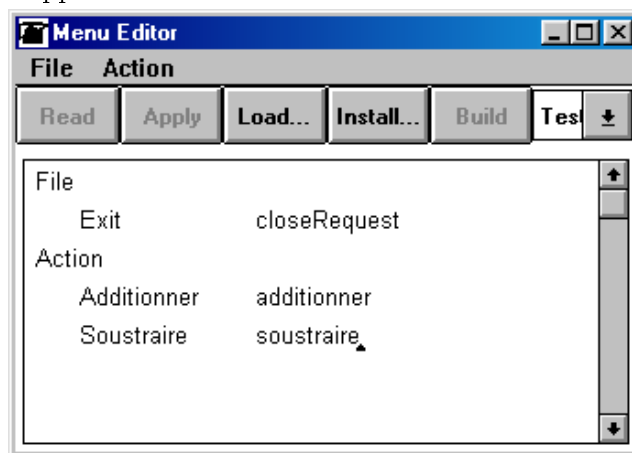


Figure 53 : Exemple guidé : l'éditeur de menus(suite)

Editer le menu `menuBar` selon l'exemple de la figure 53. Construire le menu par le bouton `Build` puis l'installer par `Install`. Quitter l'éditeur de menus.

- Réinstaller alors la spécification `windowSpec` par le menu `install` et fermer le cannevas. Les spécifications `windowSpec`, `dialogSpec` et `menuBar` sont des méthodes de classe de la classe `AdditionneurInterface`.

4 Vérification et compléments

Compléter les éléments si besoin, par exemple en agrandissant les zones des champs de texte, en modifiant le dimensionnement des fenêtres.

5 Exécution et tests

Tester l'application en sélectionnant le bouton `Start` de l'outil `Resource Finder` pour l'application `AdditionneurInterface` (cf figure 52). Vous pouvez aussi évaluer le message `AdditionneurInterface open` dans un espace de travail `Workspace`.

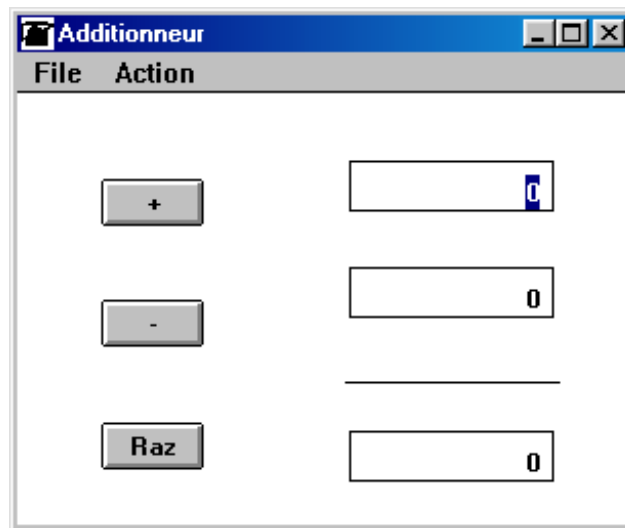


Figure 54 : *Exemple guidé : le cannevas achevé de l'additionneur*

6 Bilan

La démarche utilisée pour cet exemple sert à illustrer l'utilisation du générateur d'interface. Elle ne suffit pas pour développer une application plus complexe.

Chapitre 7

Conclusion

Nous avons développé dans ce cours les éléments essentiels de la programmation à objets avec le langage Smalltalk dans l'environnement **VisualWorks**.

Smalltalk est un environnement complet de programmation à objets. Le modèle objet est pur et très représentatif de la programmation avec des objets et des classes. Le langage est typé dynamiquement, ce qui facilite le prototypage et la mise au point. À l'inverse, la méthode de programmation doit être rigoureuse pour que le code produit soit de qualité (réutilisable, adaptable, fiable...).

Le langage est basé sur un nombre minimal de concepts. La connaissance du langage n'est donc pas la connaissance de ses structures mais celle des classes existantes. En ce sens, comme pour la plupart des langages à objets, la connaissance de l'environnement de développement et des bibliothèques de classes est fondamentale pour le développement d'applications.

L'environnement de développement **VisualWorks** pour Smalltalk-80 répond à cette attente. Avec environ 2000 classes prédéfinies et 600000 définitions de méthodes, l'environnement est complet. Voici le code à évaluer pour avoir ce résultat.

```
| nbMeth nbClasses |
  nbClasses ← Object allSubclasses size.
  nbMeth ← 0.
  Object allSubclassesDo:
    [:s | nbMeth ← nbMeth + s allSelectors size].
  Array with: nbClasses with: nbMeth
==> #(1903 603178)
```

Toutes les classes sont accessibles par les outils de développement. Cette "aide en ligne" favorise l'apprentissage du langage et la réutilisation. Les classes sont adaptables aux besoins propres des programmeurs.

VisualWorks propose aussi grand nombre d'outils pour le développement et en particulier des générateurs d'interfaces ou de grammaires. Nous avons illustré dans ce cours la création d'une application en utilisant un générateur d'interface. Des outils pour la gestion de bases de données sont aussi fournis dans l'environnement. Enfin, on trouvera dans l'environnement les classes nécessaires à l'implantation de prototypes de langages à objets ou de systèmes d'exploitation. Par exemple, pour définir l'héritage multiple, on redéfinit le protocole d'erreur dans l'envoi de message.

Ce cours contient toujours des erreurs. L'auteur remercie d'avance les lecteurs pour toute suggestion visant à améliorer ce cours.

Bibliographie

- [ACR94] Pascal André, Dan Chiorean, and Jean-Claude Royer. The formal class model. In *Joint Modular Languages Conference*, pages 59–78, Ulm, Germany, 28-30 september 1994. GI, SIG and BCS.
- [And96] Pascal André. Introduction à Smalltalk-80. Polycopié de cours, DESS GI de Nantes, Décembre 1996. (75 pages).
- [And01] Pascal André. Introduction au génie logiciel à objets et à uml. Polycopié de cours, INP-HB, Janvier 2001. (170 pages).
- [AR98] Pascal André and Jean-Claude Royer. Modélisation par objets. Rapport de Recherche RR-179, IRIN, Octobre 1998. (54 pages).
- [BDG⁺88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. X3J13 Document 88-002R, June 1988.
- [BS96] Xavier Briffault and Gérard Sabah. *Smalltalk, Programmation orientée objet et développement d'applications*. Editions Eyrolles, 1996. ISBN 2-212-08914-7.
- [Car88] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76 :138–164, 1988.
- [Coi87] Pierre Cointe. Metaclasses Are First Classes : the ObjVlisp Model. In *ACM OOPSLA'87 proceedings*, pages 156–167. ACM, October 1987. October 4-8.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley Publishing Company, 1995. ISBN 0-201-63361-2.
- [Gol83] Adele Goldberg. *Smalltalk-80 : The Language*. Addison-Wesley, 1983.
- [Gro03] Object Management Group. The OMG Unified Modeling Language Specification, version 1.5. Technical report, Object Management Group, available at <http://www.omg.org/cgi-bin/doc?formal/03-03-01>, June 2003.
- [HH95] Trevor Hopkins and Bernard Horan. *Smalltalk-80, An introduction to Applications Development*. Prentice Hall International Editions, 1995.
- [How95] Timothy G. Howard. *The Smalltalk Developer's Guide to VisualWorks*. Cambridge University Press, New York, NY, USA, 1995. ISBN 1-884842-11-9.
- [Lal94] Wilf R. Lalonde. *Discovering Smalltalk*. Object-Oriented Software Series. Addison Wesley, 1994.
- [Mey89] Bertrand Meyer. The New Culture of Software Development : Reflections on the Practice of OOD. In Jean Bezivin and Bertrand Meyer, editors, *Proceedings of TOOLS'89*, Paris, November 1989.
- [Mey97] Bertrand Meyer. *Object-oriented Software Construction*. International Series in Computer Science. Prentice Hall, 2 edition, 1997. ISBN 0-13-629155-4.
- [MNC⁺90] Gérald Masini, Amedeo Napoli, D. Colnet, D. Léonard, and Karl Tombre. *Les langages à objets*. Collection IIA. InterEditions, 1990.

- [Ner90] Jean-Marc Nerson. Case Studies in Object-Oriented Analysis. In Jean Bezivin and Bertrand Meyer, editors, *TOOLS'90*, June 1990. Tutorial.
- [RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object-Oriented Series. Addison-Wesley, 1999. ISBN 0-201-30998-X.
- [Roy94] Jean-Claude Royer. La programmation à objets. Polycopié de cours, Université de Nantes, 24 septembre 1994. (121 pages).
- [Sha97] Alec Sharp. *Smalltalk by Example*. Mc Graw Hill, 1997. The Developers Guide to VisualWorks, ISBN 0-07-913036-4.
- [Weg90] Peter Wegner. Concepts and paradigms of oop. *OOP Messenger*, 1(1), August 1990. ACM.

Annexe A

Exercices

Ce chapitre a pour but d'illustrer les différents points du cours par des exemples. Les exercices proposés le sont actuellement à titre expérimental.

1 Langage

Dans cette section l'objectif est d'appréhender les structures de base de Smalltalk.

Exercice 1.1

Ecrire une fonction qui calcule la suite de Fibonacci sur les entiers. Tester cette fonction.

Exercice 1.2

Définir des ensembles ayant un nombre pair d'objets. Définir des ensembles de couples d'objets.

Exercice 1.3

Définir des pixels et des points en trois dimensions.

Exercice 1.4

Ecrire une méthode `opposé` qui rend l'opposé d'un nombre. Par exemple, l'opposé de `-12` est `+12`.

Exercice 1.5

Soit la méthode booléenne `diviseurDe:`, qui appliquée à un entier renvoie `true` si cet entier divise le paramètre de la méthode.

a) Ecrire le code de cette méthode.

b) Comment peut-on implanter cette méthode sans modifier la classe `Integer` ?

c) Ecrire une méthode qui génère, sous forme de bloc la fonction `diviseurDe:` à partir d'un entier donné en paramètre.

C'est une fonction de première classe car une fonction rend en résultat une autre fonction.

Exercice 1.6

a) Ecrire une méthode qui calcule la somme de deux vecteurs et une méthode qui calcule la produit scalaire d'un vecteur par un entier.

b) Coder ces méthodes en Smalltalk, sans modifier les classes de base du système.

2 Classes

Dans cette section l'objectif est d'appréhender les classes essentielles de Smalltalk.

Exercice 2.1

Etudier la hiérarchie des sous-classes de la classe `Magnitude` : la modélisation des nombres et des éléments comparables (relation d'ordre).

Exercice 2.2

Etudier la hiérarchie des sous-classes de la classe `Collection`, les différentes représentations des groupes d'objets.

Exercice 2.3

Etudier les classes de la catégorie `Kernel-Objects`, à savoir le comportement de base des objets et la modélisation des booléens.

Exercice 2.4

Etudier les classes de la catégorie `Kernel-Classes`, à savoir l'organisation des classes et les méta-classes.

Exercice 2.5

Etudier les classes de la catégorie `Kernel-Methods`, à savoir l'organisation du code et la compilation en Smalltalk. Dans cette catégorie, on trouve aussi les blocs (environnements lexicaux), essentiels à la manipulation dynamique du code, les envois de message et les contextes d'exécution.

Exercice 2.6

Etudier les classes de la catégorie `Kernel-Process`, à savoir les éléments de base d'un système d'exploitation : processus, sémaphores, ordonnanceur.

Exercice 2.7

Etudier les classes de la catégorie `OS-Unix`, à savoir l'interface avec un système de fichier Unix.

3 Programmation à objets

Exercice 3.1

Programmer le jeu de Tetris avec `VisualWorks`.

Exercice 3.2

On souhaite écrire un programme qui simule une partie de tennis entre deux joueurs, selon les règles classiques du tennis (inspiré librement de [Ner90]). Un match se joue en 3 sets gagnants (soit au maximum 3 ou 5 jeux).

Un joueur gagne un set s'il a au moins 6 jeux gagnés et deux jeux d'écart avec son adversaire. Si le score est de 6 jeux à 5 alors plusieurs cas sont possibles : soit le premier joueur gagne le jeu et il remporte le set 7-5, soit le second joueur gagne le jeu et un tie-break est joué, le gagnant du tie-break remporte le set par le score de 7 à 6.

Chaque jeu se joue en 4 points maximum pour le gagnant et deux points d'écart avec son adversaire. Les points sont notés 0, 15, 30, 40, jeu. En cas d'égalité 40-40, les points sont notés : égalité, avantage au serveur ou avantage au receveur.

Le tie-break se joue en 7 points minimum pour le gagnant et deux points d'écart avec son adversaire (par exemple, 7-4, 9-7...).

C'est le même joueur qui sert pour tout un set, hormis le tie-break pour lequel, le serveur du set sert une fois puis les deux joueurs servent alternativement. Le joueur qui sert en premier est choisi au hasard. On ne tiendra pas compte des fautes ou des deux balles de service. Pour chaque balle engagée, le point sera simplement gagnant ou perdant.

On ne tient pas compte pour l'instant des impondérables : blessures des joueurs, intempéries, abandons.

a) Écrire un programme impératif qui modélise un match de tennis. On pourra utiliser des tableaux et une numérotation numérique des joueurs. On pourra aussi coder le codage des points pour faciliter le calcul des scores.

b) On souhaite pouvoir adapter le programme aux évolutions suivantes.

- 1. Dans un tournoi certains matchs se jouent en deux sets gagnants (début du tournoi, tournoi féminin...).*
- 2. On souhaite inclure ce programme dans le cadre d'un tournoi complet et conserver tous les scores jusqu'à la finale.*
- 3. On souhaite modifier le programme pour des matchs de doubles.*
- 4. On souhaite écrire un programme similaire pour un tournoi de tennis de table.*
- 5. Un joueur peut abandonner la partie.*

Commenter l'adaptabilité du programme aux modifications souhaitées.

c) Proposer une conception abstraite à objets. On proposera par exemple une modélisation avec la notation UML.

d) Reprendre la question b) avec la conception à objets.

e) Implanter la conception à objets en Smalltalk. Discuter des alternatives de codage.

4 Applications

Exercice 4.1

*Développer un agenda (nom, téléphone) en utilisant le générateur d'interface de *Visualworks*.*

Exercice 4.2

*Développer un browser d'objets géométriques, qui affiche les formes choisies, en utilisant le générateur d'interface de *Visualworks*.*

5 Un peu plus

Le lecteur trouvera dans le cours présenté dans la section 2.6 des exercices de travaux dirigés et de travaux pratiques illustrant les concepts du langage.

Dans le cadre d'un cours de Génie Logiciel avec UML [And01], le langage Smalltalk sert au prototypage des applications modélisées avec UML.

Annexe B

Compléments

1 La souris

Historiquement, les boutons de la souris ont des couleurs. Le bouton gauche est rouge `red button`. Le bouton central est jaune `yellow button`. Le bouton droit est bleu `blue button`.

2 Références autour de Smalltalk

Cette section propose quelques références bibliographiques de cours et d'ouvrages autour de la programmation à objets avec Smalltalk.

2.1 Smalltalk 4.1, un tutorial

Auteur : Antoine Beugnard

Résumé : Nous présentons dans ce document l'interface de Smalltalk-4.1, les principes de programmations, les outils de développement et les classes de bases. Notre démarche est incrémentale et sollicite le lecteur. Nous vous conseillons de lire ce document activement, en face d'un environnement Smalltalk. De nombreux exercices sont proposés qui permettent de mettre en pratique les concepts et les outils décrits. Un Index en fin de document permet de retrouver rapidement la plupart des concepts, classes, méthodes décrits.

Adresse : <http://antares.enst-bretagne.fr/Tutoriaux/smalltalk/s80part1/s80part1.html>

Commentaire : Document assez succinct, sur une version plus ancienne de Smalltalk-80. Lecture gênante à cause du format HTML.

2.2 Smalltalk Textbook (in English)

Auteur : AOKI Atsushi

Résumé : We have made AOKI Atsushi's Smalltalk Textbook available through the WWW. This online version contains cross references and links which will facilitate working through the book via a web-browser while running VisualWorks. Each chapter contains examples which can be cut from the web page and pasted into an active Visualworks window allowing you to experiment with the examples while you progress through the textbook. The Smalltalk Textbook was written by AOKI Atsushi and was translated into English by Kaoru Rin Hayashi and Brent N. Reeves. Other translations are under way!

The entire textbook in ascii text is available via SRA ftp server.

Many have asked about a postscript version, but the textbook contains so many examples which are meant to be cut-and-pasted into VisualWorks at run time that a postscript version would not really be that useful.

Email : smalltalker@srainc.com

This textbook is for VisualWorks 1.0.

1. VisualWorks 2.0 Users : Please refer to 'Patch of Textbooks' and 'Path of Appendices' for VisualWorks 2.0.

2. VisualWorks 2.5 Users : Please read (file-in) 'oldview.st' that is included in the released files, and then refer to 'Patch of Textbooks' and 'Patch of Appendices' for VisualWorks 2.5.

Adresse : <http://www.sra.co.jp/people/aoki/SmalltalkTextbook/>

Commentaire : Document contenant des exemples de programme sur différents concepts et différentes classes.. Lecture gênante à cause du format HTML.

2.3 Smalltalk on the net

Auteur : Jeff McAffer (Société OTI)

Résumé : The links below are but a sampling of the Smalltalk resources available on the web. I've attempted to review the content of all links and include those which will appeal to a broad range of people interested in Smalltalk. Pages whose sole purpose is advertising have been filtered. This set of links would not be possible without your input. I encourage you to contact me if you have related corrections, additions or comments. Thanks.

Adresse : <http://www.oti.com/jeffspg/smaltalk.htm>

jeff_mcaffer@oti.com

Commentaire : Bonne synthèse des références sur Smalltalk avec notamment les logiciels *freeware* ou *shareware* et des tutoriaux.

2.4 Introduction to VisualWorks : An Application Approach (in English)

Auteur : R. Andrew Painter and A. Joseph Turner Clemson University Department of Computer Science

Résumé : Introduction to VisualWorks : An Application Approach is free to copy and distribute as long as it is not used for profit and proper credit is given to the authors. The tutorial is available as a tar file that can be untarred to produce all of the html and gif files for the tutorial.

This tutorial is intended to introduce the basic features and tools of the VisualWorks^o Smalltalk environment. A basic knowledge of the Smalltalk language is assumed.

The tutorial was developed using VisualWorks 2.0 on a Sun/Solaris platform, and it is oriented toward the use of VisualWorks in the Department of Computer Science at Clemson University. However, the use of VisualWorks on other platforms is similar.

This tutorial was greatly influenced by ParcPlace Systems' VisualWorks Tutorial. ParcPlace Systems' VisualWorks Tutorial provides a more in-depth look at the concepts covered in this tutorial as well as an overview of the process of designing a VisualWorks Application.

Adresse : <http://www.cs.clemson.edu/~lab428/VW/VWCover.html>

Commentaire : Ce cours sur VisualWorks est destiné aux débutants. Il permet de s'initier assez simplement aux concepts et outils tant de Smalltalk que de *VisualWorks*. Des exemples concrétisent le discours.

2.5 Smalltalk : idéal pour le prototypage et la découverte de l'objet

Auteur : Rethore

Résumé : Smalltalk est un langage sans concession, il pratique le tout objet, en Smalltalk tout est objet et tout objet est instance d'une classe ce qui lui donne une cohérence forte. Il est le langage idéal pour l'apprentissage de la programmation objet, à la fois à cause de son homogénéité mais aussi parce qu'il met à la disposition du programmeur un très grand nombre de hiérarchies de classes et un environnement convivial, interactif qui permet un réel apprentissage de la réutilisation ce qui est le point fondamental. La version de Smalltalk que

nous vous proposons d'utiliser est Smalltalk Express de Parc Place. Cette version dispose d'un environnement complet qui s'impose de plus en plus comme un standard. Nous avons complété cet environnement en y ajoutant de nombreux outils.

Adresse : http://www.lifl.fr/HTPUB/rethore/public_html/smalltal.htm

Commentaire : Le site propose le téléchargement d'outils de de tuturiels.

2.6 Smalltalk : transparents de cours avec exercices

Auteur : Elisabeth Delozanne IUP MIME au Mans (France)

Résumé : Il s'agit d'un cours complet en français sur Smalltalk, qui date de de 1996. Des sujets et corrigés d'exercices sont proposés.

Adresse :

<http://groucho.univ-lemans.fr/~deloz/licence-12/smalltalk/cours1/index.htm>

Commentaire : On se perd un peu dans les fichiers mais le cours est complet. Le programmeur débutant et le programmeur confirmé trouveront des exercices sur mesure pour développer en Smalltalk.

2.7 Quelques adresses

Object-Oriented and Smalltalk Sites :

<http://ra.nilenet.com/~gaska/objects.html>

A Comparison of Object Oriented Development Methodologies :

<http://www.toa.com/pub/html/mcr.lhtml>

ParcPlace Systems, Inc :

<http://www.parcplace.com/>

Object Technology International Inc. :

<http://www.oti.com/>

Un cours à Paris 8 (France) :

<http://www.ai.univ-paris8.fr/~vi/small/Small1.htm>

2.8 Livres

Titre : Smalltalk, Programmation orientée objet et développement d'applications

Auteur : Xavier Briffault & Gérard Sabah

Référence : [BS96]

Commentaire : Un livre en français sur l'utilisation de Smalltalk et VisualWorks. Cet ouvrage est utile pour la maîtrise de l'environnement de VisualWorks.

Titre : Smalltalk, An Introduction to Application Development using VisualWorks

Auteur : Trevor Hopkins & Bernard Horan

Référence : [HH95]

Commentaire : Un livre d'introduction générale à Smalltalk et VisualWorks. Ce livre est destiné aux débutants en Smalltalk : les explications sont simples mais claires. Le lecteur reste un peu sur sa faim. Concernant VisualWorks, seul deux chapitres y font réellement allusion.

Titre : The Smalltalk Developer's Guide to VisualWorks

Auteur : Tim Howard

Référence : [How95]

Commentaire : Un ouvrage très complet sur *VisualWorks*. Il est destiné à un public connaisseur et expérimenté en Smalltalk. L'auteur dissèque les mécanismes de VisualWorks. Cela fait

beaucoup de choses à comprendre, mais le résultat en vaut la peine. Attention, les exemples du livre (muni d'une disquette) sont dans la version 2 de VisualWorks.

2.9 Actualités

Non exhaustif

- Pharo - le projet phare et libre <http://www.pharo-project.org/>
- Le groupe Smalltalk-fr <https://groups.google.com/group/smalltalk-fr?hl=fr>
- Le groupe wnc vwnc@cs.uiuc.edu
- La page de Stéphane Ducasse (livres) <http://stephane.ducasse.free.fr/FreeBooks.html>
-

3 VisualWorks 3.0/linux

Sous VisualWorks 3.0/linux :

```
[andre@www]$ mkdir VW3
[andre@www]$ cp /usr/local/opt/VW3/vwnc30/image/visualnc.im VW3/ex.im
[andre@www]$ ls -al VW3
total 3937
drwxr-xr-x  2 andre  engnt      1024 Nov 30 14:51 .
drwxr-xr-x 16 andre  engnt      1024 Nov 30 14:45 ..
-rw-r--r--  1 andre  engnt    4011820 Nov 30 14:51 ex.im
[andre@www]$ cd VW3/
[andre@www]$ /usr/local/opt/VW3/bin/linux86/visualnc ex.im &
```

Dans File>Settings, on positionne les sources, images et change Enregistrer sous 'ex'

```
[andre@www]$ ls -al
total 4080
drwxr-xr-x  2 andre  engnt      1024 Nov 30 15:05 .
drwxr-xr-x 16 andre  engnt      1024 Nov 30 14:45 ..
-rw-r--r--  1 andre  engnt       142 Nov 30 15:05 ex.cha
-rw-r--r--  1 andre  engnt    4156632 Nov 30 15:05 ex.im
-rw-r--r--  1 andre  engnt         0 Nov 30 15:04 visualnc.cha
```

DISTRIBUTION CONTENTS

```
readme.txt  (this document)
license.txt (VisualWorks Non-Commercial license)

vwnc30/    (VisualWorks Non-Commercial 3.0 release directory;
           although only .pcl parcel files are listed, there is also
           an accompanying .pst source file for each)
bin/      (empty directory for executable files)
doc/
  vwadg.pdf (VisualWorks Application Developer's Guide
            in Acrobat format)
  vwbbug.pdf (Business Graphics User's Guide in Acrobat format)
  vwig.pdf  (VisualWorks Non-Commercial Installation Guide
            in Acrobat format)
```

```
vwrn.pdf    (VisualWorks Non-Commercial Release Notes in
             Acrobat format)
SysDeps.pdf ("events" document in Acrobat format)
ChangeList.txt (change list features)
image/
  visualnc.im
  visualnc.sou
parcels/
  BOSS.pcl
  Database.pcl
  Headless.pcl
  Help.pcl
  ImageMaker.pcl
  Lens.pcl,
  MacExtras.pcl
  SysDeps.pcl
  UIPainter.pcl
removals/ (removal scripts for Image making)
  advbase.rm, db2dkit.rm, lensgen.rm, orcompat.rm,
  tablview.rm, advcompat.rm, db2ifc.rm, lenstool.rm,
  os2look.rm, uibase.rm, advex.rm, dbcompat.rm,
  maclook.rm, printing.rm, uitool.rm, advtool.rm,
  dsettool.rm, mtdflook.rm, progex.rm, winlook.rm,
  bgokex.rm, dsetview.rm, ntbktool.rm, progtool.rm,
  xtrnbase.rm, bgoktool.rm, exdi.rm, ntbkview.rm,
  sycompat.rm, xtrntool.rm, bgokview.rm, exditool.rm,
  oldview.rm, sydkit.rm, boss.rm, help.rm, oradkit.rm,
  syifc.rm, compiler.rm, lensapp.rm, oraifc.rm,
  sylens.rm, dand.rm, lensbase.rm, oralens.rm,
  tabltool.rm
online/
  cookbook.hlp
  examples/
    Adapt1.st, Adapt2.st, Adapt3.st, Adapt4.st, Adapt5.st,
    Adapt6.st, Amortize.st, Button.st, Color.st, Combo.st,
    ComboCon.st, Cursor.st, Cust1.st, Cust2.st, CustVw1,
    CustVw2.st, DB1.st, DB1add.st, DB1delet.st,
    Dataset1.st, Dataset2.st, Dataset3.st, Dataset4.st,
    Depend.st, Dialog.st, DragDrop.st, Editor1.st,
    Editor2.st, Employee.st, FldConn.st, FldFilt.st,
    FldMenu.st, FldSel.st, FldType.st, FldVal1.st,
    FldVal2.st, FldValid.st, Font1.st, Font2.st, Hide.st,
    Line.st, List1.st, List2.st, LoanCalc.st, Logo.st,
    MenuCmd.st, MenuEdit.st, MenuMod., MenuSwap.st,
    MenuVal.st, Move.st, Notebk1, Notebk2.st, Notebk3.st,
    Notebk4.st, Notebk5.st, Point., PrefCust.st, Size1.st,
    Size2.st, Size3.st, SkCon1.st, SkCon2.st, SkView1.st,
    SkView2.st, Sketch.st, Slider1.st, Slider2.st,
    Subcan1.st, Subcan2.st, Subcan3.st, Table1.st,
    Table2.st, Table3.st, TimeType.st
compat/
```

```

    OldLauncher.pcl (support for the (very) old menu-like launcher)
    UIMenuEditor.pcl (old menu editor)
goodies/
  readme.txt
  other/
    CoolImage.pcl (image [icon] editor)
    Gremlin.pcl
    HotDraw.pcl
    HotDrawAnimatedExamples.pcl
    HotDrawAnimationFramework.pcl
    HotDrawDrawingInspector.pcl
    HotDrawFramework.pcl
    HotDrawHotPaint.pcl
    HotDrawPERTChart.pcl
    HotDrawToolDevelopment.pcl
    ProgressWidget.pcl
    RefactoringBrowser.pcl
    Regex.pcl
    SmallWalker.pcl
  parc/
    BrowserChangeSetEditing.pcl
    ColorEditing.pcl (color syntax-driven editor)
    FlyByHelp.pcl
    FlyByHelpPreLoad.pcl
    GFST-Base.pcl
    GFST-Demo.pcl
    GFST-GraphLayout.pcl
    GFST-OrgChart.pcl
    GFST-VisualInspector.pcl
    gfstvw.hlp
    JuggleButtons.pcl
    ProgrammingHacks.pcl
    STAMP.pcl
    VRGoodies.pcl (tool goodies)
    VRGoodies.pdf (VR Goodies documentation in Acrobat format)
    VRLens.pcl (lens and tools extensions)
    VRPainter.pcl
    VRPainterPrereq.pcl
    VRPainterPrereq2.pcl
    VRUIPainter.pcl
    WorkspaceOrganizer.pcl
  resource/
    Aboutmsk.bmp, Arrow.bmp, Bezier.bmp, Btofrnt.bmp,
    Button.bmp, Calendar.bmp, Calmask.bmp, Clsfig.bmp,
    Comps.bmp, Dwngmask.bmp, Dwngtool.bmp,
    Editfld.bmp, Edittext.bmp, Edittool.bmp,
    Ellipse.bmp, Etcats.bmp, Gofigure.Ico, Graphs.bmp,
    Grphmask.bmp, Help.bmp, Helpmask.bmp, Image.bmp,
    Inspect.bmp, Inspmask.bmp, Lendmask.bmp, Line.bmp,
    Lineend.bmp, Lines.bmp, List.bmp, Netmask.bmp,
    Network.bmp, Orgchart.bmp, Orgmask.bmp,

```


Orthoarw.bmp, Orthopth.bmp, Para.bmp, Paths.bmp,
Polyline.bmp, Rect.bmp, Rndrect.bmp, Scroller.bmp,
Select.bmp, Sndtobck.bmp, Spline.bmp, Text.bmp,
Textfld.bmp, Zorder.bmp

Other add-ons such as the Advanced Tools or the DLL and C Connect
should be installed in the vwnc30 directory.

02/02/98 py

Annexe C

Solution des exercices

1 Conception et programmation d'une gestion de match de tennis

Cette section est à la fois une introduction à la programmation à objets et à la programmation avec Smalltalk. Le cas d'étude est la gestion d'un match de tennis. Cet exemple est inspiré de de [Ner90].

Nous partons d'un algorithme de programmation structurée pour aboutir à une programme à objets. Nous mettons en évidence un certain nombre de problèmes d'évolution du programme. Nous proposons ensuite une modélisation et une programmation à objet.

1.1 Enoncé informel

Nous reprenons l'énoncé de l'exercice 3.2.

On souhaite écrire un programme qui simule une partie de tennis entre deux joueurs, selon les règles classiques du tennis (inspiré librement de [Ner90]). Un match se joue en 3 sets gagnants (soit au maximum 3 ou 5 jeux).

Un joueur gagne un set s'il a au moins 6 jeux gagnés et deux jeux d'écart avec son adversaire. Si le score est de 6 jeux à 5 alors plusieurs cas sont possibles : soit le premier joueur gagne le jeu et il remporte le set 7-5, soit le second joueur gagne le jeu et un *tie-break* est joué, le gagnant du *tie-break* remporte le set par le score de 7 à 6.

Chaque jeu se joue en 4 points maximum pour le gagnant et deux points d'écart avec son adversaire. les points sont notés 0, 15, 30, 40, jeu. En cas d'égalité 40-40, les points sont notés : égalité, avantage au serveur ou avantage au receveur.

Le *tie-break* se joue en 7 points minimum pour le gagnant et deux points d'écart avec son adversaire (par exemple, 7-4, 9-7...).

C'est le même joueur qui sert pour tout un set, hormis le *tie-break* pour lequel, le serveur du set sert une fois puis les deux joueurs servent alternativement. Le joueur qui sert en premier est choisi au hasard. On ne tiendra pas compte des fautes ou des deux balles de service. Pour chaque balle engagée, le point sera simplement gagnant ou perdant.

On ne tient pas compte pour l'instant des impondérables : blessures des joueurs, intempéries, abandons.

1.2 Programmation structurée

1.3 Programme

Ecrire un programme impératif qui modélise un match de tennis. On pourra utiliser des tableaux et une numérotation numérique des joueurs. On pourra aussi coder le codage des points pour faciliter le calcul des scores.

Voici le programme en langage impératif type Pascal correspondant.

```

program tennis;
  (* auteur : P. André, février 2001
   programme inspiré de Nerson *)
  (* Les joueurs sont représentés par des index dans des tableaux.
   Le changement de service est simplement une opération arithmétique.
   Les comptages de points sont des entiers, une table de conversion
   fait éventuellement la conversion pour l'affichage.
   Toutes les limites sont définies par des constantes.
   Le vainqueur d'un échange est calculé aléatoirement pour éviter
   des saisies rébarbatives.
   La partie est naturellement découpée en procédures :
   match, set, jeu, tie-break.
  *)

const
  nb_joueurs = 2;
  nb_sets = 5;
  nb_sets_g = 3;
  nb_jeux = 7;
  nb_pts_jeu = 4;
  nb_pts_tie = 7;
  points = array [1..nb_pts_jeu] of integer = {0, 15, 30, 40};
  pointsup = array [1..2] of string = {'égalité', 'avantage'};
type
  score_jeux = 0..nb_jeux;
  type_jeu = {norm, tie};
var
  joueurs : array [1..nb_joueurs] of string;
  match : array [1..nb_sets, 1..nb_joueurs] of score_jeux;
  serveur : 1..nb_joueurs;
  set_gagnes : array [1..nb_joueurs] of integer;
  set_en_cours : 1..nb_sets;
  score_set : array [1..nb_joueurs] of score_jeux;
  nom : string;
  i, j : integer;

procedure afficher_points_jeu (pt : integer, jeu : type_jeu);
begin
  case jeu of
    norm : write (points[pt]);
    tie : write (pt);
  end; (* case *)
end; (* afficher_points_jeu *)

procedure afficher_msg_jeu (jeu : type_jeu);
begin
  case jeu of
    norm : write ('jeu ');
    tie : write ('tie break ');
  end; (* case *)
end; (* afficher_points_jeu *)

procedure jouer_jeu (nb_pts_max : integer,
  tj : type_jeu,
  var gagnant : 1..nb_joueurs);
  (* Cette procédure sert à la fois aux jeux normaux et au
  tie-break, car les règles sont similaires.
  La distinction se fait par les variables nb_pts_max et tj *)
var
  jeu : array [1..nb_joueurs] of integer;
  fin_jeu : boolean;
  gagn, perd : 1..nb_joueurs;
  i : integer,

```

```

begin
for i ←1 to nb_joueurs do jeu[i] ←0;
fin_jeu ←false;
repeat
  gagn ←random(nb_joueurs); (* un point est joué *)
  perd ←(gagn mod nb_joueurs) + 1;
  (* pas de sens si nb_joueurs <> 2 *)
  jeu[gagn] ←jeu[gagn] + 1;
  if (jeu[gagn] > nb_pts_max) then begin
    case (jeu[gagn] - jeu[perd]) of
      0 : writeln (pointsup[1]);
      1 : writeln (pointsup[2], joueurs[gagn]);
      2 : fin_jeu ←true
    end; (* case *)
  end else begin
    write (joueurs[gagn], ' ');
    afficher_points_jeu (jeu[gagn], tj);
    write (' - ');
    afficher_points_jeu (jeu[perd], tj);
    writeln (joueurs[perd])
  end (* if *)
until fin_jeu;
afficher_msg_jeu (tj);
writeln (joueurs[gagn]);
gagnant ←gagn
end; (* jouer_jeu *)

procedure jouer_set (var serveur : 1..nb_joueurs,
                    var set : array [1..nb_joueurs] of score_jeux;
                    var gagnant : 1..nb_joueurs);

var
  fin_set : boolean;
  gagn : 1..nb_joueurs;
  i: integer;

begin
for i ←1 to nb_joueurs do set[i] ←0;
fin_set ←false;
repeat
  jouer_jeu (nb_pts_jeu, norm, gagn); (* un jeu est joué *)
  perd ←(gagn mod nb_joueurs) + 1;
  (* pas de sens si nb_joueurs <> 2 *)
  if (perd = serveur) then
    writeln ('break pour ',joueurs[gagn]);
  (* end if *)
  set [gagn] ←set [gagn] + 1;
  writeln (joueurs[serveur], ':',set [serveur])
  write (' - ');
  serveur ←(serveur mod nb_joueurs) + 1;
  (* pas de sens si nb_joueurs <> 2 *)
  writeln (joueurs[serveur], ':',set [serveur])
  if (set [gagn] > nb_sets) then begin
    case (set [gagn] - set [perd]) of
      0 : begin
          writeln ('tie break');
          jouer_jeu (nb_pts_tie, tie, gagn);
          fin_set ←true
        end; (* case 0 *)
      2 : fin_set ←true
    end; (* case *)
  (* end if *)
until fin_set;

```

```

writeln ('set ', joueurs[gagn]);
gagnant ←gagn
end;    (* jouer_set *)

begin  (* Début du programme *)
for i ←1 to nb_joueurs do begin
  readln('nom du joueur : ', nom);
  joueurs[i] ←nom;
  for j ←1 to nb_sets do match[j, i] ←0
end; (* for *)
serveur ←random(nb_joueurs);
set_en_cours ←0;
repeat
  set_en_cours ←set_en_cours + 1;
  jouer_set (serveur, score_set, gagnant);
  (* le changement de service y est effectué *)
  for i ←1 to nb_joueurs do
    match[set_en_cours, i] ←score_set[i];
  (* end for *)
  set_gagnes[gagnant] ←set_gagnes[gagnant] + 1;
until (set_gagnes[gagnant] = nb_sets_g);
writeln('Le joueur ', joueurs[gagnant], ' a gagné en ', set_en_cours, ' sets. ');
writeln('Le score final est : ');
for i ←1 to nb_joueurs do begin
  writeln(joueurs[i], ' : ');
  for j ←1 to set_en_cours do
    write (match[j, i], ' ');
  writeln(' ')
end; (* for *)
end.  (* Fin du programme *)

```

Figure 55 : Programme impératif du match de tennis

Evolution de l'énoncé

On souhaite pouvoir adapter le programme aux évolutions suivantes.

1. *Un joueur peut abandonner la partie.*
2. *Dans un tournoi certains matchs se jouent en deux sets gagnants (début du tournoi, tournoi féminin...).*
3. *On souhaite inclure ce programme dans le cadre d'un tournoi complet et conserver tous les scores jusqu'à la finale.*
4. *On souhaite modifier le programme pour des matchs de doubles.*
5. *On souhaite écrire un programme similaire pour un tournoi de tennis de table.*

Commenter l'adaptabilité du programme aux modifications souhaitées.

Nous avons utilisé un maximum de constantes pour paramétrer le programme. Une factorisation a été obtenue en groupant jeux et tie-break. Nous avons, autant que possible, rendu indépendantes les différentes procédures (cohérence, faible couplage). La structure du programme est relativement visible.

Discutons maintenant des différentes évolutions proposées.

1. Un joueur peut abandonner la partie. L'abandon est une fonction aléatoire à l'intérieur de la procédure `jouer_un_jeu`. On arrête un set s'il y a un abandon dans un jeu. On peut utiliser les variables `fin_jeu`, `fin_set` et `match_fin` pour arrêter un jeu ou un set en cours. On ajoute un paramètre dans chaque procédure qui indique que la partie est terminée pour propager la décision. A l'appel d'un nouveau set ou d'un nouveau jeu, on vérifie les points suivants. On joue un jeu s'il n'y a pas d'abandon dans le jeu précédent. L'ensemble du programme est modifié même si les modifications sont mineures.

2. Matches en deux sets gagnants.

Il suffit de changer la constante `nb_sets_g` en 2 au lieu de 3. On peut aussi affiner `nb_sets = 3`. On gagne ainsi de l'espace mémoire.

3. Tournoi complet.

Le programme principal et ses déclarations de variables sont rangés dans une procédure. Il faut ajouter de nouvelles structures de données pour stocker les résultats des différents matchs. On peut s'inspirer de l'analogie set-match pour concevoir le binôme match-tournoi. Ainsi, les résultats d'un matchs sont recopiés au niveau du tournoi. La structure des joueurs est à élever au niveau du tournoi. Il faut faire le lien entre les joueurs d'un match et ceux du tournoi.

4. Matches en doubles.

Soit on considère l'équipe et non le joueur et il n'y a rien à changer. Soit il faut dissocier la notion de joueur (conservée pour le changement de service) de la notion d'équipe (mémorisation des scores).

5. Tennis de tables.

On change les règles de calcul des points (constantes) et la règle de jeu d'un set. Le changement de service est aussi différent. Cette dernière modification implique des modifications assez subtiles dans l'ensemble du programme.

Enseignements

Le paramétrage du programme est essentiel à son évolution. Un soin particulier est à apporter à la lisibilité des actions.

Il est difficile de mesurer l'impact d'une modification. Il faut repérer données et instructions relatives à une même action. Selon les modifications, l'ensemble du programme peut être remis en cause.

La réutilisation de morceaux de code implique une forte abstraction lors de l'écriture.

1.4 Conception abstraite à objets

Proposer une conception abstraite à objets. On proposera par exemple une modélisation avec la notation UML.

La lecture de l'énoncé fait apparaître quatre abstractions distinctes : les joueurs, les matchs, les sets et les jeux. En fait, on remarque une imbrication des différents objets entre eux : les joueurs d'un set sont ceux du matchs du set.

- Joueur. Un joueur est simplement caractérisé par un nom, un prénom et son classement.
- Match. Un match est une partie entre deux joueurs. Le match est défini par un ensemble de sets (une liste si on souhaite mémoriser la progression). Le match est terminé dès qu'un joueur a remporté le nombre de sets gagnants.
- Un set une partie d'un match. Un set est défini par un ensemble de jeux (une liste si on souhaite mémoriser la progression). On souhaite ici ne mémoriser que le score de chaque jeu du set. Le set est terminé lorsqu'un joueur a remporté six jeux avec deux jeux d'écart ou un tie-break.
- Un jeu met en compétition deux joueurs, le serveur et le receveur. Le jeu se joue en 4 points minimum. Le vainqueur a au moins 4 points avec deux points d'acart. les points sont notés selon les convention 0, 15, 30, 40 et la règle de l'égalité ou de l'avantage. Un jeu gagné par le receveur est un avantage psychologique, appelé *break*.

Nous proposons dans la figure 56 une conception abstraite du problème. Nous avons adopté une notation à la Smalltalk des variables et associations pour faciliter l'implantation. La navigation des associations indique une orientation non symétrique des liens entre objets.

Nous proposons, comme pour le programme structuré, des constantes pour paramétrer le programme.

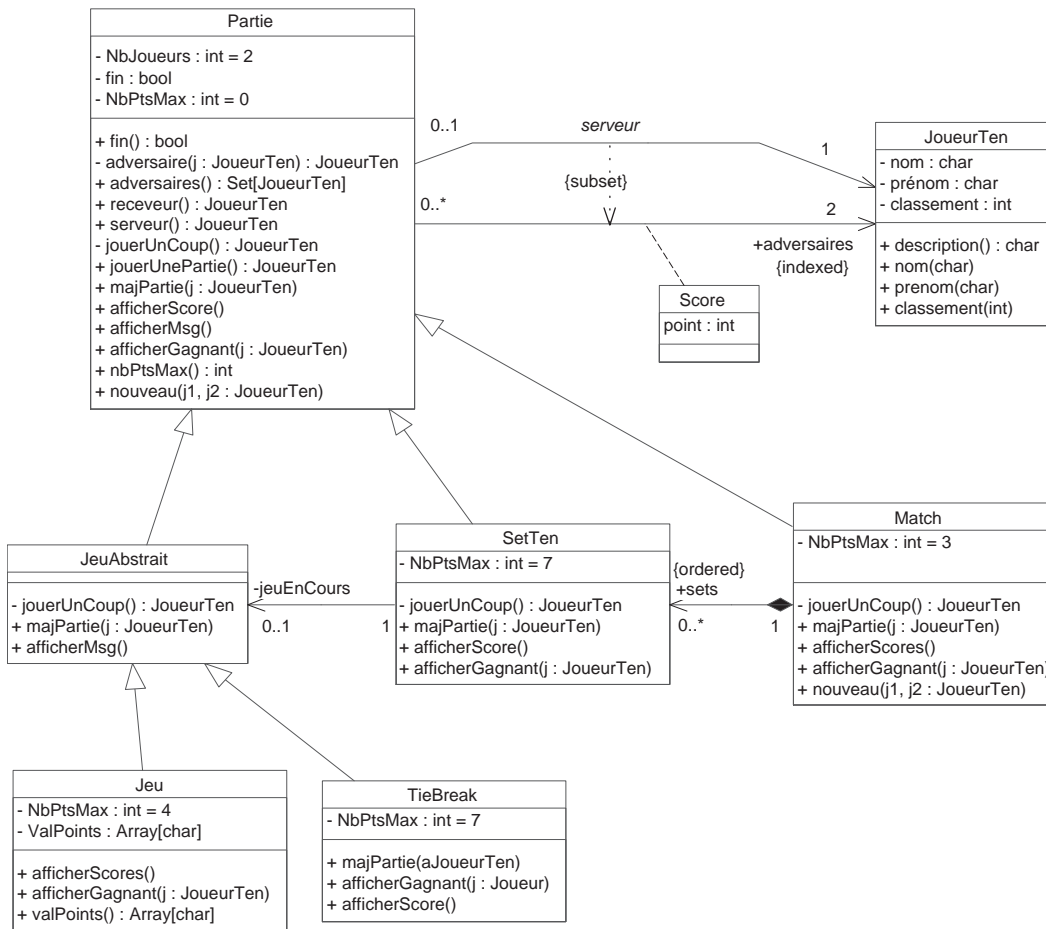


Figure 56 : Conception objet du programme *tennis*

Nous avons mis en évidence un schéma d'abstraction : match, set, jeu et tie-break sont des parties entre adversaires. Il y a `NbJoueurs` adversaires (variable de classe). Pour simplifier, concerne deux joueurs (opération `adversaires`) dont l'un doit servir (opération `serveur`). Ces joueurs sont indexés (à partir du joueur, on trouve son score). Le receveur est l'adversaire qui ne sert pas (opération `receveur`). Le principe est le suivant, la partie est une suite de coups (opération `jouerUnePartie`). Chaque coup est gagné par un joueur qui augmente son score (opération `jouerUnCoup`). La fin de la partie est déterminée dans la mise-à-jour de la partie (opération `majPartie`) et stockée dans la variable `fin` (opération `fin`). Elle est fonction de nombre de points à atteindre (variable de classe `NbPtsMax`). Cette variable évoluant dans les sous-classe, elle sera implantée comme une variable d'instance de la métaclasse en Smalltalk. Un message est affiché en début de partie (opération `afficherMsg`), à chaque coup (opération `afficherScore`) et à la fin de la partie (opération `afficherGagnant`).

context `Partie` inv:

```

self . adversaires -> size = Partie.nbJoueurs
-- nous utilisons une notation propre pour accéder à la variable de classe
self . adversaires -> includesAll(self.serveur)
-- le serveur est un des adversaires
self . adversaires -> forAll( j : JoueurTen | self.score[j] <= Partie.nbPtsMax )
-- le score est limité, s'agissant d'une collection indexée,
-- nous utilisons la notation des associations qualifiées

```


Cette structure est ensuite redéfinie au besoin dans les sous-classes. Nous avons ajouté un niveau d'abstraction pour marquer les éléments communs entre jeu et tie-break. Jouer un coup signifie jouer un points. L'affichage des scores, l'évolution de la partie et le nombre de points diffèrent entre jeu et tie-break. L'affichage des points des jeux ordinaires est noté 0, 15, 30, 40, égalité, avantage (opération `valPoints`). Le changement de service intervient dans le tie-break (opération `majPartie`). Les messages sont personnalisés.

Un set est une partie dont les coups sont des jeux. On ne mémorise dans le score que les résultats des jeux. Le set s'arrête si le nombre de jeux gagnants est atteint et qu'il y a un écart de deux jeux. Un tie break est joué si le score est de 6-6. Le changement de service intervient à chaque jeu (opération `majPartie`). Les affichages de scores varient aussi.

Un match est une partie dont les coups sont des sets. On ne mémorise dans l'ordre les résultats des sets (association `sets`). Le match s'arrête si le nombre de sets gagnants est atteint. Le changement de service est fonction du set précédent. Au départ, le serveur est tiré au sort (opération `nouveau`). Les affichages de scores varient aussi.

Les contraintes suivantes sont ajoutées aux contraintes de la classe `Partie`. Les joueurs d'un set sont les joueurs du match du set. Le nombre maximum de sets d'un match est de $(\text{Match.nbPtsMax} * 2) - 1$ par exemple 5 avec 3 sets gagnants. Si le match est fini, le vainqueur a exactement `nbPtsMax` sets gagnants.

```
context Match inv:
  self .sets->forAll( s : SetTen | s.adversaires = self.adversaires )
  self .sets->size <= (Match.nbPtsMax * 2) - 1
  -- nbPtsMax est le nombre de sets gagnants d'un match (var. de classe)
  self .fin() implies
    self .sets.adversaires->exists( j : JoueurTen |
      self.score[j] = Match.nbPtsMax and
      ( self .sets.adversaires->forAll( i : JoueurTen |
        i <> j and self.score[i] < Match.nbPtsMax ))
    )
  -- on vérifie l'unicité
```

Pour le set en cours, les adversaires du jeu en cours sont ceux du set. Si le jeu est fini, alors au moins un des scores est supérieur à `nbPtsMax - 1` (en principe le set se joue en 6 jeux).

```
context SetTen inv:
  self .jeuEnCours.adversaires = self.adversaires
  self .fin() implies
    self .sets.adversaires->exists( j : JoueurTen |
      self.score[j] >= SetTen.nbPtsMax - 1 and
      ( self .sets.adversaires->forAll( i : JoueurTen |
        i <> j and (( self.score[i] <= self.score[j] - 2) or
          ( self.score[i] = SetTen.nbPtsMax - 1 and
            self.score[j] = SetTen.nbPtsMax ))))
    )
  -- on vérifie l'unicité avec un éventuel tie-break
```

Dans la classe `JeuAbstrait`, nous avons la contrainte suivante : si le jeu ou le tue-break est fini, alors au moins un des scores est supérieur ou égal à `nbPtsMax`.

```
context SetTen inv:
  self .fin() implies
    self .sets.adversaires->exists( j : JoueurTen |
      self.score[j] >= SetTen.nbPtsMax and
      ( self .sets.adversaires->forAll( i : JoueurTen |
        i <> j and (self.score[i] <= self.score[j] - 2)))
    )
```

1.5 Implantation en Smalltalk

Implanter la conception à objets en Smalltalk. Discuter des alternatives de codage.

Le code ci-après est une traduction quasi-systématique de la conception. Les assertions n'ont pas été programmées sous cette forme, mais on peut vérifier qu'elles sont valides. Un certain nombre d'optimisations dues à l'héritage ont été effectuées.

Les associations sont représentées par des variables d'instances, sauf `jeuEnCours` car c'est une variable locale. Du fait des choix de navigation, une seule variable suffit. Elle porte le nom du rôle associé et se trouve dans la classe à l'origine de la navigation. La contrainte `indexed` est représentée en utilisant les dictionnaires de Smalltalk :

```
score : Dictionary [JoueurTen] of Integer.
```

L'héritage est simple et ne pose pas de difficultés. Noter que la variable de classe `NbPtsMax` est définie par une variable d'instance de ma métaclasse car sa valeur est redéfinie par la méthode de classe `initialize` dans les sous-classes de `Partie`.

Code Smalltalk de l'application tennis

Listing C.1 – Code Smalltalk de l'application tennis

```
Object subclass: #JoueurTen
  instanceVariableNames: 'nom prenom classement '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Tennis'!

JoueurTen comment:
  'Cette classe implante la classe JoueurTen du match de tennis.

Les variables d''instance sont :
nom : String
prenom : String
classement : Number'!

!JoueurTen methodsFor: 'printing'!

description
  "Ecrit le nom, le prénom et le classement dans le flot ."

  | aStream |
  aStream ← WriteStream on: (String new: 16).
  self description: aStream.
  ↑aStream contents!

description: aStream
  "Ecrit le nom et le prénom dans le flot ."

  aStream cr; nextPutAll: 'Je m''appelle ', nom, ', ', prenom.
  aStream nextPutAll: ' et je suis classé ', classement printString!

printOn: aStream
  "Ecrit le nom et le prénom dans le flot ."

  aStream nextPutAll: nom, ', ', prenom! !

!JoueurTen methodsFor: 'accessing'!

classement
  "rend le classement du JoueurTen"

  ↑classement!
```

```

classement: aNumber
    "affecte le classement du joueur"

    classement ← aNumber!

nom
    "rend le nom du joueur"

    ↑ nom!

nom: aString
    "affecte le nom du joueur"

    nom ← aString!

prenom
    "rend le prenom du joueur"

    ↑ prenom!

prenom: aString
    "affecte le prenom du joueur"

    prenom ← aString!

"***** !"
Object subclass: #Partie
    instanceVariableNames: 'score fin serveur '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Tennis'!
Partie comment:
'Cette classe implante une version abstraite d'une partie du match de tennis.
Elle rassemble le comportement commun à un set, un jeu et un tie break.
La partie est définie par des adversaires dont on mémorise les scores
dans un dictionnaire, l'adversaire qui sert pour le gain de la partie
(serveur) un booléen indiquant si la partie est terminée ou pas.

Les variables d'instance sont :
    score : Dictionary [JoueurTen] : Integer
    serveur : JoueurTen
    fini : Boolean

- la variable score mémorise le score actuel de chaque adversaire du
  jeu dans un dictionnaire
- la variable fini mémorise la fin du jeu. Sa valeur pourrait être
  calculée à partir du score, mais nous utilisons une variable pour
  optimiser le temps de traitement.
- la variable serveur indique le joueur qui sert pour ce jeu.

Les variables d'instance de la métaclasse sont :
    NbPtsMax : Integer nombre de coups nécessaires pour gagner la partie.'!

!Partie methodsFor: 'initialize-release'!

initializeJ1 : aJoueurTen1 j2: aJoueurTen2
    "Démarre un nouveau jeu, les scores sont mis à zéro pour chaque

```

joueur. Par défaut, le serveur est le premier joueur."

```
fin ← false.
score ← Dictionary withKeysAndValues: (Array
    with: aJoueurTen1
    with: 0
    with: aJoueurTen2
    with: 0).
serveur ← aJoueurTen1.
↑self !!
```

!Partie methodsFor: 'printing'!

```
afficherGagnant: aJoueurTen
    Transcript cr; show: aJoueurTen printString , ' gagne'!
```

```
afficherMsg
    "Affichage d'un message. Version abstraite .!"
```

```
afficherScores
    "Affichage du score de la partie. Version abstraite ."
```

```
    Transcript show: (score at: self serveur) printString , ' - ' ,
        (score at: self receveur) printString!!
```

!Partie methodsFor: 'update'!

```
jouerUnCoup
    "On tire le gagnant au sort.
    Si le gagnant est 1 alors le serveur gagne, sinon le receveur gagne."
```

```
| gagnant sort |
sort ← (Random new next * 10000 rem: 2) truncated + 1.
sort = 1
    ifTrue: [gagnant ← self serveur]
    ifFalse: [gagnant ← self receveur].
score at: gagnant put: (score at: gagnant)
    + 1.
↑gagnant!
```

```
jouerUnePartie
    "On joue les coups jusqu'à ce que la partie soit terminée."
    "On rend le gagnant"
```

```
| gagn |
self afficherMsg.
[self fin]
    whileFalse:
        [gagn ← self jouerUnCoup.
        self afficherScores.
        self majPartie: gagn].
self afficherGagnant: gagn.
↑gagn!
```

```
majPartie: aJoueurTen
    "On met à jour la partie connaissant le gagnant d'un coup"
```

```
fin ← true! !
```

```

!Partie methodsFor: 'private'!

adversaire: aJoueurTen
    "rend l'adversaire d'un joueur"

    | adv |
    (score keys includes: aJoueurTen)
        ifFalse: [↑nil]
        ifTrue: [self adversaires do: [:a | a == aJoueurTen
            ifFalse: [adv ←a]]].
    ↑adv! !

!Partie methodsFor: 'accessing'!

adversaires
    "rend les adversaires opposés dans le jeu"

    ↑score keys!

fin
    "Détermine la fin de la partie. Dans cette version on rend la valeur
    de la variable "

    ↑fin! !

receveur
    "rend le serveur du jeu"

    ↑self adversaire: self serveur!

serveur
    "rend le serveur du jeu"

    ↑serveur! !
"-----"!

Partie class
    instanceVariableNames: 'NbPtsMax' !

!Partie class methodsFor: 'instance creation'!

nouveauJ1: aJoueurTen1 j2: aJoueurTen2
    "Démarre une nouvelle partie "

    ↑self new initializeJ1 : aJoueurTen1 j2: aJoueurTen2! !

!Partie class methodsFor: 'accessing'!

nbPtsMax
    "accès à la variable d'instance de la métaclasse "

    ↑NbPtsMax! !

!Partie class methodsFor: 'class initialization'!

initialize
    "Partie initialize "

```

```

NbPtsMax ←0.!!

"***** !"
Partie subclass: #JeuAbstrait
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Tennis'!
JeuAbstrait comment:
'Cette classe implante une version abstraite de la classe Jeu du
match de tennis. Elle est définie par spécialisation de la classe
abstraite Partie. Elle rassemble le comportement commun à un jeu et
un tie break.
  jouerUnCoup signifie jouerUnPoint ici
  jouerUnePartie signifie jouerUnJeu ici'!

!JeuAbstrait methodsFor: 'update'!

jouerUnCoup
  "On tire le gagnant au sort.
  Si le gagnant est 1 alors le serveur gagne, sinon le receveur gagne."

  | gagnant sort |
  sort ←(Random new next * 10000 rem: 2) truncated + 1.
  sort = 1
    ifTrue: [gagnant ←self serveur]
    ifFalse: [gagnant ←self receveur].
  score at: gagnant put: (score at: gagnant)
    + 1.
  ↑gagnant!

majPartie: aJoueurTen
  "On met à jour la partie connaissant le gagnant d'un coup"

  | gagn sg sp |
  gagn ←aJoueurTen.
  sg ←score at: gagn.
  sp ←score at: (self adversaire: gagn).
  sg >= self class nbPtsMax & (sg - sp >= 2) ifTrue: [fin ←true]!!

!JeuAbstrait methodsFor: 'printing'!

afficherMsg
  "Affichage d'un message en début de partie."

  Transcript cr; show: self serveur printString , ' sert'!!

"***** !"
JeuAbstrait subclass: #Jeu
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Tennis'!
Jeu comment:
'Cette classe implante la classe Jeu du match de tennis.
Elle est définie par héritage de JeuAbstrait.
Le jeu évolue selon la spécification de la classe abstraite JeuAbstrait.

```

Seul l'affichage des scores change.

Les variables d'instance de la métaclasse sont :

NbPtsMax : Integer est redéfinie avec la valeur 4.

ValPoints : Array[String] valeur d'affichage des points

Cette dernière variable permet de transcrire le score selon les règles habituelles :

elle contient une table de codage : score --> affichage.

Elle est utilisée par une méthode de classe valPoints et initialisée à la création de la classe.!

!Jeu methodsFor: 'printing'!

afficherGagnant: aJoueurTen

super afficherGagnant: aJoueurTen.

Transcript show: ' le jeu'!

afficherScores

"Affichage du score du jeu"

| gagn |

(score at: self serveur)

> (score at: self receveur)

ifTrue: [gagn ←self serveur]

ifFalse: [gagn ←self receveur].

(score at: gagn)

>= self class nbPtsMax

ifTrue:

[[diff |

diff ←(score at: gagn)

– (score at: (self adversaire: gagn)).

diff = 0

ifTrue: [Transcript cr; show: 'égalité']

ifFalse: [diff = 1

ifTrue: [Transcript cr; show: 'avantage ', gagn printString]

ifFalse: [fin ←true]]]

ifFalse: [Transcript cr; tab; show:

(self class valPoints at: (score at: self serveur) + 1)

, ' - ', (self class valPoints at:

(score at: self receveur)+ 1)]! !

"-----"

Jeu class

instanceVariableNames: 'ValPoints'!

!Jeu class methodsFor: 'class initialization'!

initialize

"Jeu initialize "

NbPtsMax ←4.

ValPoints ←#('0' '15' '30' '40' 'jeu')! !

!Jeu class methodsFor: 'accessing'!

valPoints

"accès à la variable d'instance de la métaclasse"

↑ValPoints! !

```

***** !
JeuAbstrait subclass: #TieBreak
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Tennis'!
TieBreak comment:
'Cette classe implante la classe Tie Break du match de tennis.
Elle est définie par héritage de JeuAbstrait.

Seule l'évolution du jeu change : le serveur change tous les points pairs.

Les variables d'instance de la métaclasse sont :
  NbPtsMax : Integer    est redéfinie avec la valeur 7. '!'

!TieBreak methodsFor: 'update'!

majPartie: aJoueurTen
  "On met à jour la partie connaissant le gagnant d'un coup.
  Cette méthode est redéfinie pour inclure le changement de service ."

  | gagn sg sp |
  gagn ← aJoueurTen.
  sg ← score at: gagn.
  sp ← score at: (self adversaire: gagn).
  sg >= self class nbPtsMax & (sg - sp >= 2)
    ifTrue: [fin ← true]
    ifFalse: [(sg + sp rem: 2)
              = 1
              ifTrue:
                [serveur ← self receveur.
                 Transcript tab; show: 'changement de service ', self serveur printString]]! !

!TieBreak methodsFor: 'printing'!

afficherGagnant: aJoueurTen
  super afficherGagnant: aJoueurTen.
  Transcript show: ' le tie break'!

afficherScores
  "Affichage du score de la partie. Version abstraite ."

  Transcript cr; tab.
  super afficherScores! !
"----- !

TieBreak class
  instanceVariableNames: ''!

!TieBreak class methodsFor: 'class initialization'!

initialize
  "TieBreak initialize "

  NbPtsMax ← 7.! !

***** !
Partie subclass: #SetTen

```



```

instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Tennis'!
SetTen comment:
'Cette classe implante la classe SetTen (pour ne pas confondre avec la
classe Set) du match de tennis. Elle est définie par héritage de JeuAbstrait.

jouerUnCoup signifie jouerUnJeu ici
jouerUnePartie signifie jouerUnSet ici

L'évolution du jeu et l'affichage des scores changent.
Le serveur change à chaque jeu.

Les variables d'instance de la métaclasse sont :
NbPtsMax : Integer est redéfinie avec la valeur 7. '!'

!SetTen methodsFor: 'printing'!

afficherGagnant: aJoueurTen
super afficherGagnant: aJoueurTen.
Transcript show: ' le set'!

afficherScores
"Affichage du score du jeu. Version Set."

Transcript cr; tab; show: 'Set : ', self serveur printString , ' '.
super afficherScores .
Transcript show: ' ', self receveur printString! !

!SetTen methodsFor: 'update'!

jouerUnCoup
"Jouer un coup signifie jouer un jeu dans le set.
Le serveur est donné en premier. "

| jeuEnCours gagnant |
jeuEnCours ←Jeu nouveauJ1: self serveur j2: self receveur.
gagnant ←jeuEnCours jouerUnePartie.
score at: gagnant put: (score at: gagnant)
+ 1.
↑gagnant!

majPartie: aJoueurTen
"On joue un set, on change de serveur entre chaque jeu.
Si le score est de 6 à 6, on joue un tie-break. "

| gagn sg sp |
gagn ←aJoueurTen.
gagn = self receveur ifTrue: [Transcript tab; tab; show:
' / Break pour ', self receveur printString].
sg ←score at: gagn.
sp ←score at: (self adversaire: gagn).
sg >= (self class nbPtsMax - 1) & (sg - sp = 1) not
ifTrue:
[sg - sp >= 2
ifTrue: [fin ←true]
ifFalse:

```

```

    [ "sg = sp"
      Transcript cr; cr; show: 'Tie break'.
      gagn ←(TieBreak nouveauJ1: self receveur j2: self
              serveur) jouerUnePartie.
      score at: gagn put: (score at: gagn)
              + 1].
    fin ←true]
  ifFalse: [serveur ←self receveur]! !
"-----"!

SetTen class
  instanceVariableNames: ''!

!SetTen class methodsFor: 'class initialization'!

initialize
  "SetTen initialize "

  NbPtsMax ←7.! !

"*****"!
Partie subclass: #Match
  instanceVariableNames: 'sets adversaires '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Tennis'!
Match comment:
'Cette classe implante la classe Match de tennis. Elle est définie
par héritage de JeuAbstrait car elle définit deux adversaires et
le déroulement d'une partie.

Les variables d'instance sont :
  sets : OrderedCollection [SetTen]

- la variable sets mémorise dans l'ordre les sets joués
- le score contient le nombre de sets gagnés par joueur.

Les variables d'instance de la métaclasse sont :
  NbPtsMax : Integer   nombre de sets gagnants pour gagner le match.

jouerUnCoup signifie jouerUnSet ici
jouerUnePartie signifie jouerUnMatch ici'!

!Match methodsFor: 'printing'!

afficherGagnant: aJoueurTen
  super afficherGagnant: aJoueurTen.
  Transcript show: ' le match'!

afficherScores
  "Affichage du score du match.
  Problème : l'ordre des joueurs dans les sets peut varier d'un set à l'autre."

  Transcript cr; tab; show: 'Score du match : '.
  sets do: [:s | s afficherScores]! !

!Match methodsFor: 'update'!

```

jouerUnCoup

*"Jouer un coup signifie jouer un set dans le match.
Le serveur est donné en premier. Les scores sont mémorisés.
La fin du set détermine le changement de service."*

```
| gagnant setEnCours |
setEnCours ← SetTen nouveauJ1: self serveur j2: self receveur.
gagnant ← setEnCours jouerUnePartie.
score at: gagnant put: (score at: gagnant)
    + 1.
sets add: setEnCours.
serveur ← setEnCours receveur.
↑gagnant!
```

majPartie: aJoueurTen

*"On met à jour le match connaissant le gagnant d'un set.
Cette méthode est redéfinie pour inclure le changement de service.
On s'arrête lorsque le nombre de set gagnants est atteint."*

```
(score at: aJoueurTen)
    = self class nbPtsMax ifTrue: [fin ← true]! !
```

```
!Match methodsFor: 'initialize-release'!
```

initializeJ1 : aJoueurTen1 j2: aJoueurTen2

*"Démarre un nouveau match.
Par défaut, le serveur est tiré au sort."*

```
super initializeJ1 : aJoueurTen1 j2: aJoueurTen2.
(Random new next * 10000 rem: 2) truncated + 1 = 1
    ifTrue: [serveur ← aJoueurTen1]
    ifFalse: [serveur ← aJoueurTen2].
Transcript cr; show: 'Le tirage au sort donne le service à ',
    serveur printString , '.'.
sets ← OrderedCollection new.
↑self! !
```

```
"-----"
```

Match class

```
instanceVariableNames: ''!
```

```
!Match class methodsFor: 'class initialization'!
```

initialize

```
"Match initialize"
```

```
NbPtsMax ← 3! !
```

```
!Match class methodsFor: 'examples'!
```

unMatch

```
"Match unMatch"
```

```
| joueur1 joueur2 leMatch |
joueur1 ← (JoueurTen new) nom: 'Pierre'; prenom: 'Afeux'; classement: 12.
joueur2 ← (JoueurTen new) nom: 'René'; prenom: 'Gossie'; classement: 11.
Transcript cr; show: joueur1 description; show: joueur2 description.
leMatch ← self nouveauJ1: joueur1 j2: joueur2.
```

leMatch jouerUnePartie! !

Partie initialize !
 Match initialize !
 SetTen initialize !
 Jeu initialize !
 TieBreak initialize !

Exécution

1.6 Evolution de la conception

Reprendre la question de l'évolution avec la conception à objets.

Nous avons utilisé des variables de classes pour paramétrer le programme. Une factorisation a été obtenue par abstraction dans la classe `Partie`. Une fois cette classe conçue et programmée, l'écriture des sous-classe est rapide. Il faut noter qu'à chaque fois qu'un comportement commun est mis en évidence, il est factorisé dans les super-classes. La structure du programme est bien visible par le modèle des classes de la figure 56.

Discutons maintenant des différentes évolutions proposées.

1. Un joueur peut abandonner la partie. L'abandon est une fonction aléatoire à l'intérieur de la procédure `jouerUnePartie`. A priori, il suffit de mettre à jour la variable `fin` pour arrêter le jeu. Deux précautions sont à prendre :
 - (a) Pour répercuter, l'abandon dans un jeu au niveau du set puis du match, on rend un résultat supplémentaire dans la méthode `jouerUnCoup`. Dans la méthode appelante, on teste ce résultat.
 - (b) Les contraintes posées dans la conception ne sont valables que si le jeu est fini normalement.

La solution consiste donc à ajouter une variable d'instance `abandon` dans la classe `Partie` et à modifier les méthodes suivantes :

- (a) La méthode `jouerUnCoup` rend un paramètre supplémentaire..
- (b) La méthode `jouerUnePartie` boucle jusqu'à la fin ou l'abandon.

Il faut vérifier la cohérence avec la redéfinition de ces méthodes dans les sous-classes. Seule une partie (verticale) du code est modifiée.

2. Matches en deux sets gagnants.

Il suffit de changer la constante `NbPtsMax` en 2 au lieu de 3. dans la classe `Match`.

3. Tournoi complet.

Un tournoi complet est un ensemble de matchs. Des contraintes doivent être ajoutées sur les matchs joués : tours, adversaires qualifiés, etc. Le code existant n'est nullement remis en cause : c'est la réutilisation horizontale.

4. Matches en doubles.

Soit on considère l'équipe et non le joueur et il n'y a rien à changer. Soit il faut dissocier la notion de joueur (conservée pour le changement de service) de la notion d'équipe (mémoire des scores).

5. Tennis de tables.

On change les règles de calcul des points (constantes) et la règle de jeu d'un set. Le changement de service est aussi différent.

Les deux dernières évolutions implique des modifications assez subtiles dans l'ensemble du programme. Nous avons deux possibilités :

- Soit on recopie l'ensemble de la hiérarchie de classe, en renommant les classes. Puis on modifie localement les méthodes `jouerUnCoup`, `jouerUnePartie` `majPartie` et les variables de classes.
- Soit on définit un schéma d'héritage multiple comme le montre partiellement la figure 57. Pour être exploitable, il faudrait définir un héritage de groupes de classes, qu'on peut qualifier de *pattern* ici.

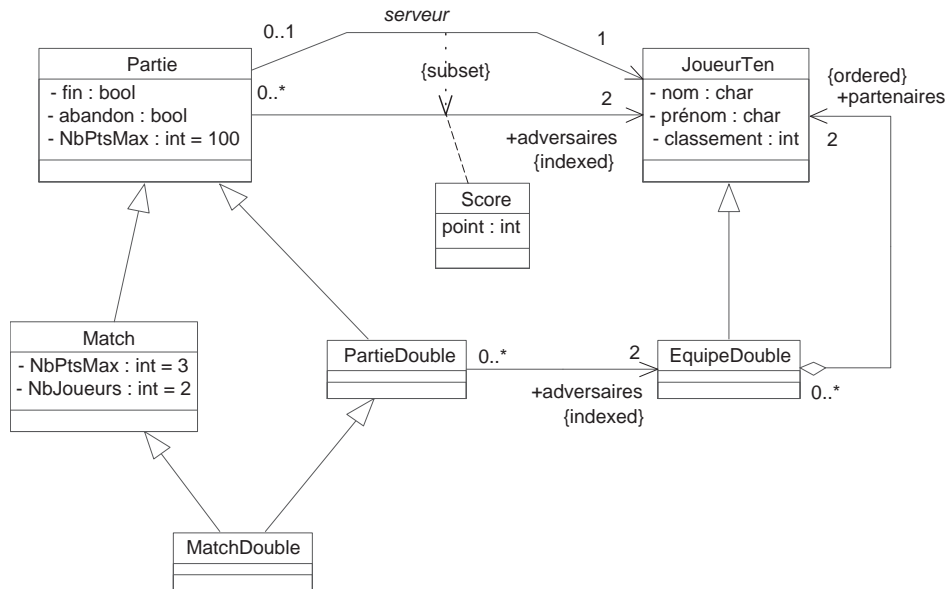


Figure 57 : Conception objet du programme tennis en double

Enseignements

La conception objet est plus perméable aux évolutions que la conception structurée. Par réutilisation verticale (héritage) ou horizontale (copie ou composition), il est souvent aisé de modifier une conception. Néanmoins, l'ajout de nouvelles sous-classe peut amener, pour une utilisation optimale de l'héritage et une meilleure lisibilité du code, à restructurer les classes de niveau intermédiaire.

1.7 Vers un pattern

Nous avons mis en évidence deux *patterns*. Le premier regroupe l'ensemble des classes de la figure 56. Deux personnalisations sont : parties de doubles et parties de tennis de table. Le second regroupe l'ensemble des classes relative à une partie d'un jeu dans la figure 58.

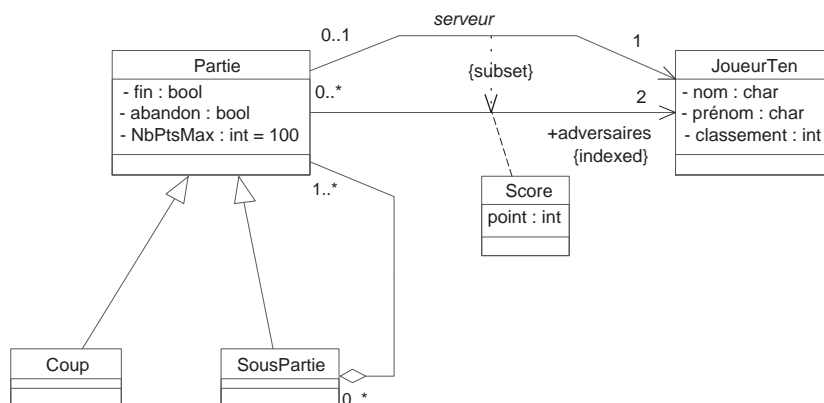


Figure 58 : Conception objet du programme tennis en double

1.8 Bilan

La conception abstraite est en fait issue d'une suite d'itérations, **conception - implantation**. Les classes de Smalltalk jouent un rôle dans la réutilisation.

A travers ce petit exemple, nous avons mis en évidence qu'un découpage en objet permet de définir une meilleure abstraction du code. Ce découpage facilite la lisibilité, la conception, le test des différents modules. La cohésion des objets et l'héritage facilitent l'extension de l'application et la réutilisation de code. Noter aussi que la notation UML est un bon outil de documentation du code.

2 Générateur de compilateurs

Des outils similaires à Lex et Yacc ont été mis au point pour Smalltalk. Il faut pour les utiliser importer le fichier `AP-ParserCompiler.st`. Nous en faisons ici un bref exposé.

2.1 Compilation

Les règles de grammaires sont écrites sous forme de méthodes avec une syntaxe particulière. Dans une sous-classe de la classe `ExternalLanguageParser`. Cette classe contient des méthodes pour les règles de grammaires et des méthodes Smalltalk habituelles pour la génération de code. Nous détaillons la syntaxe de la grammaire avant de présenter quelques exemples.

Notations BNF

Pour simplifier, nous donnons les grammaires, auxquelles on a ôté les actions sémantiques.

Les règles sont de type `left = right`. Les symboles non terminaux sont des suites de caractères. Les symboles terminaux sont soit des chaînes de caractères (symboles au sens Smalltalk) préfixées par un `#` soit des caractères préfixés par un `$`. Un certain nombre de tokens sont déterminés par le parser Smalltalk (`number`, `string` ("`. .`"), `word`, etc.). Le signe `@` devant une règle indique que qu'un retour est possible dans l'évaluation en cas d'erreur. Comme dans les conventions usuelles, `|` désigne l'alternative, `*` l'itération positive ou nulle, `+` l'itération positive, `\$`, l'itération positive avec le caractère "`,`" comme séparateur.

2.2 Utilisation

L'analyseur s'utilise

- soit en invoquant une méthode (un non-terminal de la grammaire) avec un texte en entrée (une chaîne de caractères)
- soit en définissant la classe de l'analyseur comme compilateur par défaut d'une autre classe dont les corps des méthodes deviennent alors les chaînes à analyser. Dans ce cas, l'axiome invoqué est `method`. Le résultat est alors un couple : nom de la méthode, objet associé. De même ou pourra changer la méthode de formatage par défaut.

L'analyse se fait en utilisant une pile (variable d'instance `stack`), accessible dans les actions.

2.3 Exemples commentés

Les exemples ci-après sont extraits d'une application de saisie et édition de classement de football. L'application est décomposée en deux parties : les classes de la compilation et les classes du domaine. L'analyse d'un texte en entrée génère une structure d'objets du domaine.

Axiome par défaut

L'axiome par défaut, du moins lorsqu'on utilise l'analyseur comme compilateur d'une classe est `method`.

```
!FootCompiler methodsFor: 'general parsing'!

method =
  [Transcript clear] .
  [Transcript show: 'Analyse du texte saisi ... '; cr] .
  axiome
  [ Transcript show: '... fin de l''analyse'; cr] .! !
```

Dans cet exemple, on vide la console puis on affiche un message de bienvenue. Les envois de message entre [] sont des actions associées à la grammaire. Le résultat des envois de message est empilé sur la pile d'analyse. Le point qui suit l'action dépile le sommet car il n'a pas d'intérêt pour la suite de l'analyse.

N.B. Le résultat des actions est empilé sur la pile d'analyse. N.B. Les caractères ! sont des délimiteurs de la version textuelle du code Smalltalk, ils ne font pas partie de la grammaire.

Axiome par défaut

L'axiome définit par une expression régulière une inscription, un nombre quelconque de saisies ou d'éditations et une édition. Chaque élément donne lieu à une autre règle de grammaire (une autre méthode). L'opérateur `*` évalue plusieurs fois l'expression associée. Le résultat dans la pile est un tableau des éléments évalués.

```
!FootCompiler methodsFor: 'yacc : analysis'!

axiome =
  " les inscriptions modifient la variable champ.
  Pour des problèmes de conflits , nous avons ajouté un séparateur
  avant l' édition finale .
  La solution de retour arrière avec @ ne marche pas ici .
  Les points Âttent le sommet de pile. après l' itération *, nous avons
  un tableau vide "

  inscription $.
  [ self trace: 'inscription terminée' ] .
  (saisie
  [ champ ajoutJournée: stack removeLast ] .
  [ self trace: 'saisie terminée' ] .
  | édition
  [ self trace: 'édition ok' ] .
  ) *
  $.
  édition .
  $.
  [ self trace: 'édition finale ok' ] .!
```

Les messages `self trace:` impriment sur la console des informations sur l'évolution de l'analyse. Ces traces prennent effet seulement si la variable de classe `IsTraceOn` est positionnée à `true`.

Inscription

L'inscription enregistre les caractéristiques des équipes. On précise la division, puis on saisit les équipes.

```

inscription =
  " inscription des équipes du championnat"

  [self traceIn: 'inscription'].
  #Division
  word [: asSymbol]
  $.
  [champ ←Championnat newDivision: stack last].
  (equipe \$, )
  [pasDeDoublesAssoc:] [equipes:]
  [champ inscriptionEquipes: stack removeLast]
  [:->]
  [self traceOut: 'fin des inscriptions'] !

```

Les caractères sont notés par `$c` tandis que les mots-clés sont définis par des symboles `#mot`. Le non-terminal `word` signifie qu'un mot est reconnu. Il s'agit en fait d'un token prédéfini de l'analyseur lexical. Les mots sont séparés par des séparateurs (blancs, des tabulations ou des passages à la ligne, etc.). La classe associée à l'objet reconnu est `String`. L'action `[: asSymbol]` convertit le sommet de pile en symbole, en lui envoyant le message unaire `asSymbol`.

L'action `[champ := Championnat newDivision: stack last]` crée un objet de classe `Championnat` et le place en sommet de pile.

L'expression `(equipe $,)` évalue au moins une fois le non terminal `equipe`, c'est l'opérateur `+` des expressions régulières. Chaque saisie d'équipe est séparée par le caractère `,`. Le résultat est un tableau d'équipes. L'action `[pasDeDoublesAssoc:]` vérifie les doublons dans ce tableau. L'action `[equipes:]` crée un ensemble d'objets de la classe `Equipe`. Ces équipes sont ensuite ajoutées au championnat.

```

equipe =
  " inscription des équipes du championnat"

```

```

word word [:->!

```

Une équipe est définie par un nom et un sigle. L'action `[:->]` crée et génère une instance de la classe `Association` (un couple) en prenant dans l'ordre l'objet au sommet de pile -1 puis l'objet au sommet de pile.

Saisie d'un match

La saisie d'un match est la saisie de l'équipe qui reçoit, celle qui visite, les scores des équipes, l'arbitre, le commentaire et la saisie des buts. Les vérifications et la création d'objets de la classe `MatchF` se font dans une méthode de la classe `FootComplier` appelée `receveur: visiteur: scoreR: scoreV: arbitre: com: buts:`.

```

saisieMatch =
  " saisie d'un match d'une journée du championnat"

  [self traceIn: 'saisie d'un match journee'].
  equipe [equipeExiste:]
  equipe [equipeExiste:]
  number [: asInteger]
  number [: asInteger]
  word
  commentaire
  (((saisieBut) \$, ) [: asSet] | [Set new])
  [receveur: visiteur: scoreR: scoreV: arbitre: com: buts:]
  [self traceOut: 'fin de la saisie d'un match'] ! !

```

Evaluation d'un exemple

```

entrée :
Division D1. Bordeaux Girondins, Nantes FC , Paris FC , Marseille Olympique.
Journée numéro 5 du : 08 03 2001
Bordeaux Girondins Nantes FC 0 1 Alain 'Bon match'
michel albert Nantes FC 12 ;
Paris FC Marseille Olympique 0 3 Ali 'Bon match'
denis lon Marseille Olympique 25 ,
denis lon Marseille Olympique 35 ,
my lon Paris FC 72 csc
editer
Journée numéro 6 du : 09 03 2001
Bordeaux Girondins Paris FC 0 1 Aleau 'Match équilibré'
my lon Paris FC 59 ;
Nantes FC Marseille Olympique 2 1 Ali 'Bon match'
denis lon Marseille Olympique 25 ,
denis lon Marseille Olympique 35 csc,
michel albert Nantes FC 76
Journée numéro 7 du : 09 03 2001
Bordeaux Girondins Marseille Olympique 1 1 Aleau 'Match dur'
denis large Marseille Olympique 65 ,
bon vin Bordeaux Girondins 69 ;
Nantes FC Paris FC 1 1 Alain 'Match moyen'
my lon Paris FC 59 ,
michel lucien Nantes FC 70
. editer.
résultat :
----
Championnat de division : D1
Journées effectuées
Journée 5 du : 8/3/2001
Paris FC 0 - 3 Marseille Olympique
Match arbitré par Ali
but à la 25 minute marqué par denis lon ( Marseille Olympique )
but à la 35 minute marqué par denis lon ( Marseille Olympique )
but à la 72 minute marqué par my lon ( Paris FC ) contre son camp
Commentaire : Bon match
Bordeaux Girondins 0 - 1 Nantes FC
Match arbitré par Alain
but à la 12 minute marqué par michel albert ( Nantes FC )
Commentaire : Bon match
Journée 6 du : 9/3/2001
Bordeaux Girondins 0 - 1 Paris FC
Match arbitré par Aleau
but à la 59 minute marqué par my lon ( Paris FC )
Commentaire : Match équilibré
Nantes FC 2 - 1 Marseille Olympique
Match arbitré par Ali
but à la 25 minute marqué par denis lon ( Marseille Olympique )
but à la 35 minute marqué par denis lon ( Marseille Olympique ) contre son camp

```

but à la 76 minute marqué par michel albert (Nantes FC)
 Commentaire : Bon match
 Journée 7 du : 9/3/2001
 Bordeaux Girondins 1 - 1 Marseille Olympique
 Match arbitré par Aleau
 but à la 65 minute marqué par denis large (Marseille Olympique)
 but à la 69 minute marqué par bon vin (Bordeaux Girondins)
 Commentaire : Match dur
 Nantes FC 1 - 1 Paris FC
 Match arbitré par Alain
 but à la 70 minute marqué par michel lucien (Nantes FC)
 but à la 59 minute marqué par my lon (Paris FC)
 Commentaire : Match moyen
 Classement des équipes : Pts Jo G N P BP BC
 Nantes FC 7 3 2 1 0 4 2
 Marseille Olympique 4 3 1 1 1 5 3
 Paris FC 4 3 1 1 1 2 4
 Bordeaux Girondins 1 3 0 1 2 1 3
 Classement des buteurs
 denis lon (Marseille Olympique) : 3
 michel albert (Nantes FC) : 2
 my lon (Paris FC) : 2
 denis large (Marseille Olympique) : 1
 michel lucien (Nantes FC) : 1
 bon vin (Bordeaux Girondins) : 1

Figure 59 : *Evaluation d'une expression par l'analyseur FootCompiler*

Table des figures

1	Relation d'instanciation du point p1	10
2	Un exemple de copie en Smalltalk	13
3	La super-méthode en Smalltalk	14
4	La classe MétaClasse	17
5	Classe PluvioSimple	23
6	Conception objet du programme pluvio	30
7	Héritage des fiches	31
8	Environnement initial	50
9	La souris Smalltalk-80	51
10	Fenêtre de navigation de ressources	53
11	Une fenêtre de dialogue texte	54
12	Un espace de travail	55
13	Une fenêtre de dialogue de type liste	56
14	Une fenêtre flâneur système	57
15	Une fenêtre inspecteur	59
16	Une fenêtre signalement d'erreur	60
17	Une fenêtre débogueur	60
18	Environnement initial	64
19	Création de la catégorie, jeu de Nim	68
20	Une fenêtre flâneur système	70
21	Implantation Smalltalk de la classe Joueur	72
22	Implantation Smalltalk de la classe JoueurIntelligent	72
23	Implantation Smalltalk de la classe Arbitre	74
24	Version schématique du modèle MVC	77
25	Le modèle MVC avec les dépendants	78
26	Architecture générale des composants visuels selon le MVC	80
27	Un exemple de MVC : l'éditeur de trajets	81
28	Un exemple d'architecture MVC : l'éditeur de trajets	81
29	Un exemple de MVC : le compteur	82
30	Un exemple d'architecture MVC : le compteur	82
31	Un exemple de MVC : les histogrammes	83
32	Un exemple d'architecture MVC : les histogrammes	83
33	Un exemple d'architecture générale d'application selon le MVC	84
34	La hiérarchie des <i>widgets</i> sous <i>VisualWorks</i> version 1	87
35	La hiérarchie des <i>widgets</i> sous <i>VisualWorks</i> version 2	88
36	Un exemple d'architecture MVC VisualWorks	89
37	Diagramme des objets du modèle d'application	91
38	Hiérarchie des fenêtres	92

39	Un exemple de MVC sous VisualWorks 1.0 : la calculette	93
40	Un exemple d'architecture MVC VisualWorks : la calculette 1/2	93
41	Un exemple d'architecture MVC VisualWorks : la calculette 2/2	94
42	Un exemple de MVC sous VisualWorks 2.5 : la calculette	94
43	Les outils de construction d'interface	96
44	Les outils de construction d'interface	96
45	Exemple guidé : le canevas de l'additionneur	97
46	Exemple guidé : les propriétés du bouton +	98
47	Exemple guidé : les propriétés de l'opérande 1	99
48	Exemple guidé : les propriétés de la fenêtre	100
49	Exemple guidé : les fenêtres d'installation	100
50	Exemple guidé : l'éditeur de menus	101
51	Exemple guidé : le dialogue de validation	102
52	Exemple guidé : l'installation des ressources	103
53	Exemple guidé : l'éditeur de menus(suite)	104
54	Exemple guidé : le canevas achevé de l'additionneur	105
55	Programme impératif du match de tennis	126
56	Conception objet du programme <code>tennis</code>	128
57	Conception objet du programme <code>tennis en double</code>	141
58	Conception objet du programme <code>tennis en double</code>	141
59	Evaluation d'une expression par l'analyseur <code>FootCompiler</code>	146