Construire des Applications interactives en Smalltalk-80

Bernard Pottier

Janvier 2012

Sommaire

Programmation d'interfaces spécifiques

Les applications interactives peuvent aujourd'hui se diviser en deux classes : les applications purement locales et celles dont le centre de décision se trouve sur un serveur distant.

Dans le premier cas la rapidité des machines permet d'obtenir des réactions instantanées aux opérations de l'utilisateur. Dans le second cas, les délais de transmission sur le réseau introduisent des latences sensibles. On peut cependant réduire les sensations d'attente en téléchargeant du code sur le poste client, et en réduisant les opérations de transferts avec le serveur.

Ce sont les applications interactives locales qui nous intéressent ici. Ce livre propose d'apprendre en trois temps comment réaliser des programmes interactifs non triviaux d'un point de vue graphique et algorithmique.

La première partie donne, ou redonne, les bases de la programmation en Smalltalk-80. Elle comporte un chapitre d'apprentissage de la syntaxe du langage, et un second chapitre présentant l'outillage algorithmique mis à disposition des programmeurs au travers des collections, et des streams.

La seconde partie présente d'abord les outils de production d'interfaces de la plateforme Visualworks [1], connus sous le nom de *Canvas*, puis de *Painter*. Ces outils permettent de construire des interfaces très rapidement, tout en restant extensibles et ouverts. À coté des interfaces standards, il est courant d'avoir à réaliser des interfaces spécifiques, présentant de manière Visuelle les détails d'un Modèle de données, que l'utilisateur pourra Contrôler. Cette organisation est connu sous le nom de MVC pour Model-View-Controller. Le propos central que s'est fixé cet ouvrage est d'amener à réaliser des interfaces locaux spécifiques s'appuyant sur des modèles algorithmiques plutôt que transactionnels.

La troisième partie décrit des outils techniques que nous estimons indispensables pour une programmation efficace et professionnelle. Il s'agit des outils de mise au point (debugger), ici particulièrement efficaces et attractifs, d'observation dynamique du comportement des programmes (profilers), de gestion de version et de partage des sources, et enfin des outils de déploiement.

L'ouvrage se base sur de nombreux exemples qui sont distribués en paquetages accessibles sur l'Internet. Il sera utile à des étudiants ou ingénieurs souhaitant découvrir un langage de programmation objet puissant et professionnel, Smalltalk-80, et cela avec un point de vue graphique. Les applications possibles sont nombreuses dans le domaine industriel, qu'il s'agisse

1. http ://cincommalltalk.com

d'applications embarquées, ou du développement d'outils de domaine.

Première partie

Programmation

Chapitre 1

Classes : rappels, et pratique

1.1 Rappels : role des classes

Les classes servent à décrire le comportement des *objets* en les *groupant* selon leurs propriétés (Integer, Boolean, LineSegment ...). La *terminologie objet* met en avant quelques concepts particuliers à ce type de programmation.

Classe

Une classe, abstraitement, c'est un ensemble d'objets qui ont les mêmes propriétés, les mêmes structures de données internes, opérables via les mêmes procédures. Concrètement, les classes sont associées à des outils qui permettent de décrire comment les objets d'une classe sont constitués, et comment on agit sur ces objets. Les *Browsers* et *Inspecteurs* sont de tels objets.

Une classe Boule permettrait de décrire un objet concret sphérique, dont le centre est localisé en un point, et qui est peint d'une certaine couleur.
Chaque Boule aura donc deux variables, son *centre*, et sa *couleur*.

Instance

Un objet qui est membre d'une classe "C" est désigné comme étant une *instance de "C"*. Tout objet est membre d'une classe et une seule. On peut en avoir la connaissance en lui adressant le message *class*. On peut visiter

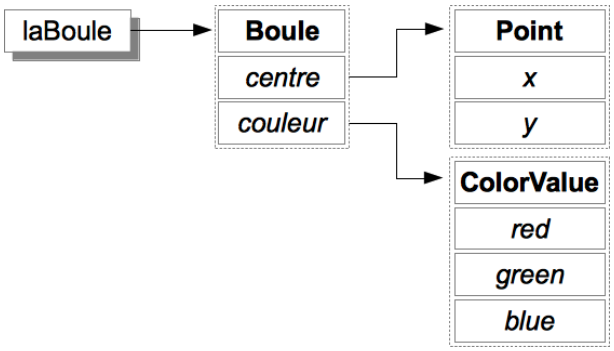


FIGURE 1.1 – Structure de l'objet Boule en mémoire. Chaque objet dispose d'une référence sur la classe à laquelle il appartient.

tout objet en ouvrant un inspecteur avec le message *inspect* (voir en figure).

Une instance de la classe Boule sera désignée par un nom commun *maBoule, uneBoule*. Dans le cas présent, elle présente deux variables d'instance *centre*, et *couleur*. Ces variables sont encapsulées dans un objet "étanche" et ne peuvent être modifiées, lues et opérées que par des messages d'instances définis par la classe Boule, par exemple les accesseurs (ici `centre` et `centre:`).
maBoule centre : 10@10.
sonCentre := maBoule centre.

Message

C'est une activation d'exécution associée à un symbole que l'on désigne par le nom *sélecteur*. L'activation est opérée en recherchant une procédure dont le nom est celui du message. "Expédier un message à un objet" revient donc à demander l'exécution de cette procédure. Un gain important de la programmation objet est que le message est associé à une action ayant une signification abstraite bien comprise.

Toutes les classes géométriques (sous classes de Geometry) comprennent des messages communs tels que `scaledBy: dim` et `translatedBy: dir`. C'est le cas de Circle, qui peut aussi servir à représenter une boule grâce au message `displayOn:` .

Pour créer un cercle :
`cercle := Circle center: unPoint radius: rayon.`
Pour créer une boule :
`maBoule := Boule position: unPoint couleur:`
`(ColorValue black).`
Pour bouger une boule :
`maBoule := maBoule translatedBy: 0@20.`
Pour afficher une boule :
`maBoule displayOn: aGraphicContext.`

Le programmeur associe ainsi des opérations à des noms qu'il retient aisément parcequ'ils produisent un effet similaire..

Méthode

C'est le nom que l'on donne aux procédures décrivant la manière de répondre aux messages.

L'activité du programmeur va se concentrer principalement sur la création et la structure des objets, et sur la description de leurs comportements au travers de l'écriture de *méthodes*. Les méthodes concernent :

les instances , car elles décrivent les réactions des objets *instances de la classe* aux messages,

la classe , car elles décrivent la réaction de la Classe, objet unique, aux messages associés.

Dans l'exemple ci-dessus, on distingue deux méthodes de classes et deux méthodes d'instances :

Classe :
```
cercle := Circle center: unPoint radius: rayon.
```
Classe :
```
maBoule := Boule position: unPoint couleur:
(ColorValue black).
```
instances :
```
maBoule := maBoule translatedBy: 0@20.
```
instances :
```
maBoule displayOn: aGraphicContext.
```

La *classe* est aussi un cadre de programmation dans laquelle on décrit la structure des objets, alors que l'*instance* représente les données d'un objet ou d'une action du monde réel.

Il est très important de bien disjoindre ces rôles que l'on repère par un procédé syntaxique. Les noms de classes sont des noms *propres*, que l'on démarre par une majuscule (*Boule*) dénotant l'unicité, alors que les noms d'instances, associés à des variables dans le cas le plus général (*maBoule*), démarrent par une minuscule.

Les classes peuvent être créées à partir d'assistants (figure 1.3), directement dans le Browser (figure 1.4), ou par programme. Lorsque l'on utilise le Browser, la définition présentée est un *message Smalltalk-80* qui est exécuté lorsque l'on *accepte* le code.

1.1.1 Flot méthodologique

La création d'une classe est techniquement une affaire simple que l'on effectue en quelques minutes.

En général, on a intérêt à utiliser les assistants qui font automatiquement un travail complet et cohérent. La figure 1.3 présente un tel assistant. Les points les plus importants sont l'implantation de la nouvelle classe dans l'arbre d'héritage, et la déclaration des variables d'instances, c'est à dire des champs de l'objet. Ici on a décidé que Boule hériterait de Object, et on a déclaré les 3 variables d'instances *position rayon couleur*.

L'exécution de l'assistant provoque la création d'une classe et appelle une consultation du Browser. La figure 1.4 présente cette classe, que l'on trouve

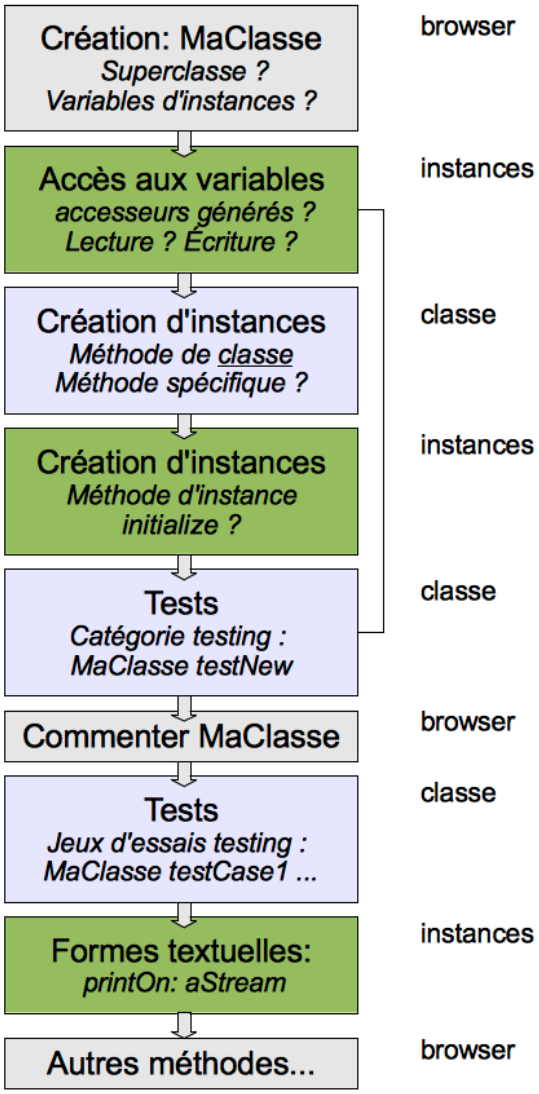


FIGURE 1.2 – Flot initial de conception d'une classe : actions à effectuer, et localisation de l'outil à utiliser.

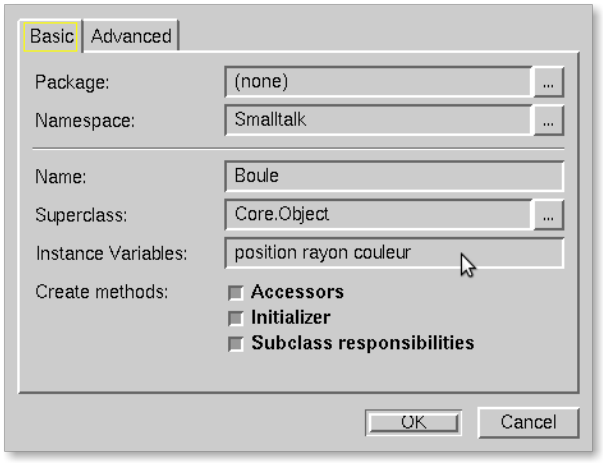


FIGURE 1.3 – Dialogue de création de classe appelé à partir du Browser.

initialement *sans catégorie*, et sans paquetage. Ici, on a modifié le paramêtre catégorie pour référencer *Exemples*.

L'assistant a aussi procédé à d'autres opérations, telles que :

- la création des méthodes d'accès aux variables d'instances, en lecture (par exemple *couleur*), et en écriture (par exemple *couleur : c*),
- la création d'une méthode d'initialisation (*initialize*), sans effet, mais qui permet de donner des valeurs initiales souhaitables aux champs de l'objet. Dans notre cas, on pourrait être intéressé à donner par défaut une couleur noire, et une position 0@0.
- une méthode de création d'instance *new*, qui peut également être retouchée si cela est nécesssaire.

1.2 Méthodes de création d'instances

On peut s'intéresser par exemple à : `Boule position: unPoint couleur: color rayon: r`

Ces méthodes sont décrites en tant que méthodes de classe, catégorie *instance creation*. En général elles se décomposent de la manière suivante :

1. choix d'un sélecteur expressif, exemple : `position: point couleur: colorValue`
2. création d'un nouvel objet `maBoule` avec la méthode de classe `new`, ou `basicNew`.

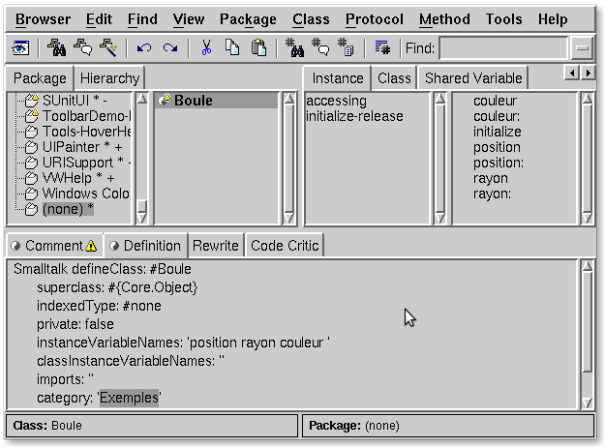


FIGURE 1.4 – Allure du Browser après création de la classe Boule.

```
maBoule := self basicNew.
```

3. initialisation : `maBoule initialize`
   Ceci suppose que la méthode d'instance *initialize* a été créée
4. chargement des variables d'instances :

   ```
   maBoule position: point; couleur: colorValue
   ```

5. renvoi de la nouvelle instance : ↑ `maBoule`

```
position: point couleur: colorValue
"Boule position: 20@20 couleur: ColorValue green"
maBoule := self basicNew.
maBoule initialize.
maBoule position: point; couleur: colorValue; rayon: 10.
^ maBoule
```

1.3 Test

Le test doit être mené systématiquement, dès qu'un petit nombre de méthodes ont été développées ou modifiées. La figure 3 montre une méthode de classe permettant de valider la création d'instance en ouvrant un inspecteur sur un nouvel objet créé.

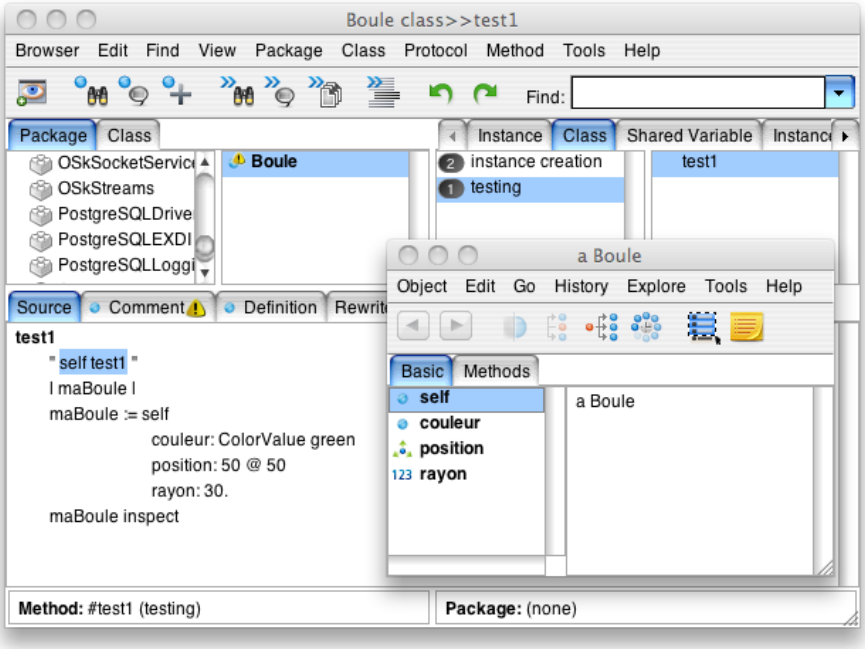


FIGURE 1.5 – Méthode de test classe Boule, et inspecteur sur une instance.

1.4 Formes textuelles

Une des premières actions du programmeur peut être de définir la forme textuelle associée à l'objet, en décrivant une méthode d'instances `printOn:`, dans une catégorie *printing*. Cette méthode est appelée par des messages tels que `printString` et il est souvent commode d'en disposer.

Par exemple, ici, il pourrait s'agir de la méthode suivante :

```
printOn: aStream
   " maBoule printString "
  aStream nextPutAll: '(Boule '.
  self couleur printOn: aStream.
  aStream space.
  self position printOn: aStream.
  aStream space.
  self rayon printOn: aStream.
  aStream nextPut: $)
```

Un peu plus tard, on s'intéressera à l'écriture d'une méthode de classe (*from : aStream*), permettant de regénérer une Boule à partir de la forme textuelle choisie.

1.5 Source de la version de Boule produite par l'assistant de création

Le code qui suit présente la classe Boule, telle qu'elle a été générée à partir de l'assistant (figure 1.4). Ce texte a été obtenu en opérant un *File out* sur la classe Boule, sélectionnée dans le Browser, en produisant ainsi un fichier que l'on nomme *Boule.st*. Le format présenté ici est dénommé 'source chunks', il est accessible en tant qu'option de présentation des sources dans les *Settings* du Transcript en alternative à XML.

Ce fichier peut être relu dans une autre image Smalltalk-80, à partir de sa sélection dans un brosseur de fichiers, et en opérant cette fois un *File in*.

Note concernant self : les sources produits, et ceux que l'on programme dans les classes référencent le receveur du message activé par une pseudo-variable *self*.

```
Smalltalk defineClass: #Boule
        superclass: #{Core.Object}
```

```
        indexedType: #none
        private: false
        instanceVariableNames: 'position rayon couleur '
        classInstanceVariableNames: ''
        imports: ''
        category: 'Exemples'!

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!


!Boule class methodsFor: 'instance creation'!

new
        "Answer a newly created and initialized instance."

        ^super new initialize! !

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!


!Boule methodsFor: 'accessing'!

couleur
        ^couleur!

couleur: anObject
        couleur := anObject!

position
        ^position!

position: anObject
        position := anObject!

rayon
        ^rayon!

rayon: anObject
        rayon := anObject! !
```

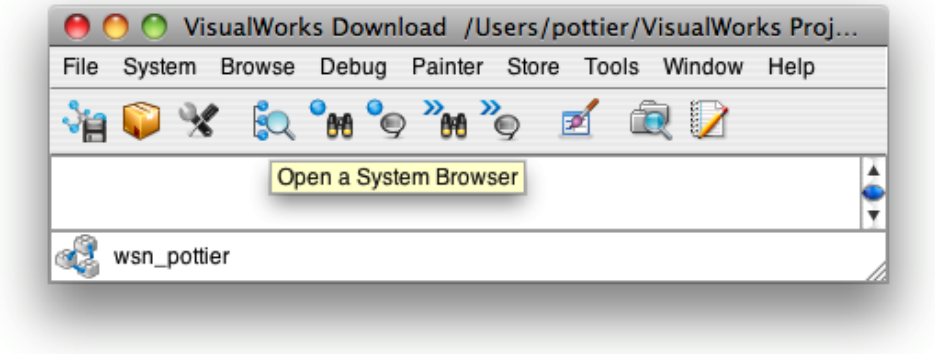


FIGURE 1.6 – Icone de lancement du Browser dans le cockpit Visualworks

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

initialize
        "Initialize a newly created instance. This method must answer the recei

        " *** Edit the following to properly initialize instance variables ***'
        position := nil.
        rayon := nil.
        couleur := nil.
        " *** And replace this comment with additional initialization code ***
        ^self! !
```

1.6 Exercice : créer une classe Boule

Cet exercice propose de produire une classe complète en apprenant à manipuler les outils de développement. Commencer par se familiariser avec le Browser (figure 1.6) et l'assistant de création de classe (figure 1.7).

1.6.1 Création d'une classe

1. Construire la classe Boule décrite en section 1.1.1, vérifier sa présence dans le Browser. Vérifier la présence des accesseurs.
   Utiliser new pour créer une première instance et l'inspecter :

   ```
   | maBoule|
   ```

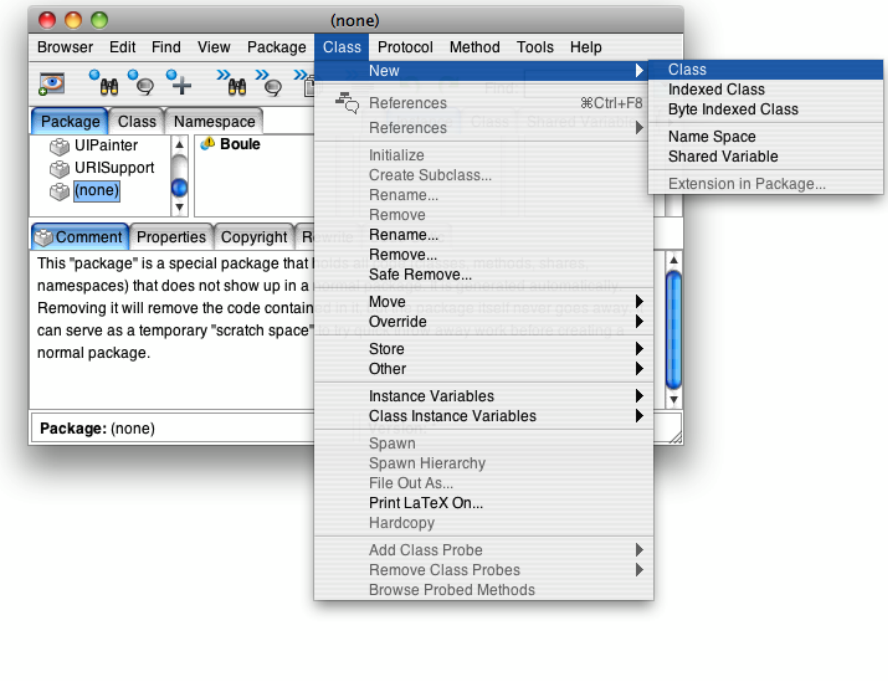


FIGURE 1.7 – Lancement de l'assistant Creation dans le browser

```
maBoule := Boule new.
maBoule inspect
```

2. Ecrire une méthode de classe permettant de produire une nouvelle Boule, avec ses 3 variables d'instance initialisées :

   `position: point couleur: uneCouleur rayon: r`

3. Ecrire une méthode de classe `test1` permettant de tester la méthode précédente. Voir figure

1.6.2 Sauvegarde et restauration d'une classe

1. On peut sauvegarder une classe en effectuant un *File out* sur un fichier de texte (figure 1.8). Effectuer un essai sur votre version de la classe Boule et un nom de fichier tel que Boule.st
2. Pour restaurer la classe, il faut utiliser un brosseur de fichier (icone double dossier avec la loupe dans cette version), pointer sur votre fichier, et appeler *File in* dans le menu (figure 1.9).

1.6.3 Gestion d'une forme textuelle

Implémenter `printOn:` dans les méthodes d'instances, et une méthode de test associée. On se reportera à la section 1.4.

1.6.4 Méthode de tests et génération aléatoire

1. Ecrire une méthode `test1NbBoules: nbBoules inRect: rectangle`. Cette méthode génère un système de boules colorées installées à des positions aléatoires dans le rectangle fourni en paramêtre.
2. Ecrire une méthode `test2NbBoules: nbBoules inRect: rectangle`. Cette méthode génère un système de boules colorées installées à des positions aléatoires dans le rectangle fourni en paramêtre. Le contour des boules doit tenir dans ce rectangle et l'algorithme doit finir..

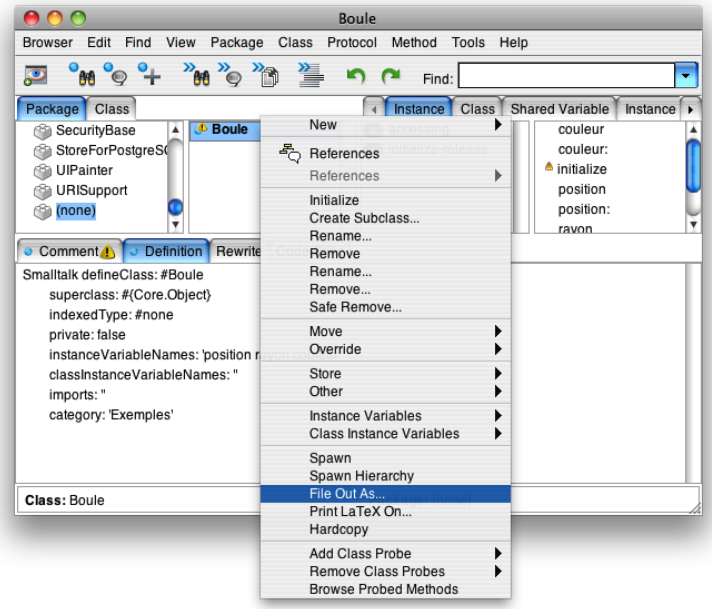


FIGURE 1.8 – Sauvegarde d'une classe dans un fichier de texte .st : on appelle la fonction *File out as* du menu dans la sous-fenêtre des classes, des catégories, ou des méthodes.

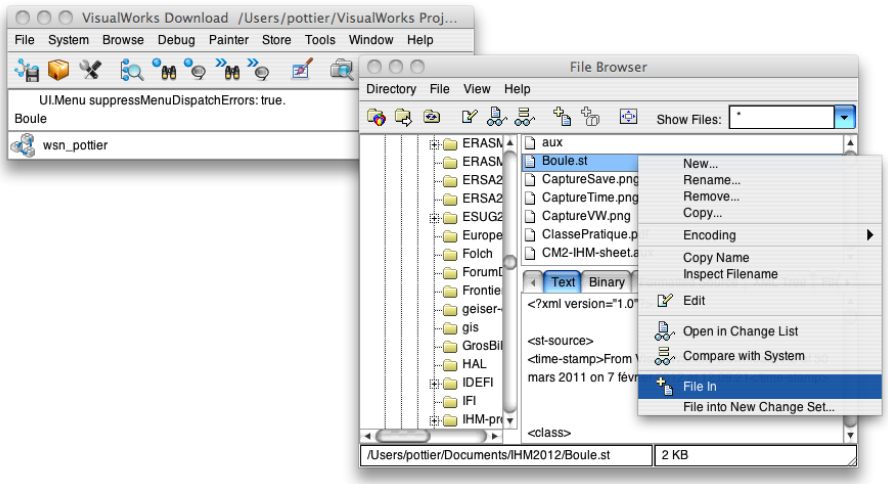


FIGURE 1.9 – Restauration d'une classe à partir d'un fichier de texte .st : on sélectionne le fichier source et on appelle la fonction *File in*

Chapitre 2

Les collections

2.1 Présentation

Les collections sont une famille de classes fournissant un support pour la gestion des données et l'algorithmique de base.

La classe Collection est la racine de l'arborescence conduisant aux différentes classes. Elle n'a pas directement d'instances, c'est une classe "abstraite", mais elle occupe une position logique et propose un jeu de messages que l'on retrouvera dans les sous-classes.

Ce document présente d'abord les niveaux et les classes de la hiérarchie, et les messages les plus significatifs acceptés ou refusés par les classes. Dans un second temps on trouve une description des messages les plus courants accompagnés d'exemples illustrant les principaux usages que l'on peut en faire.

2.1.1 Classification des collections

Chaque noeud dans l'arbre correspond à une, ou un petit nombre de propriétés et opère une disjonction en sous-arbres en fonction de cette propriété. Par exemple (section 2.2.1), la classe Collection amène à disjoindre les classes ne disposant pas d'ordre (Bag, Set) des classes qui en disposent (SequenceableCollection).

2.1.2 Hiérarchie simplifiée

La hiérarchie des collections est accessible dans le brosseur Smalltalk en sélectionnant la classe Collection, et dans le menu de la deuxième sous-fenêtre l'option `hierarchy`. Il faut noter que certaines classes d'applications

apparaissent lors de cette présentation .

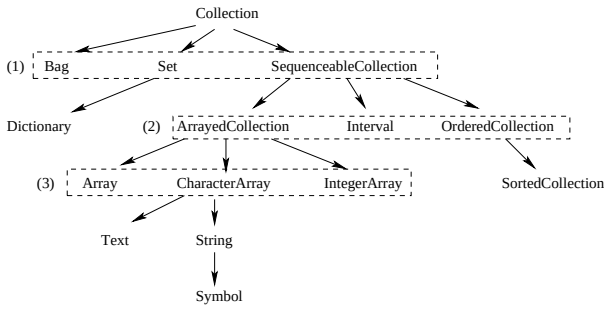


FIGURE 2.1 – Hiérarchie des classes

2.1.3 Méthode de programmation

Le code Smalltalk-80 est le plus dense parmi ceux des langages de programmation courants. Cette densité est liée à la réutilisation de mécanismes ouverts, fiables et bien organisés. C'est le cas des collections.

Ecrire moins de code c'est gagner sur deux plans : la productivité, et la fiabilité.

langage	ligne de code par fonction
Smalltalk	21
Ada 95	49
Java	53
C++	53
COBOL	107
C	128

TABLE 2.1 – Concision comparée de quelques langages selon une source industrielle

Dans le cas des collections, on peut effectuer les recommandations suivantes :

1. Prise en main du cadre de programmation :
   (a) Apprendre les caractéristiques des classes majeures dans la hiérarchie (propriétés caractéristiques au sens de la classification, et principaux messages).

(b) Apprendre les messages principaux.

2. Développer des réflexes de programmation par rapport à ce cadre :

   (a) Identification du besoin, ou de la succession de besoins à un stade du problème en cours d'analyse.

   (b) Identification des classes et méthodes permettant de recouper ce besoin, et implémentation via les messages et les conversions de classes.

Illustration

Dans les exemples qui suivent on montre deux enchainements conduisant à la solution du problème posé. Ces enchainements sont basés sur des successions de propriétés trouvées pour aboutir à la propriété réalisant la question posée.

Exemple 1 *Soit la chaine s :='une chaine de caractère'.*

Compter le nombre de voyelles.

Compter les mots (séparés par des blancs).

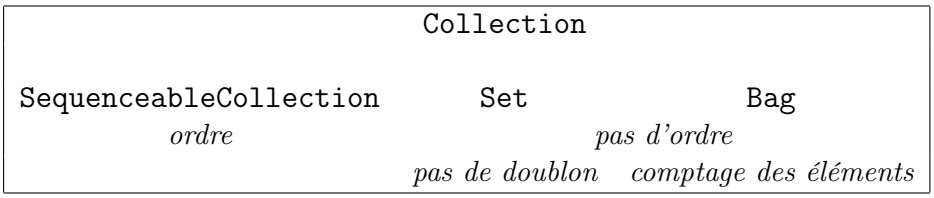
```
| s |
s:='une chaine de caractère'.
(s select: [ :char | char isVowel] ) asSet size  " 5"
"Algo: Extraire les voyelles, enlever les doublons, compter"

| s mots |
s:='une chaine de caractère'.
mots := ( s tokensBasedOn: $ ) reject: [:mot | mot isEmpty].
mots size "4"
"Algo: Casser en mots, enlever les mots vides, compter"
```

2.2 Classification par propriétés

2.2.1 Collections avec ou sans ordre

Le premier exemple de disjonction se trouve au niveau 2 de la hiérarchie où l'on disjoint les sous-classes de Collection disposant d'un ordre (SequenceableCollection), des classes sans ordre (Bag et Set) :



Notion d'ordre dans les éléments des collections

- Bag and Set n'ont pas d'ordre, et leurs éléments ne peuvent pas être désignés par une clé externe.
- Les doublons ne sont pas possibles dans les ensembles (Set).
- Les éléments des SequenceableCollection sont ordonnés, soit par un ordre externe comme dans le cas des ArrayedCollection, soit par un mécanisme interne comme dans le cas des Interval.

- Les messages requis au niveau de la classe Collection sont listés si-dessous. Attention, certains de ces messages sont parfois interdits dans quelques sous-classes (`add:` dans Array), ou redéfinis plus efficacement (`add:` dans les Set).
- D'autres messages sont reconstitués en utilisant les messages requis, comme c'est le cas pour `collect:` en utilisant `do:`.

catégorie	C/i	messages
instance creation	Classe	new new :
accessing	instance	size
testing	instance	includes :
adding	instance	add : addAll : addFirst : addLast :
removing	instance	remove : removeFirst removeLast
enumerating	instance	do :
converting	instance	asSet asBag asArray asOrderedCollection asSortedCollection

TABLE 2.2 – Messages requis des sous-classes au niveau de Collection

catégorie	C/i	messages
instance creation	Classe	with : with :with : with :with :with : withAll :
testing	instance	isEmpty occurencesOf :
adding	instance	add : addAll : addFirst : addLast :
removing	instance	remove : removeFirst removeLast
enumerating	instance	collect : detect : select : reject : inject :into :

TABLE 2.3 – Quelques autres messages disponibles au niveau de Collection

2.2.2 Collections sans ordre : les ensembles et les sacs

Set

est une collection dynamique dont l'interface est basée sur l'ajout et la suppression. Au moment de l'ajout, on vérifie si l'élément existe ou pas afin de ne pas provoquer de doublons.

– Set permet un certain nombre d'opérations ensemblistes directement (soustraction -), ou via l'interface des collections (addAll : ...).

catégorie	C/i	messages
testing	instance	includes : occurencesOf :
adding	instance	add :
removing	instance	remove :ifAbsent : - (soustraction d'ensembles)

TABLE 2.4 – Messages associés à Set

Bag

a un interface similaire à Set. Au moment de l'ajout, on incrémente un compteur associé à chaque élément du sac. Si on ajoute un très grand nombre d'éléments identiques à un Bag, celui-ci reste petit. D'autre-part le comptage est immédiat.

– Bag dispose de messages particuliers permettant la gestion du comptage, y compris au moment de l'itération (valuesAndCountsDo : ...).

catégorie	C/i	messages
testing	instance	includes : occurencesOf :
adding	instance	add : add :withOccurrences :
removing	instance	remove :ifAbsent : removeAllOccurrencesOf :ifAbsent :
enumerating	instance	do : valuesAndCountsDo :

TABLE 2.5 – Messages associés à Set

2.2.3 Les dictionnaires

Dictionary

est une classe héritant de Set. Les éléments des dictionnaires sont des associations couplant une clé et une valeur. L'unicité de l'association est assurée au niveau de la clé : on ne peut pas ranger deux éléments différents

La classe Association

Pour créer une association on utilise en règle générale le message binaire $->$. Le receveur est une clé qui fixe le "nom" de l'association, et l'argument est la valeur.

```
| asso |
   asso := #Nono->12.
   asso key. "#Nono"
   asso value.  "12"
```

La clé n'est pas nécessairement un symbole. On peut utiliser n'importe quel objet, par exemple des nombres :

```
| asso1 asso2 |
   asso1 := 4->12.
   asso2 := 12->18.
```

```
   asso1 key = asso2 key.  "false"
   asso1 value = asso2 value. "false"
```

Dictionary : création et insertions

Une fois le dictionnaire créé, on peut insérer des éléments avec le message `at:put:`. Alternativement, on peut aussi ajouter des associations. Pour lire une valeur, on adresse le dictionnaire avec le message `at:` .

```
| dictio   |
   dictio := Dictionary new.
   dictio add:  4->12.
   dictio. " Dictionary (4->12 )"
   dictio at: 12 put: 18.
   dictio.  "Dictionary (4->12 12->18 )"
   dictio at: 4. "12"
```

Itérations sur les dictionnaires

Pour itérer sur un dictionnaire, on a plusieurs choix :
- `keysAndValuesDo: [ :key :value | ]` : permet de disposer à la fois de la clé et la valeur. Voir l'exemple donné en 7.
- `associationsDo: [ :asso | ]` : meme possibilité si on veut bien extraire la clé et la valeur de l'association.
- `keysDo: [ :key | ]` : permet de balayer l'ensemble des clés.
- `collect: [ :valeur | ]` : les messages collect : ...portent sur les valeurs et non les associations ou les clés.

Accès aux dictionnaires

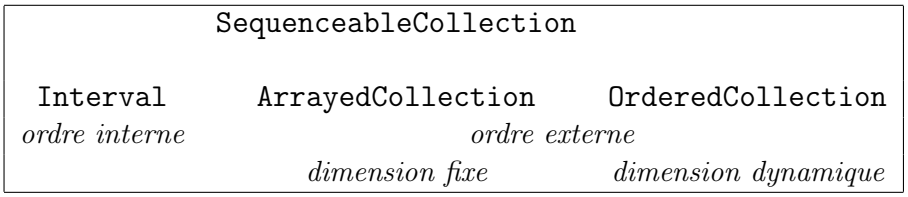
Pour extraire les clés, on adresse le message `keys` au dictionnaire. Le résultat est un Set. Pour extraire les valeurs, on adresse le message `values`. Le résultat est un Bag. On peut ajouter des associations, on ne peut pas enlever d'éléments.

2.2.4 Collections séquençables, avec ordre interne ou externe

Le second cas de disjonction se trouve au niveau 3 de la hiérarchie où l'on sépare les sous-classes de SequenceableCollection selon que l'ordre soit une notion interne (Interval) ou externe (ArrayedCollection, OrderedCollection).

catégorie	C/i	messages
accessing	instance	at : at :ifABsent : at :put : associationAt : keyAtValue : keys values
testing	instance	includes : occurencesOf :
adding	instance	add : declare :From :
enumerating	instance	collect : keysAndValuesDo : associationsDo : keysDo :
removing	instance	remove :ifAbsent : remove : (interdit !)
	instance	removeKey :ifAbsent : removeKey :

TABLE 2.6 – Principaux messages associés à Dictionary



Qualité interne ou externe de l'ordre

- Interval dispose d'un ordre interne permettant le séquencement à partir d'une valeur de départ (`start`), jusqu'à une valeur finale (`stop`), en utilisant un pas (`step`)
- ArrayedCollection et OrderedCollection rangent les éléments en fonction des opérations d'accès (`at: at:put:`) ou d'ajouts (`add: remove:`).

- Toutes ces classes sont indexables par des entiers à l'aide des messages `at:` et `at:put:` (sauf Interval).
- Elles permettent la recherche d'un élément selon leur ordre (`findFirst:`).
- Elles autorisent des copies de sous-ensembles.
- Elles permettent la création de Stream en lectures ou écritures, et des opérations plus complexes telles que l'éclatement basé sur un élément

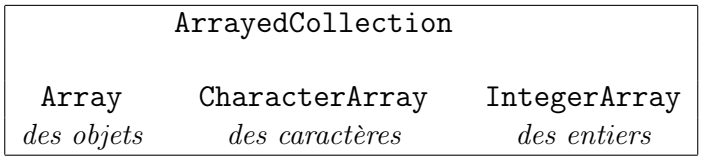
particulier (`tokensBasedOn:`).

catégorie	C/i	messages
adding	instance	grow
adding/removing	instance	add : remove : interdits sauf OrderedCollection
converting	instance	readStream writeStream asArray tokensBasedOn :
enumerating	instance	keysAndValuesDo :

TABLE 2.7 – Messages ajoutés et enlevés dans les sous-classes au niveau de SequenceableCollection

2.2.5 La hiérarchie des tableaux

ArrayedCollection est le chapeau de la hiérarchie des tableaux, collections de dimension fixe et indexables par des entiers. Ses deux classes les plus importantes sont Array et CharacterArray. La classification sous-jacente repose sur les classes d'objets enregistrés.



- Ces classes servent de support à des fonctionalités pour des classes particulières d'éléments.
- CharacterArray est la racine des classes gérant une variétés de chaines de caractères avec des fonctionalités particulières (String, Symbol, Text).
- D'autres classes (ByteArray) servent de support aux interfaces avec le matériel : réseau, visualisation...

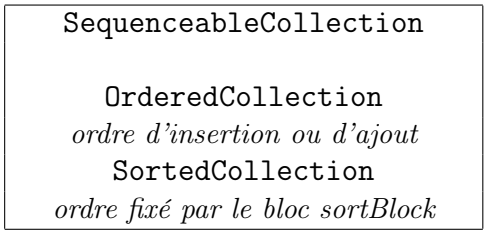
2.2.6 Les collections ordonnées

OrderedCollection est une collection disposant de l'indexation et de la dynamicité de l'espace de rangement (`add: remove:`). SortedCollection est

catégorie	C/i	messages
accessing	instance	findString :startingAt :
converting	instance	asFilename asNumber asLowercase asSymbol asText
comparing	instance	=<=>= match :

TABLE 2.8 – Messages apportés par CharacterArray

une OrderedCollection où la position d'insertion est commandée par une clé de tri, et non un index donné par l'utilisateur.



- Ces deux collections changent de taille pour s'adapter aux besoins et cela de manière transparente.
- OrderedCollection peut être utilisée pour décrire des piles, ou des queues.

catégorie	C/i	messages
accessing	instance	at : at :put :
adding	instance	add : addFirst : addLast : addAll : add :after : add :before :
removing	instance	remove :ifAbsent : removeFirst removeLast removeAtIndex : removeAllSuchThat :

TABLE 2.9 – Messages associés à OrderedCollection

catégorie	C/i	messages
accessing	instance	at : (at :put : interdit)
adding	instance	add : addAll :
sorting	instance	sortBlock :

TABLE 2.10 – Messages associés à SortedCollection

2.3 Nomenclature des messages

2.3.1 Itération do :

uneCollection do : [:elt |]

Ce message itère sur la collection en appliquant le bloc paramètre à chacun de ses élément. Le résultat est l'objet `uneCollection` *non modifié.*

Exemple 2

```
| max |
    max := SmallInteger minVal.
     #( 3 7 -2) do: [ :i |  max := i max: max ].
    max " 7, le maximum du tableau "
```

Exemple 3

```
| max |
    max := 1.
     (1 to: 200)  do: [ :i |  i printString reverse = i printString ifTrue: [ max:=i] ].
    max " 191,  le plus grand palyndrome de l'intervalle 1,200 "
```

Exemple 4

```
     'verbatim'  do: [ :char |  char isVowel ifTrue:
                            [Transcript show: (String with: char) ; space ]  ].
    "imprime e a i,  les voyelles de la chaine initiale "
```

Exemple 5

```
    | aSet |
    aSet := Set withAll:  'verbatim'.
    aSet  do: [ :char |  Transcript show: (String with: char) ; space   ].
    "imprime v i m a b e r t  les caractères de la chaine initiale "
```

2.3.2 Itération keysAndValuesDo :

uneCollection keysAndValuesDo : [:key :elt |]

Ce message itère sur la collection en appliquant le bloc paramètre à chacun de ses élément associé à sa clé. Le résultat est l'objet `uneCollection` *non modifié. Ce message est effectif sur les collections séquenceables et les dictionnaires.*

Exemple 6

```
| somme |
    somme := 0.
     #( 3 7 -2) keysAndValuesDo: [ :index  :nombre  |
                                 somme :=   somme + (index * nombre) ].
    somme "11 = (1*3) + (7*2) +(3*-2)"
```

Exemple 7

```
| somme dico |
    dico := Dictionary new.
    dico at: 2 put: 10; at: 11 put: -2.
    somme := 0.
     dico keysAndValuesDo: [ :index  :nombre  |  somme :=   somme + (index * nombr
    somme  "-2"
```

Exemple 8

```
| dico |
    dico := Dictionary new.
    dico at: #nono put: 10; at: #nini  put: 5.
    dico keysAndValuesDo: [ :nom  :nombre  |
                  Transcript show: nom asString; tab; show: nombre printString ; c
```

2.3.3 Itération reverseDo :

uneCollection reverseDo : [:elt |]

Ce message est similaire à `do:` *mais opère du dernier élément vers le premier élément. Il est opérationnel sur les collections disposant d'un ordre.*

2.3.4 Itération do : separatedBy :

uneCollection do : [:elt |] separatedBy : [ ]

Ce message itère sur la collection en appliquant le bloc paramètre à chacun de ses élément associé à sa clé. En plus du comportement normal du `do:`*, il permet de provoquer une évaluation entre chaque élément de la collection.*

Exemple 9

```
| chaine |
    chaine := String new.
     #( 1 2 3) do:  [ :nombre  |   chaine := chaine , nombre printString ]
                    separatedBy: [ chaine := chaine, ' et ' ].
    chaine  " '1 et 2 et 3'"
```

2.3.5 Itération collect :

uneCollection collect : [:elt |]

Ce message itère sur la collection en appliquant le bloc paramètre à chacun de ses éléments. Les résultats sont accumulés dans une nouvelle collection d'une classe identique ou proche de la classe initiale (par exemple un Interval rendra un Array). Cette nouvelle collection est rendue en résultat.

Exemple 10

```
| max maxProgres |
    max := SmallInteger minVal.
     maxProgres := #( 3 7 -2) collect: [ :i |  max := i max: max ].
    maxProgres "  #(3 7 7) "
```

Exemple 11

```
 | nombres |
nombres := (17 to: 27 by: 2)  collect: [ :i |  i printString reverse asNumber
nombres  "  #(71 91 12 32 52 72)  Noter que ce n'est plus un intervalle, mais
```

Exemple 12

```
| aSet |
aSet := Set withAll:  (-7 to:   7 by: 2).   " Set (1 3 5 -7 7 -5 -3 -1)"
aSet  collect: [ :i |  i squared ].              "Set (1 9 49 25)"
"Noter que la classe initiale a été conservée"
```

2.3.6 Itération select :

uneCollection select : [:elt |]

Ce message itère sur la collection en appliquant le bloc paramètre à chacun de ses éléments. Les éléments pour lesquels le bloc est vrai sont accumulés dans une nouvelle collection d'une classe identique ou proche de la classe initiale. Cette nouvelle collection est rendue en résultat.

Exemple 13

```
|  entiersImpairs |
    entiersImpairs := #( 3 7 -2 0 11 ) select: [ :i |  i odd ].
   entiersImpairs "  #(3 7 11) "
```

Exemple 14

```
 | nombres |
nombres := (17 to: 27 by: 3)  select: [ :i |  i even ].
nombres  "#(20 26) , noter la mutation Interval vers Array "
```

Exemple 15

```
| aSet genAlea |
genAlea := Random new.
aSet := Set new.   "on génère 20 entiers aléatoires >=0 et <20 "
20 timesRepeat: [ aSet add: ( genAlea next * 20) rounded].
aSet select: [ :i |  i  odd ]  " Set (17 1 19 3 5 7 11) nombres impairs dans le tirage"
```

2.3.7 Itération reject :

uneCollection reject : [:elt |]

Ce message itère sur la collection en appliquant le bloc paramètre à chacun de ses élément. Les éléments sont accumulés dans une nouvelle collection d'une classe identique ou proche de la classe initiale, à l'exclusion de ceux pour lesquels le bloc est vrai. La nouvelle collection est rendue en résultat.

Le fonctionnement est très proche de celui de `select:`.

Exemple 16

```
|  entiersPairs |
     entiersPairs := #( 3 7 -2 0 11 ) reject: [ :i |  i odd ].
    entiersPairs   " #(-2 0)"
```

2.3.8 Test et sélection detect :

uneCollection detect : [:elt |]

Ce message itère sur la collection en appliquant le bloc paramètre à chacun de ses élément. Le premier élément pour lequel le bloc est vrai est rendu en résultat. Si aucun élément n'a la propriété, une erreur est produite. Dans les collections sans ordre, l'élément choisi est imprédictible.

Exemple 17

```
|  unEntierPair |
     unEntierPair := #( 3 7 -2 0 11 ) detect: [ :i |  i even ].
    unEntierPair   " -2"
```

Exemple 18

```
| aSet genAlea impairs |
genAlea := Random new.
impairs := (1 to: 5) collect: [ :tirage |
        aSet := Set new.   "on génère 20 entiers aléatoires >=0 et <20 "
        20 timesRepeat: [ aSet add: ( genAlea next * 20) rounded].
        aSet detect: [ :i |  i  odd ] ].
impairs " #(19 1 3 1 1) sont 5 nombres impairs détectés dans 5 tirages successifs"
```

uneCollection detect : [:elt |] ifNone : [ ]

Ce message itère sur la collection en appliquant le bloc paramètre à chacun de ses élément. Le premier élément pour lequel le bloc est vrai est rendu en résultat. Si aucun élément n'a la propriété, le second bloc est évalué et son résultat produit celui du message `detect: ifAbsent:` .

Exemple 19

```
| aSet genAlea detection |
genAlea := Random new.
detection := (1 to: 10 ) collect: [ :tirage |
        aSet := Set new.
        20 timesRepeat: [ aSet add: ( genAlea next * 20) rounded].
        aSet detect: [ :i |  i  = 13 or: [i= 11]  ]  ifNone: [nil]].
detection        " #(13 11 11 11 13 11 nil 11 13 nil)"
```

2.3.9 Cumuls inject :into :

uneCollection inject : valeurInitiale into : [:valeurCourante :elt |]

Ce message itère sur la collection en disposant d'une valeur courante (`valeurInitiale`*). Pour chaque élement, le bloc rend la nouvelle valeur courante à partir de l'ancienne et de l'élément considéré. Le résultat est la valeur courante.*

Exemple 20

```
|  somme |
     somme := #( 3 7 -2 0 11 ) inject: 0 into:  [ :cumul  :i | cumul + i ].
    somme   19
```

Exemple 21

```
|  lettres |
     lettres := 'verbatim'  inject: true into:  [ :propriete  :char |
                                                          propriete and:[ char isAlphabetic
    lettres   " true"
```

Exemple 22

```
| genAlea tirages aBag |
genAlea := Random new.
tirages := (1 to: 10 ) collect: [ :tirage |
        aBag := Bag new.
        2000 timesRepeat: [ aBag add: ( genAlea next * 20) rounded].
        aBag inject: 0  into: [ :max :i | max max: (aBag occurrencesOf: i) ]].
tirages  " #(115 123 123 110 121 112 121 119 118 112)  sur 10 tirages, la plus
                  grande occurence d'un entier dans le sac "
```

2.3.10 Produit de collections with : do :

collection1 with : collection2 do : [:elt1 :elt2 |]

Ce message apparie 2 à 2 les éléments de 2 collections de même dimension. Le bloc spécifié est appliqué à chaque couple formé. Le résultat est la première collection. Ce message est restreint aux collections séquenceables.

Exemple 23

```
| ligne colonne produit |
ligne := #( -1 0 1).
colonne := #( 2 3 4).
```

```
produit := 0.
ligne with: colonne do:  [ :x :y | produit := produit + ( x * y )].
produit                  " 2"
```

Exemple 24

```
| masque chaine s |
masque := #( true false false true).
chaine := 1234 printString.
s := String new.
chaine with: masque do:  [ :char :bool |  | newChar |
                                          newChar := bool ifTrue: [char] ifFalse: [
s := s , (String with: newChar)].
s asNumber " 1004"
```

Exemple 25

```
| poids points barycentre |
poids := #( 2 1 2).
points := Array with: 1@2 with: 3@4 with: 7@0.
barycentre := 0@0.
poids with: points do:  [ :unPoids :unPoint |
           barycentre := barycentre + ( unPoint * unPoids )].
barycentre / ( poids inject: 0 into: [ :sum :unPoids | sum + unPoids ])
" (19/5)@(8/5)"
```

Exemple 26

```
| matrice vecteur |
 matrice := #(
                  ( 1 1 1 )
                  ( 0 1 1 )
                  ( 0 0 1 )).
vecteur := #( 2  0  -2).
matrice collect: [ :ligne | | r | r:= 0.
       ligne with: vecteur do:  [ :m :x | r := r + ( m * x)]. r].
" #(0 -2 -2)"
```

2.3.11 Eclatement de collections tokensBasedOn :

collection tokensBasedOn : element

Ce message éclate les collections séquenceables en une collection ordonnée de sous-collections sur chaque occurrence d'un élément particulier.

Exemple 27

```
| strings |
strings := 'Vous avez gagné   au grand concours Euro2001'
        tokensBasedOn: Character space.
strings := strings select: [:s| s isEmpty not].
strings  "OrderedCollection ('Vous' 'avez' 'gagné' 'au' 'grand' 'concours' 'Euro2001')"
```

Exemple 28

```
| numbers arrays |
numbers := #(  1 2 6 5 6 6 0 10 6 2 3).
arrays := numbers  tokensBasedOn: 6.
arrays inject: #() into: [ :maxArray :array |
     array size > maxArray size ifTrue: [array] ifFalse: [maxArray]]
" #(1 2)"
```

2.4 Messages simples

2.4.1 Taille, nombre d'éléments

collection size

Nombre d'éléments dans la collection

Exemple 29

```
| set |
set := Set withAll: 'Vous avez gagné   au grand concours Euro2001' .
set size "19"
```

collection capacity

Place disponible dans une collection. Dans le cas des Set, Bag, Dictionary, OrderedCollection, la place disponible est presque toujours supérieure au nombre d'élément. Le programmeur peut controler l'efficacité de ses algorithmes en surveillant la capacité.

Exemple 30

```
| set |
set := Set withAll: 'Vous avez gagné   au grand concours Euro2001' .
set capacity  "67"

| oc |
oc := OrderedCollection withAll: 'Vous avez gagné   au grand concours Euro2001' .
oc capacity   "80"
```

2.4.2 Opérations liées à l'ordre

collection reverse

Dans le cas d'une collection identique, rend une collection de même classe ou similaire dont les éléments sont rangés à l'envers.

Exemple 31

```
 'Vous avez gagné au grand concours Euro2001' reverse
 "'1002oruE sruocnoc dnarg ua   éngag zeva suoV'"

| string strings |
string :=  'Vous avez gagné au grand concours Euro2001'.
strings := (string tokensBasedOn: $ ) reverse.
string := strings inject: '' into: [ :s :mot | s, ' ', mot].
string "' Euro2001 concours grand au gagné avez Vous'"
```

collection first

collection last

Premier élément.

Exemple 32

```
'Vous avez gagné   au grand concours Euro2001' first    "$V "16r0056""

#( (1 0) (0 1)) first  "#(1 0)"
```

2.5 Conversions de collections

Ne pas oublier d'avoir recours aux conversions de collections qui permettent de gagner progressivement des propriétés et de résoudre quantité de problèmes :

`asArray asBag asSet asOrderedCollection asSortedCollection asString` etc...

2.6 Résumé des propriétés d'accès et d'ordre

Remplissez-vous même cette table en annotant les particularités des classes par rapport au message figurant en colonne...Lorsque la classe sait exécuter le message, mettre oui. Noter que le message peut être hérité, ce qui différencie la situation de celle traitée section suivante.

2.7 Exemple appliqué aux collections

Problème

Produire une table croisée de messages sur les collections, que vous choisissez, et des sous-classes de collection implémentant au moins 2 de ces messages. Les sous classes sont présentées dans l'ordre lexicographique, et lorsque le message est implémenté dans la classe, on figure la mention 'oui'.

Ici on montre une solution en LaTeXqui alourdit quelque peu le code au profit de la présentation. Attention : ce n'est pas parce qu'un message est implémenté qu'il peut etre exécuté. Pour écarter un message, on le décrit très souvent en provoquant une erreur (Cf. Dictionary et remove :).

	at :	at :put :	add :	remove :	first
Set					
Bag					
Dictionary					
Array					
OrderedCollection					
SortedCollection					

TABLE 2.11 – table résumant les propriétés d'accès et d'ordre

2.7.1 Proposition de codage

```
    | messages subclasses msgCount hasSelectors stream |
    messages := #(#do: #collect:  #add: #remove: #at:) asArray.
    stream := String new writeStream.
    subclasses := (Collection allSubclasses
                select:
                    [:subclass |
                    msgCount := messages inject: 0 into:
                            [:count :message |
                                   count + (subclass selectors occurrencesOf: mess
                    msgCount >= 2]) asArray.
    stream nextPutAll: '\begin{table}[htb]
\begin{center}
\begin{tabular}{'.

    (1 to: messages size + 2)
        do: [:elt | stream nextPutAll: '|']
        separatedBy: [stream nextPutAll: 'c'].
    stream nextPutAll: '}\hline'.
    stream nextPutAll: '
```

classe/message	do :	collect :	add :	remove :	at :
ArborAbstractTree	oui		oui		
Bag	oui		oui		
Dictionary	oui	oui	oui	oui	oui
IdentityDictionary	oui		oui		
Interval	oui	oui	oui	oui	oui
KeyedCollection	oui				oui
LinkedList	oui		oui		oui
LinkedOrderedCollection	oui	oui			
List	oui		oui		oui
OrderedCollection	oui	oui	oui		oui
OrderedSet	oui	oui	oui		oui
Palette			oui		oui
SequenceableCollection	oui	oui			
Set	oui	oui	oui		oui
SortedCollection		oui	oui		
TableAdaptor			oui		oui
WeakAssociationDictionary	oui		oui		
WeakNameSpaceBindings	oui		oui		

TABLE 2.12 – Implémentation des messages dans quelques classes

```
classe/message & '.
    messages do: [:message | stream nextPutAll: message asString]
        separatedBy: [stream nextPutAll: ' & '].
    stream nextPutAll: ' \\\hline
'.
    subclasses:=(subclasses asSortedCollection: [:c1 :c2 | c1 name < c2 name])
                                    asArray.
    subclasses
        do:
            [:subclass |
            hasSelectors := messages collect: [:message |
                                                subclass selectors includes: message].
            stream nextPutAll: subclass name , ' & '.
            hasSelectors do: [:hasSel | hasSel ifTrue: [stream nextPutAll: ' oui ']]
                separatedBy: [stream nextPutAll: ' & '].
            stream nextPutAll: ' \\\hline
```

```
'].
    stream nextPutAll: '\end{tabular}
\caption{Implémentation des messages dans quelques classes}
\end{center}
\end{table}'.
^    stream contents
```

2.7.2 Code latex produit sur un exemple plus court

Cette fois les messages sont :

```
{\tt messages := #( #collect:  #add:  #at:) asArray.}.
```

classe/message	collect :	add :	at :
Dictionary	oui	oui	oui
Interval	oui	oui	oui
LinkedList		oui	oui
List		oui	oui
OrderedCollection	oui	oui	oui
OrderedSet	oui	oui	oui
Palette		oui	oui
Set	oui	oui	oui
SortedCollection	oui	oui	
TableAdaptor		oui	oui

TABLE 2.13 – Implémentation des messages dans quelques classes

```
\begin{table}[htb]
\begin{center}
\begin{tabular}{|c|c|c|c|}\hline
classe/message & collect: & add: & at: \\\hline
Dictionary &  oui  &  oui  &  oui  \\\hline
Interval &  oui  &  oui  &  oui  \\\hline
LinkedList &  &  oui  &  oui  \\\hline
List &  &  oui  &  oui  \\\hline
OrderedCollection &  oui  &  oui  &  oui  \\\hline
OrderedSet &  oui  &  oui  &  oui  \\\hline
```

```
Palette &  &  oui  &  oui  \\\hline
Set &  oui  &  oui  &  oui  \\\hline
SortedCollection &  oui  &  oui  &  \\\hline
TableAdaptor &  &  oui  &  oui  \\\hline
\end{tabular}
\caption{Implémentation des messages dans quelques classes}
\end{center}
\end{table}
```

Chapitre 3

Stream : les accès séquentiels

3.1 Définition

`Stream` et les sous-classes de `Stream` fournissent un mécanisme d'accès séquentiel sur des objets, internes ou externes. Ce mécanisme est une abstraction de programmation, bien commode pour unifier diverses situations pratiques (textes, fichiers, collections, communications), et permettant au programmeur de disposer d'un interface simplifié pour exprimer ses algorithmes.

Il faut respecter quelques règles d'usage qui sont les suivantes :

- *Lorsqu'une instance de* `Stream` *est utilisée, elle a seule le contrôle de l'objet accédé, qui ne devrait plus être utilisé directement.*
- La resynchronisation de l'objet lu ou écrit peut impliquer une opération de mise à jour (`flush`, ou `commit` pour assurer la mise à jour d'une entrée-sortie...). La raison est que le mécanisme séquentiel tamponne les données en lecture ou en écriture afin d'économiser les transferts.
- L'abandon de gestion de l'objet écrit est très important, puisque le mécanisme d'accès peut décider de le remplacer, dans le cas de réceptacles trop petits, par exemple ! Observer la démonstration donnée en section 3.3.4.

3.2 Créations et accès

3.2.1 Créations, readStream, writeStream

On crée un stream [1] en s'appuyant sur des classes telles que `ReadStream, WriteStream, ReadWriteStream`, lors que l'objet est interne. D'autres classes sont utilisées pour les accès séquentiels externes, pour les entrée-sorties. Certain objets savent construire des `Stream` en réponse au message `readStream` ou `writeStream`; c'est le cas des collections, et des fichiers référencés par leurs noms, à partir de la classe `Filename`.

Dans le premier cas, on obtient un stream en expédiant le message `on:` à la classe à instancier. On passe un objet vide lorsque l'on veut écrire, une collection séquençable *pleine* lorsque l'on veut lire. Le stream en lecture s'occupe de faire grossir votre objet lorsque le besoin s'en fait sentir, et cela en respectant sa classe.

- `rs := ReadStream on: 'Aujourd''hui gare au verglas!'. " a ReadStream"`
- `ws := WriteStream on: (Array new: 3). " a WriteStream"`
- `rs := 'Aujourd''hui beau soleil !' readStream.`
- `ws := (Array new: 3) writeStream.`

Les ReadWriteStream permettent de compléter un contenu existant, par exemple en se positionnant à la fin de ce contenu.

3.2.2 Extraction du contenu

Le contenu du stream ne devant pas être accédé directement, Il faut récupérer ce contenu explicitement, en expédiant le message `contents`. L'exemple qui suit montre l'insertion et l'extraction de l'objet :

```
| monStream |
monStream := ReadStream on: 'A Plouescat, la joie eclate!'.
monStream contents. " 'A Plouescat, la joie eclate!'"
```

3.2.3 Tests et positionnements, atEnd, position

Il est possible de savoir si on est parvenu au terme d'une lecture (fin de fichier, dernier élément d'un objet), à l'aide du message `atEnd`. On peut se positionner en fin de stream à l'aide de `setToEnd`. La position dans le stream est contrôlable à l'aide du message `position`.

```
| monStream |
```

1. en français, un flôt séquentiel

```
monStream := ReadStream on: 'A Plouescat, la joie eclate!'.
monStream atEnd. "(printIt) false"
monStream setToEnd ; position. "(printIt) 28"
monStream atEnd. "(printIt) true"
monStream position: 12; position "(printIt) 12"
```

3.2.4 Lecture séquentielle d'un élément, next

La lecture est opérée en expédiant le message `next`. Le stream fait progresser son index interne et effectue d'autres opérations si nécessaire.

```
| array stream |
array := #( 2  3 6 7 9 0 ).
stream := ReadStream on: array.
stream next." 2" stream next. " 3"
```

3.2.5 lecture d'une séquence, nextAvailable : n

On peut prélever un nombre arbitraire d'éléments en utilisant le message `nextAvailable: unEntier`. Bien entendu, on ne peut dépasser la capacité de l'objet accédé, si celle-ci est bornée. Dans ce cas la sous-collection rendue a une taille inférieure à la quantité demandée.

```
| array stream |
array := #( 2  3 6 7 9 0 ).
stream := ReadStream on: array.
stream nextAvailable: 3. " (printIt) #(2 3 6)"
stream nextAvailable:12." (printIt) #(7 9 0)"
```

lecture d'une séquence sur condition, upTo : obj

Il est souvent intéressant de prélever des éléments jusqu'à ce qu'une condition soit obtenue.

upToEnd :
épuise le contenu de l'objet séquencé.

```
| array stream |
array := #( 2  3 6 7 9 0 ).
stream := ReadStream on: array.
stream nextAvailable: 3. " (printIt) #(2 3 6)"
stream upToEnd." (printIt) #(7 9 0)"
```

through : limit :
lit jusque la prochaine occurence de l'objet `limit`, en incluant cet objet.

```
| stream a1 a2 a3 |
stream := ReadStream on: 'La Java du Verglas'.
a1 := stream through:  $a.
a2 := stream through:  $a.
  a3 := stream through:  $a.
     Array with: a1 with: a2 with: a3. " #('La' ' Ja' 'va')"
```

upTo : limit :
lit jusque la prochaine occurence de l'objet `limit`, en excluant cet objet.

```
| array stream subArrays |
array := #( 2  3 6 7 9 0 3 3 9 ).
      subArrays := OrderedCollection new.
stream := ReadStream on: array.
4 timesRepeat: [ subArrays add: (stream upTo: 3)].
subArrays     " OrderedCollection (#(2) #(6 7 9 0) #() #(9))"
```

3.2.6 Exercice

Ecrire un programme qui lit une phrase, et la recopie dans un objet String en enlevant les caractères blancs.

- cas d'une phrase sans blancs consécutifs et ne commençant pas par un blanc
- traiter le cas d'une phrase queconque ayant des blancs dans des positions arbitraires.

Indication :

```
| phrase rs mots mot |
phrase := 'les pieds du zouave baignent'.
rs := phrase readStream.
mots := OrderedCollection new.
[ rs atEnd ] whileFalse: [
mot := rs upTo: $  .
mot isEmpty ifFalse: [ mots add: mot].].
mots  " OrderedCollection ('les' 'pieds' 'du' 'zouave' 'baignent')"
```

3.3 Streams en écriture

3.3.1 Insertion simple, ws nextPut : anOnject

3.3.2 Insertion multiple, nextPutAll : uneCollection

En écriture, on envoie `nextPut: unElement`, qui insère l'éleément au bout de l'objet.

```
|  stream |
stream := WriteStream on: (Array new: 4).
stream nextPut:  3/2.
stream nextPut:  2.2.
stream nextPut:  222.2s .
stream contents "#((3 / 2) 2.2 222.2s)"
```

3.3.3 Insertion multiple, nextPutAll : uneCollection

On peut provoquer l'insertion d'une collection d'éléments avec nextPutAll : uneCollection

```
|  stream |
stream := WriteStream on: (Array new: 4).
stream nextPut:  3/2.
stream nextPutAll: (1 to: 10 by: 3).
stream nextPut:  0 .
stream contents   #((3 / 2) 1 4 7 10 0)
```

3.3.4 Démonstration d'usage non conforme

Dans le code ci-dessous, la variable *array* devrait être oubliée après la création du stream. Noter l'accroissement de capacité effectuée pour gérer les insertions, et noter la classe de l'objet restitué.

```
| array ws |
      array := Array new: 1.
      ws := array writeStream. "a devrait être oublié"
      ws nextPut: 1.
      Transcript cr; show: array first printString. "usage non conforme"
      ws nextPut: 2.
      Transcript cr; show: array last printString."usage non conforme"
      Transcript cr; show: ws contents printString.
```

```
" Dans le Transcript, on lit : 1
nil
#(1 2)"
```

3.3.5 Application à l'affichage des messages

La fenêtre `Transcript` sert à l'affichage des messages. On peut obtenir le texte contenu dans cette fenêtre via le message `value` :

```
Transcript value asString
```

On écrit dans le Transcript comme dans un stream de caractères :

```
Transcript nextPut: Character cr.
Transcript nextPutAll: 'A Plouarzel, ne fais pas de zele'.
Transcript flush.
```

Noter que le message `flush` est indispensable pour provoquer l'affichage immédiat des caractères insérés. Les Stream sur les textes savent gérer l'insertion de caractères non imprimables plus simplement :

```
Transcript cr; tab; nextPutAll: 'A Plouarzel, on fait du zele';
space;cr; flush
```

3.3.6 Application à l'affichage textuel des objets

Un usage courant des streams en écriture est la présentation textuelle des objets. Cette présentation est obtenue en utilisant le message `printString`, qui repose lui-même sur le message `printOn: aStream`.

Toutes les classes ne définissent pas une forme textuelle, mais par héritage, cette impression se résoud au minimum dans la classe racine `Object`.

Voici quelques exemples de formes textuelles :

Fraction : (1 + 2) / 2 " (3/2)"

Float : (1.0 * 2) " 2.0"

Array : (Array with : 1 with : 2) " #(1 2)"

Compiler : (Compiler new) " a Compiler"

On peut appliquer cette facilité aux classes que l'on crée pour supporter une application. Par exemple, si on avait à utiliser des boules de différentes tailles et couleurs, on serait amené à créer une classe `Boule` dotée des variables d'instances `diametre, couleur`.

Une hypothétique méthode d'impression serait alors :

```
   printOn: aStream
aStream nextPutAll: '(Boule '.
```

```
self couleur printOn: aStream.
aStream space.
self diametre printOn: aStream
aStream nextPut: $)
```

Noter le *caractère récursif* de cette méthode, qui permet de déléguer aux objets internes le soin de s'imprimer.

3.4 Opérations sur les fichiers

Un fichier est repéré par un *nom*, instance de la classe `Filename`. On peut choisir un nom de fichier (`String`) via un dialogue spécialisé :

```
| nom |
nom := Dialog requestFileName: 'nom du fichier?'
```

Une fois ce nom choisi, on peut lui associer un fichier en convertissant la chaine en une instance de `Filename` :

```
| nom fileName |
nom := Dialog requestFileName: 'nom du fichier?'.
fileName := nom asFilename.
```

On obtient ensuite aisément un stream en lecture ou écriture en expédiant les messages `writeStream` :

```
| stream |
stream := 'Transcript.file' asFilename writeStream.
stream nextPutAll: Transcript value.
stream commit ; close.
```

ou `readStream` , en lecture :

```
| stream nomDuFichier |
nomDuFichier := Dialog requestFileName: 'nom du fichier?'.
stream := nomDuFichier asFilename readStream.
Transcript nextPutAll: stream contents.
Transcript flush. stream close.
```

Exemple d'un ReadWriteStream complété par une opération d'écriture :

```
| myFileName ws |
myFileName := 'dates.txt' asFilename.
ws := myFileName writeStream.
```

```
Time now printOn: ws.
ws flush ; close.
ws := myFileName readWriteStream.
ws setToEnd ; cr.
Time now printOn: ws.
ws flush ; close.
```

3.4.1 Tests sur les fichiers

La classe Filename supporte un certain nombre de tests utiles pour éviter des opérations irréalisables sur les fichiers. Pour utiliser ces tests, on crée une instance de Filename désignant un fichier et on envoit des messages unaires :

```
|  nomDuFichier canRead exists canWrite directory |
nomDuFichier := 'file.txt' asFilename.
exists := nomDuFichier exists.
canRead := nomDuFichier isReadable.
exists ifTrue: [canWrite := nomDuFichier isWritable.
                    directory := nomDuFichier isDirectory.]
```

3.5 Exercices de base

3.5.1 Analyse lexicale

Extraire les phrases

On peut obtenir le texte de la fenêtre Transcript en expédiant le message `value`. *Construire une collection des phrases contenues dans le Transcript.*

```
 | rStream phrases |
rStream := ReadStream on:  Transcript value asString. "lecture sur la chaine"
phrases := WriteStream on: (Array new: 100). "ecriture sur  des tableaux"
[ rStream atEnd ] whileFalse:
     [ phrases nextPut: ( rStream upTo: $. )].
phrases contents}
```

Renverser les mots

On utilise le texte de la fenêtre Transcript. Produire un texte où les mots sont écrits à l'envers...

```
 | rStream texte ligneStream mot |
rStream := ReadStream on:  Transcript value asString. "lecture sur la chaine"
texte := WriteStream on: (String new: 1000). "ecriture du nouveau texte."
[ rStream atEnd ] whileFalse:
       [ ligneStream := ReadStream on: ( rStream upTo: Character cr).
                [ ligneStream atEnd ] whileFalse:
                   [mot := ligneStream upTo:$ .
                    texte nextPutAll: mot reverse ; space  ].
                    texte cr.
           ].
texte contents}
```

3.6 Usages spécifiques

3.6.1 Analyse de code : Scanner

Plusieurs classes ont des comportements de Stream. C'est par exemple le cas de l'analyseur lexical Smalltalk (Scanner), qui lit du code correctement formé, en restituant les collections d'objets trouvés. Le Scanner peut être utilisé pour d'autres tâches :

```
| texteScanner item items |
item := ''. items := OrderedCollection new.
text := 'Carthago delenda est. 1 + 1 = 2. end'.
texteScanner :=Scanner new on:  text readStream.
[ item = 'end'   ]  whileFalse: [ item :=texteScanner scanToken. items add: item].
items
```

3.6.2 Générateur de nombres aléatoires : Random

Random produit des streams en lecture dans lesquels on peut trouver des nombres tirés au hasard dans l'intervalle `[0, 1[`. En utilisant ces nombres, on peut ensuite produire des objets divers. Ici des entiers, puis des caractères.

```
| rand  ws |
```

```
rand := Random new.
(1 to: 10) collect: [ :i | ( rand next * 10) rounded ].
" #(6 4 3 10 2 7 5 2 1 10)"

ws := String new writeStream.
(1 to: 10) do: [ :i |  | alea aleaChar |
   alea := ( rand next * 10) rounded. "dans [0,9] de N "
   aleaChar := Character value: ( $a asInteger + alea).
   ws nextPut: aleaChar].
 ws contents
" 'fkjfeckaji'"
```

3.7 Algorithmes séquentiels

Cette section est un guide méthodique pour construire sûrement un code séquentiel. Il est orienté vers l'analyse de texte, telle qu'elle pourrait être utilisée pour la lecture de représentations textuelles des objets produites par exemple par les messages `printOn:` .

3.7.1 Construction du contrôle

On s'intéresse en premier lieu à la manière de *poursuivre* le texte à traiter. On va associer aux sous-séquences une boucle de poursuite des blancs et une boucle de poursuite des non-blancs. Cette première version du code se préoccupe simplement de tracer le fonctionnement des automates élémentaires dans le Transcript. Pour cela on affiche le nom de l'automate, loop1, pour la lecture du texte, loop2, pour les blancs ou loop3, pour les non-blancs. On trace également le numéro du pas dans l'étape concernée, et la valeur du caractère courant (*ch*).

```
| texte rs mots ch separ loop1 loop2 loop3 |
texte := 'les sanglots longs    '.
rs := texte readStream.
mots := OrderedCollection new.
ch := separ := Character space.
loop1 := loop2 := loop3 := 0.
[rs atEnd]
  whileFalse:
    [Transcript
      show: 'loop1 ' , loop1 printString;
```

```
        cr.
      loop1 := loop1 + 1.
      [rs atEnd or: [ch ~= separ]]
        whileFalse:
          [ch := rs next.
          Transcript tab;
            show: 'loop2 ' , loop2 printString , ch printString;
            cr.
          loop2 := loop2 + 1].  " boucle 3 "
      rs atEnd
        ifFalse:
          [[rs atEnd or: [ch = separ]]
            whileFalse:
              [Transcript tab;
                show: 'loop3 ' , loop3 printString , ch printString;
                cr.
              loop3 := loop3 + 1.
              ch := rs next]]].
^mots
```

La trace d'exécution de ce programme, simplifiée, est présentée ci-dessous. On notera le passage du premier caractère non-blanc d'un mot entre la boucle 2 et la boucle 3.

```
loop1 0
        loop2 0     $l
        loop3 0     $l
        loop3 1     $e
        loop3 2     $s
loop1 1
        loop2 1     $s
        loop3 3     $s
        loop3 4     $a
        loop3 5     $n
        loop3 6     $g
        loop3 7     $l
        loop3 8     $o
        loop3 9     $t
        loop3 10     $s
loop1 2
        loop2 2     $l
```

```
        loop3 11      $l
        loop3 12      $o
        loop3 13      $n
        loop3 14      $g
        loop3 15      $s
loop1 3
        loop2 3 Character space
        loop2 4 Character space
```

3.7.2 Gestion des variables

Il nous faut maintenant définir les variables et gérer les résultats. On choisit de construire chaque mot à partir d'un stream ouvert en écriture sur une chaine de caractère. Le texte qui suit présente le programme achevé, débarassé du traçage dans le Transcript, et avec l'impression du tableau mots.

```
| texte rs mots ch separ ws |
texte := 'les sanglots longs   '.
rs := texte readStream.
mots := OrderedCollection new.
ch := separ := Character space.
[rs atEnd]
  whileFalse:
    [[rs atEnd or: [ch ~= separ]] whileFalse: [ch := rs next].
    rs atEnd
      ifFalse:
        [ws := String new writeStream.
        [rs atEnd or: [ch = separ]]
          whileFalse:
            [ws nextPut: ch.
            ch := rs next].
        mots add: ws contents]].
^mots
" OrderedCollection ('les' 'sanglots' 'longs')"
```

3.7.3 Exercice

On se donne à retrouver dans un fichier de texte les éléments suivants :

mots : séquences sans séparateurs commençant par une lettre

nombres : séquences de chiffres

séparateurs : caractères blancs, tabulations, fin de lignes, ...

- Ecrire un analyseur qui construit deux collections de *mots* et de *nombres*, en ignorant les autres caractères.
- Même question en cherchant à obtenir l'unicité des mots (enlever les doublons), et en triant les mots par ordre alphabétique.

3.7.4 Problème

Construire un programme de références croisées (cross-reference), qui affiche les mots d'un texte, avec les numéros de ligne où chaque mot est apparu. Les mots sont triés et les doublons sont éliminés. Les numéros de lignes sont insérés dans une collection ordonnée qui est la valeur associée à une clé dans un dictionnaire. Les clés sont les mots du texte.

Pour un exemple de ce traitement voir, par exemple, la commande `cxref` de Linux.

Chapitre 4

Set, Dictionary et Bag

4.1 Collections non-ordonnées

Les éléments de ces collections ne sont pas rangés selon un ordre prédictible : on ne peut pas accéder aux éléments via une clé externe, tel qu'un index, ou un ordre connu. Cette propriété est liée au mécanisme d'adressage dit *associatif*, où la position d'un élément dans la collection dépend de la valeur de cet élément et de l'historique des accès (voir les explications en section 4.7).

La hiérarchie de classes se présente de la manière suivante :

```
Object ()

    Collection ()
        Bag ('contents')
        Set ('tally')
            Dictionary ()
            IdentitySet ()
```

- Set : collection garantissant l'unicité des éléments.
- Dictionary : accès par une clé, qui est en général un objet d'une classe déterminée. La clé garantit l'unicité.
- Bag : chaque élément a un compteur associé,

Une première utilisation de ces collections est la mise en œuvre d'algorithmes très simples reposant uniquement sur leurs propriétés :

Trouver tous les mots apparus dans un texte

```
| bobyADit lesMotsDeBoby |
bobyADit := 'ta pa ta pa tapa tout dit tapa tout dit a ta dou dou'.
lesMotsDeBoby := bobyADit tokensBasedOn: Character space.
lesMotsDeBoby asSet asSortedCollection asArray
```

```
"#('a' 'dit' 'dou' 'pa' 'ta' 'tapa' 'tout')"
```

4.2 Set

4.2.1 Création

Les ensembles sont des collections dynamiques, qui grandissent en fonction des besoins. On crée de nouveaux ensembles par `Set new`, ou, si on est capable d'apprécier correctement une taille idoine, par `Set new: nElements`. Si on connait déjà les éléments, parce qu'ils sont rangés dans une autre collection, alors on peut aussi instancier par `Set withAll: une Collection`.

L'algorithme (hachage) qui sert à la mise en œuvre des ensembles a de bonnes caractéristiques en temps de recherche d'un élément.

4.2.2 Accès

L'usage simple de Set peut être résumé de la manière suivante :

- `unSet add: unObjet`,
  `unSet addAll: uneCollection`, ajout dans l'ensemble.
- `unSet includes: unObjet` vrai ou faux selon que `unObjet` soit présent ou absent.
- `unSet remove: un Objet` ou
  `unSet remove: un Objet ifAbsent:unBlocException.`
  Dans le premier cas, on enlève `unObjet` de `unSet`, si cet objet existe. Sinon, on déclenche une erreur.
  Si le programmeur n'est pas certain de la présence de l'objet cherché, il utilise la seconde forme en passant un bloc d'exception, qui peut être vide...
- `unSet size` : nombre d'éléments présents dans l'ensemble,
- `unSet capacity` : capacité de stockage de l'ensemble. Plus le rapport *capacity/size* est grand, plus le temps d'accès risque d'être rapide.

4.3 Dictionary

Un Dictionnaire est un ensemble dont les éléments sont des instances de la classe `Association`, couplant une clé (`key`), et une valeur (`value`). L'unicité d'une association dans un ensemble donné est garantie par l'unicité de la clé, propriété héritée de la classe `Set`.

Les instances de Association sont fréquemment créées à l'aide du message binaire $->$. Par exemple, `#boulesRouges->40` associe l'entier 40 au symbole `#boulesRouges`.

La structure de données de Dictionary est intéressante à plusieurs titres :

- Possibilité de désigner un objet par une clé souvent symbolique (Symbol, String, nombres...)
- Rapidité d'accès, due au hachage (mêmes performances que les ensembles, voir en section 4.5).
- Possibilité d'utiliser les dictionnaires comme des tables de hachage, et pour des clés tout à fait quelconques.
- Dynamicité de la collection qui grandit avec les besoins.

4.3.1 Création et propriétés héritées de Set

On peut se reporter à la description donnée dans `Set`. Si on souhaite utiliser `Dictionary withAll: uneCollection`, il faut cependant retenir que tous les éléments de `uneCollection` doivent être des Associtations. L'exemple qui suit montre comment ajouter un dictionnaire à un autre, comment créer un dictionnaire initialisé.

La clé servant à adresser le dictionnaire est simplement un entier, la valeur est un caractère extrait d'une chaine. On remplit le premier dictionnaire en notant les index des blancs et les index des ponctuations. Le dictionnaire apparait alors comme un tableau "creux".

```
testDico1
   "self testDico1"

   | dico1 dico2 vers |
   dico1 := Dictionary new.
   dico2 := Dictionary new.
   vers := 'le petit homme de la jeunesse, a casse son lacet de soulier,'.
   vers keysAndValuesDo: [:index :car | car = $
         ifTrue: [dico1 at: index put: car]
         ifFalse: [car isLetter ifFalse: [dico1 at: index put: car]]].
```

```
dico1 addAll: dico2 associations. "ajout d'un dictionnaire a un autre dictionna
^Dictionary withAll: dico1 associations "creation d'un dictionnaire pre-rempli"
```

Parmi les messages hérités de Set, se trouvent les opérations ensemblistes qui portent sur les clés : intersection, union, soustraction...Dans le code ci-dessus, la soustraction des deux dictionnaires serait simplement spécifiée par `dico1 - dico2`.

4.3.2 Accès, ajouts et suppressions

Les méthodes les plus simples pour insérer, supprimer, tester sont les suivantes :

- `unDictionnaire at: uneCle put: unObjet`,
  par exemple `dico at: #titi put: 12`.
- `unDictionnaire add: uneAssociation`,
  par exemple `dico add: #titi->12`.
- `unDictionnaire removeKey: uneCle`,
  par exemple `dico removeKey: #titi`. Le résultat est la valeurs associée à la clé (12).
  On dispose aussi de la variante `unDictionnaire removeKey: uneCle ifAbsent: unBloc`.
- `unDictionnaire includesKey: uneCle`,
  par exemple `dico includesKey: #titi` rendrait `true` dans le contexte ci-dessus.
- `unDictionnaire includes: unObjet`,
  par exemple `dico includes: 12` rendrait `true`, alors que `dico includes: #titi` rendrait `false`,
  De même `unDictionnaire occurencesOf: unObjet`, rend le nombre de fois où `unObjet` est apparu en valeur de l'association.
- `unDictionnaire keyAtValue: unObjet` renvoie une clé associée à une valeur, si toutefois cette valeur existe. par exemple `dico keyAtValue: 12` rendrait `#titi`...

4.3.3 Itérations

On peut distinguer les itérations générales, portant sur les *valeurs* plutôt que sur les clés, les itérations portant sur les clés, et enfin celles qui portent sur le couple clé-valeur.

- `unDico do: unBloc`
  itération sur toutes les valeurs, par exemple :

```
| dicoDesMots versDePrevert stream |
dicoDesMots := Dictionary new.
stream := WriteStream on: ''.
versDePrevert := 'le petit homme qui dansait dans ma tete'.
(versDePrevert tokensBasedOn: $ )
      do: [:mot | dicoDesMots at: mot asSymbol put: mot size].
dicoDesMots do: [:longueurs | stream nextPutAll: longueurs printString; space].
^stream contents
  " '2 4 4 5 7 3 2 5 '"
```

La transformation de la chaine de caractère en symbole (`asSymbol`) n'est pas indispensable.

– `keysDo:` itère sur les clés du dictionnaires :

```
...
dicoDesMots keysDo: [:mot | stream nextPutAll: mot; space].
^stream contents
" 'dans tete homme dansait qui ma petit le '"
```

– `keysAndValuesDo:` permet d'accéder simultanément à la clé et à la valeur :

```
...
dicoDesMots keysAndValuesDo: [:mot :taille |
              stream nextPutAll: mot, '(', taille printString,')'; space].
..
" 'tete(4) homme(5) ma(2) petit(5) qui(3) le(2) dans(4) dansait(7) '"
```

– `collect:` et fonctionne sur les valeurs associées aux clés.

```
    | dicoDesMots versDePrevert  |
    dicoDesMots := Dictionary new.
    versDePrevert := 'les sept eclats de glace de ton rire etoile'.
    (versDePrevert tokensBasedOn: $ )
        do: [:mot | dicoDesMots at: mot size put: mot asSymbol].
    ^ dicoDesMots collect:  [:mot   | mot reverse]"
" OrderedCollection ('ed' 'not' 'erir' 'ecalg' 'eliote')"
```

– `select: unBloc`, `reject: unBloc` opèrent sur les valeurs associées aux clés, mais renvoient un dictionnaire.

```
...
    ^ dicoDesMots select:  [:mot   | mot size <= 4 ]
      " Dictionary (2->#de 3->#ton 4->#rire )"
```

4.4 Bag

Cette classe s'appuie sur Dictionary ou IdentityDictionary pour compter le nombre d'occurences d'un élément. L'intérêt est triple : vitesse d'accès (due aux dictionnaires), comptage, et compacité de la structure de données si le nombre des occurrences est élevé.

4.4.1 Ajouts et suppressions

Les sacs (classe `Bag`) ont un interface permettant d'ajouter ou d'enlever des éléments :

- `unSac add: unObjet`
  Le compteur des occurences de `unObjet` est incrémenté.
- `unSac remove: unObjet`
  `unSac remove: unObjet ifAbsent: aBlock`
  Le compteur des occurences de `unObjet` est décrémenté. Si ce compteur tmobe à 0, l'objet disparait du sac.
  Dans la seconde version le bloc d'exception est évalué au cas où l'objet n'existe pas : enlever un objet absent est une erreur.
- `unSac addAll: uneCollection` `unSac removeAll: uneCollection`
  ajoute ou enlève une collection d'objets.
- `unSac add: newObject withOccurrences: n`
  `unSac removeAllOccurrencesOf: anObject ifAbsent: aBlock`
  ajoute n occurences du même objet, ou enlève toutes les occurences d'un objet.

4.4.2 Énumérations

- `do: collect: inject: ...` : Le principe utilisé est d'évaluer les blocs paramêtres pour chaque élément de l'ensemble. Par exemple, si `unObjet` apparait 10 fois dans le sac, il y aura 10 évaluations du bloc paramêtre. Lorsque l'itérateur renvoie une collection, celle-ci est également un `Bag`. Dans l'exemple qui suit `voyelles` est un `Bag` qui ne comporte que `false` et `true`.
  Compter les voyelles dans une chaine

```
| b voyelles |
    b := Bag new.
    b addAll: 'il pleut sans cesse sur Brest'.
    voyelles := (b collect: [:lettre | lettre isVowel ]).
    voyelles occurrencesOf: true
```

```
"8"
```

– `valuesAndCountsDo: unBloc` : S'il est utile d'associer les éléments et leurs compteurs associés, on utilise ce message et on construit un bloc à deux paramêtres (élément et son compteur).

Imprimer par ordre croissant le nombre de caractères apparus dans une chaine

```
| stream sort |
stream := WriteStream on: (String new:200).
"preparation d'une collection comportant des Associations triees selon le champs value"
sort := SortedCollection new sortBlock: [:a1 :a2 | a1 value < a2 value].
'le petit homme qui chantait sans cesse' asBag valuesAndCountsDo: [:v :c | sort add: v->c].
sort do: [:association | stream nextPut: association key ; space.
     stream nextPutAll: association value printString, ' - '].
     stream contents
"'q 1 - p 1 - o 1 - l 1 - u 1 - n 2 - m 2 - h 2 - c 2 -
  i 3 - a 3 - t 4 - s 4 - e 5 -   6 - '"
```

4.5 Performances

On veut comparer les performances en insertion et suppression sur les collections Set, SortedCollection et OrderedCollection.

Dans un premier temps, on laisse l'ensemble grossir au fur et à mesure des besoins, en testant les 3 collections. Dans un second temps, on fixe d'emblée la taille de l'ensemble à 4 fois la taille des éléments.

Dans les deux cas les objets insérés sont des nombres réels aléatoires.

4.5.1 Boucle externe du test, formatage

On y reconnait un `stream` en écriture sur une chaine de caractères, qui sera utilisé pour produire une table au format LaTeX(voir la table 4.1).

Le bloc `eval` possède deux paramêtres, l'un contient la classe sur laquelle on effectue le test de performance, l'autre est un bloc servant à instancier cette classe. On repère les 6 évaluations de ce bloc en fin de méthode.

Dans le bloc, le temps de calcul insertion/suppression est produit en expédiant le message `testCollection: aClass creationBlock: aBlock times: nTimes`. Ce message est décrit plus bas.

Il faut noter que l'on répète `nTimes` fois les opérations et que les temps sont rendus dans une OrderedCollection de taille `nTimes`.

```
testNew
   "Test testNew"

   | stream eval resultats nTimes |
   nTimes := 3.
   stream := WriteStream on: ''.
   stream nextPutAll: 'Classe & '.
   nTimes timesRepeat: [stream nextPutAll: 'add', ' & ', 'remove', '& '].
   stream nextPutAll: 'add moy.', ' & ', 'remove moy. ', '\\\hline'; cr.
   eval :=
         [:aClass :aBlock |
         | sumX sumY |
         stream nextPutAll: aClass name, ' & '.
         resultats := self
                  testCollection: aClass
                  creationBlock: aBlock
                  times: nTimes.
         sumX := sumY := 0.
         resultats
            do:
               [:point |
               stream nextPutAll: point x printString, ' & ', point y printString,
               sumX := sumX + point x.
               sumY := sumY + point y].
         stream nextPutAll: (sumX // resultats size) printString, ' & ',
                           (sumY // resultats size) printString, '\\\hline'; cr].
   eval value: Set value: [Set new].
   eval value: SortedCollection value: [SortedCollection new].
   eval value: OrderedCollection value: [OrderedCollection new].
   eval value: Set value: [Set new: 8000].
   eval value: SortedCollection value: [SortedCollection new: 8000].
   eval value: OrderedCollection value: [OrderedCollection new: 8000].
   ^stream contents
```

4.5.2 Boucle interne du test

Ici, on effectue une série de mesures identiques sur une classe passée en paramêtre, que l'on instancie via un bloc également passé en paramêtre.

tRand est le temps de génération de 2000 nombres. tAdd est le temps d'insertion, tRemove, le temps de suppression.

Le temps est évalué en millisecondes en utilisant le message `Time millisecondsToRun: unBloc`. *On renvoie une collection de points comportant en abcisse le temps d'insertion, en ordonnée le temps de suppression (par pure commodité).*

```
testCollection: collClass creationBlock: aBlock times: n
   | rand set2 tAdd tRemove tRand set1 resultats |
   resultats := OrderedCollection new: n.
   rand := Random new. "Generateur de nombre aleatoire"
   n
      timesRepeat:
         [set2 := aBlock value.
         tRand := Time millisecondsToRun: [2000 timesRepeat: [rand next * 100]].
         tAdd := Time millisecondsToRun: [2000 timesRepeat: [set2 add: rand next * 100]].
         set1 := set2 copy asArray.
         tRemove := Time millisecondsToRun: [set1 do: [:elt | set2 remove: elt]].
         resultats add: tAdd - tRand @ tRemove].
   ^resultats
```

4.5.3 Bilan

Classe	add	remove	add	remove	add	remove	add moy.	remove moy.
Set	514	946	503	908	507	920	508	924
SortedCollection	769	2983	805	1564	759	2787	777	2444
OrderedCollection	50	1574	41	2876	17	1568	36	2006
Set	267	361	244	353	264	349	258	354
SortedCollection	1380	2869	852	1551	1326	2846	1186	2422
OrderedCollection	49	1544	5	2887	53	1613	35	2014

TABLE 4.1 – Performances comparées. Le premier groupe de tests porte sur des collections créées avec `new`. Le second groupe porte sur des ensembles portés, d'emblée à une capacité de 8000

- On peut constater que le coût du sous dimensionnement de la structure du départ peut être important : dans le cas de Set, cela compte pour un facteur 2.
- Si la structure est grande, le coût en suppression est réduit (le second Set est 4 fois trop grand, et on remarque que les suppressions sont à peine plus lentes que les insertions). Dans le cas contraire, le coût des suppressions peut être jusque deux fois plus élevé que celui des insertions.
  L'explication se trouve dans la gestion des collisions et le nombre de celles-ci. . .
- l'accès en insertion sur les OrderedCollectiuon est très rapide, on ne peut pas en dire autant des suppressions.
- les SortedCollection marquent un avantage en recherche qui n'apparait pas ici.

4.6 Exercices

4.6.1 Décomposition d'un nombre entier en facteurs premiers

Décrire un algorithme de décomposition s'appuyant sur un ensemble de diviseurs possibles. L'ensemble de ces diviseurs est réduit chaque fois qu'un de ses éléments est traité. S'il s'agit d'un diviseur, on réduit les nombres de l'ensemble à l'aide de ce diviseur. Dans le cas contraire on élimine tous les multiples du diviseur. Les diviseurs sont accumulés dans un `Bag`.

```
!Integer methodsFor: 'set-exemple'!
decompose
   | diviseurs diviseursPossibles nb minBlock min maxBlock max sizeColl |
   diviseurs := Bag new.
   diviseurs add: 1. nb := self.
   sizeColl := OrderedCollection new.
   diviseursPossibles := (2 to: self // 2) asSet.
   min := 2. max := 1.
   minBlock := [:number | min := min min: number].
   maxBlock := [:number | max := max max: number].
   [diviseursPossibles isEmpty or: [nb = 1]]
      whileFalse:
         [| diviseur division |
         sizeColl add: diviseursPossibles size.
```

```
         diviseur := min.
         division := nb / diviseur.
         min := nb // max.
         (division isKindOf: Integer)
            ifTrue:
               [diviseurs add: diviseur.
               maxBlock value: diviseur.
               minBlock value: diviseur.
               nb := division.
               diviseursPossibles := diviseursPossibles
                        collect:
                           [:n |
                           | x | x := n / diviseur.
                           x := (x isKindOf: Integer)    ifTrue: [x]    ifFalse: [n].
                           x > (nb / max)
                              ifTrue: [x := min]
                              ifFalse: [(#(0 1) includes: x) ifTrue: [x := max]].
                           minBlock value: x.
                           x]]
            ifFalse: [diviseursPossibles := diviseursPossibles
                        reject:
                           [:n |  minBlock value: n.
                           ((n / diviseur isKindOf: Integer) or: [n > (nb / max)])
                              or: [n = diviseur]]]].
   Transcript cr; show: sizeColl printString ; cr.
   ^diviseurs
"(3*25*8*10*4) decompose
 OrderedCollection (11999 5999 2999 1499 749 374 186 93 93 20 13 13 1 1)
 Bag (1 2 2 2 2 2 2 3 5 5 5)"
```

4.7 Hachage

Le hachage d'un objet permet d'adresser une table par le contenu. Cette technique permet d'éviter des opérations de recherche par énumération ou dichotomie. Cette technique est à la base des Set, donc de Dictionary, donc de Bag qui encapsule un dictionnaire.

La présente section décrit sommairement le mécanisme. La réalisation d'algorithmes performants requière une plus grande attention que cette première approche.

Les trois figures qui suivent montrent une opération de recherche :

– On utilise une fonction qui rend un index à partir d'un objet (message `hash` expédié à l'objet). Des exemples très simples de hachage sont un modulo appliqué à un objet entier, une fonction $c1 \times 256 + c2$ appliquée aux deux premiers caractères d'une chaine...

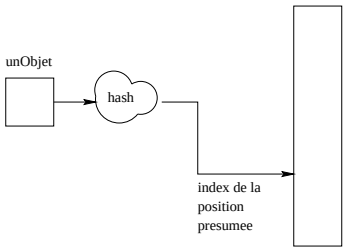


FIGURE 4.1 – L'objet est haché et on obtient un index dans la table

– L'index présumé ayant été produit, on accède à la table. Si la table ne contient rien (`nil`), alors l'objet n'est pas dans la table. Sinon on compare l'objet cherché et l'objet trouvé (`a = b`). Voir figure 4.2. S'il y a égalité, on a trouvé l'objet...

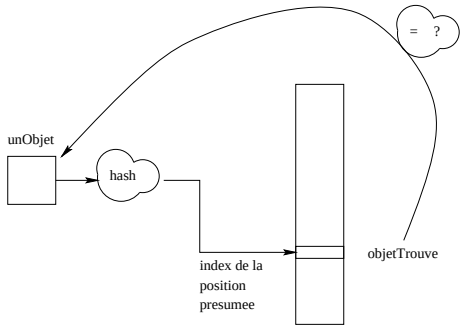


FIGURE 4.2 – On compare l'objet implanté dans la table et l'objet cherché

– Sinon, on descend en séquence en cherchant soit `nil`, soit l'égalité (voir figure 4.3).

Les opérations d'insertion sont menées de la même manière, il s'agit ici de trouver le premier index libre lors de la recherche en séquence. Lorsque la quantité de collisions grandit (place libre de plus en plus petite), il est préférable de reconstruire la table en doublant sa taille.

Les suppressions requièrent une reconstruction au moins partielle de la table.

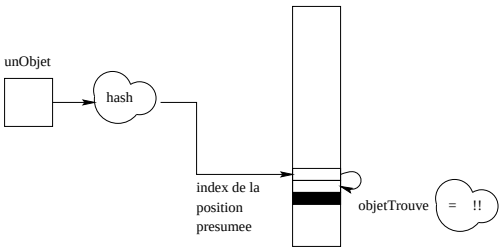


FIGURE 4.3 – Recherche en séquence

Deuxième partie

Interfaces standards et spécifiques

Chapitre 5

Construction d'interfaces standard

5.1 Composition des objets – dépendances

La disponibilité de mécanismes permettant de composer des objets complexes les uns avec les autres est un aspect fondamental de la méthodologie objet :

- la réutilisation de composants ne doit pas se faire au prix de modifications sur ces composants.
- l'héritage, qu'il soit multiple ou simple n'est pas une solution rationnelle pour la spécialisation des composants.
- il est nécessaire de pouvoir associer des composants dont les fonctionalités sont différentes et complémentaires les unes avec les autres.

Un des outils de composition offert par Smalltalk est le système de *dépendances*. Il s'agit d'un mécanisme très simple permettant de répercuter un changement opéré à l'intérieur d'un composant sur une série d'autres composants "*clients*". L'usage de ce mécanisme n'oblitère pas la logique interne des composants, il permet simplement d'exprimer des contraintes de comportement dans un assemblage.

5.1.1 Dépendances

Deux objets A et B sont dépendants lorsqu'une modification de A entraine une modification de B ou le contraire. Lorsque la relation est double les objets sont inter-dépendants.

Exemples :

– Dans les browsers, la liste des méthodes dépend de la sélection dans la liste des classes.
– Dans un interface utilisateur, la présentation graphique d'un objet doit refléter immédiatement l'état de cet objet.

5.1.2 Gestionnaires de valeurs

La gestion automatique des dépendances s'appuie sur l'encapsulation d'un objet dans un gestionnaire, chargé d'en surveiller l'accès. Le comportement est fixé dans la classe *ValueHolder*.

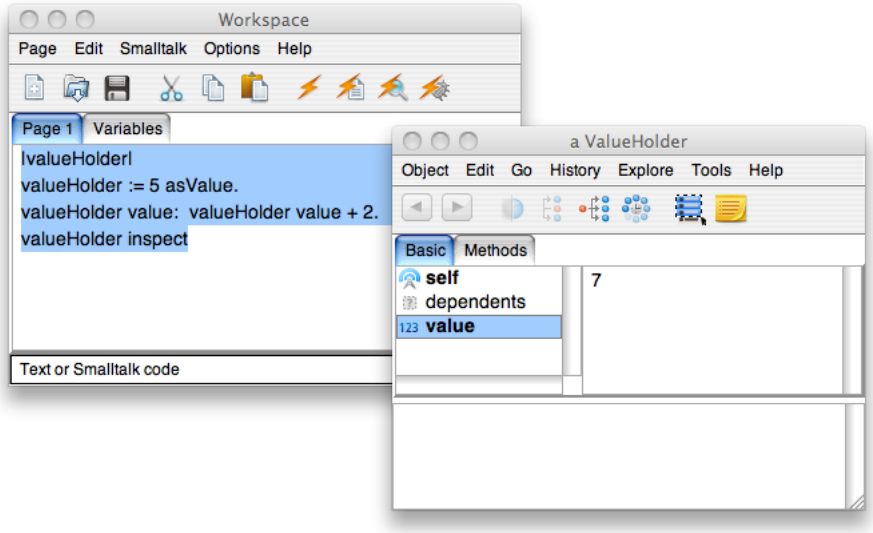


FIGURE 5.1 – Un ValueHolder encapsule une valeur qui doit être prélevée par value, et rechargée par value :

Le gestionnaire est créé en envoyant le message **asValue** à un objet quelconque.

Les interventions sur l'objet s'effectuent par :
– **value** pour accéder l'objet,
– **value : aValue** pour changer l'objet.

5.1.3 Implantation d'une dépendance

Le gestionnaire sait conserver des listes de messages à expédier à des objets lorsqu'une modification se produit.

Pour programmer une dépendance, on provoque l'intallation d'un **DependencyTransformer**. Ceci se fait par le message (section 5.5.3, `UIBibli..initialize`) :

unValueHolder onChangeSend : #unMessage to : unObjetDependant

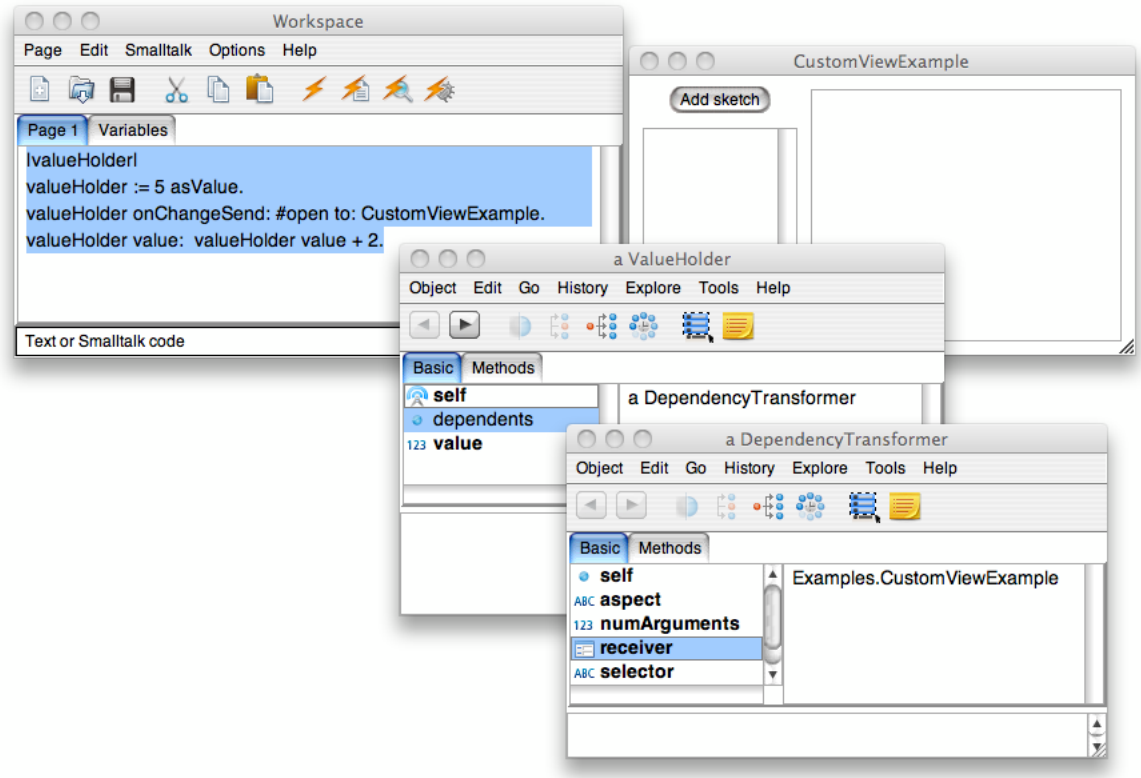


FIGURE 5.2 – La figure montre un ValueHolder dans lequel on implante une dépendance qui provoque l'ouverture de la fenêtre CustomViewExample. L'inspecteur ouvert sur le value holder montre clairement l'implantation de la dépendance (receiver, selector) et la valeur encapsulée.

5.1.4 Dépendances circulaires

Une situation plus complexe est celle où A dépend de B et B dépend de A. Il est alors impossible d'actionner une modification comme dans la structure précédente car cela provoquerait une récursivité croisée dsans fin...

Dans ce cas on est amené à inhiber les dépendances avant d'effectuer les modifications. Ceci s'opère par expédition du message

retractInterestFor : unObjetDependant

au **valueHolder**.

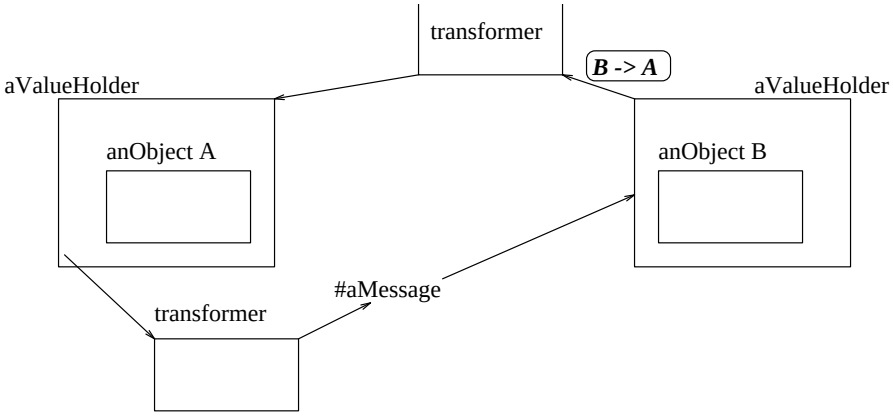


FIGURE 5.3 – double dépendance

5.2 Conception des applications interactives : MVC

Il s'agit d'un domaine où les technologies objet sont très efficaces. VisualWorks s'appuie sur une méthode de développement mettant en action plusieurs composants :

- le modèle, où le domaine du modèle,
  il s'agit des données sur lesquelles on souhaite agir.
- la vue graphique,
  il s'agit de représentation graphiques composites hiérarchiques et s'appuyant sur un ensemble de composants extensible : boutons, décors, listes, champs de saisie, éditeurs...
- le contrôleur de la vue. C'est le mécanisme fixant les interactions potentielles entre l'utilisateur et les deux précédentes entités. Les contrôleurs ont une architecture hiérarchique similaire à celle des vues. Ils permettent de définir la gestion d'évènements clavier ou souris dans un cadre standardisé.

MVC produit des interfaces robustes grace à son système hiérachique et l'abstraction des détails de fonctionnement d'un niveau à un autre.

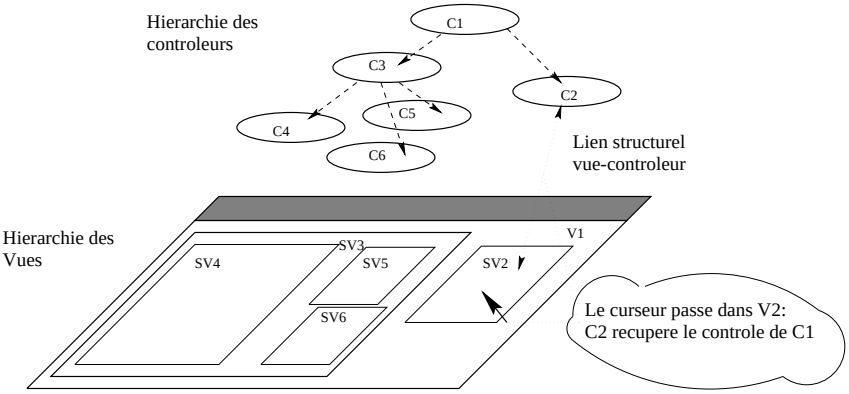


FIGURE 5.4 – Structure hiérarchique de MVC : chaque composant visuel est associé à un contrôleur. Ceux-ci s'appellent mutuellement lorsque le curseur bouche, lorsque la souris clique etc...

5.2.1 Interface et Modèle

Une application comporte typiquement ces deux composantes. La règle conceptuelle est de séparer *strictement* leurs comportements. L'outillage de VisualWorks crée les interfaces en tant que sous classe de `ApplicationModel` qui fournit tous les éléments nécessaires à la construction, l'activation et le contrôle des interfaces.

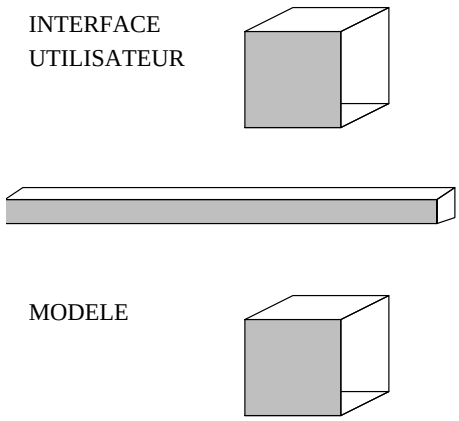


FIGURE 5.5 – Séparation Interface-Modèle

5.2.2 Définitions

- **Widget :** (Window Gadget) composant d'interface doté d'une partie visuelle et d'un contrôleur optionnel. Un tel composant peut être actif (réaction à la souris, au clavier), ou passif.
  Exemples : champs, étiquettes, cadres, menus, listes, tables, icones...
- **Builder :** constructeur d'interface. Il interprête une spécification textuelle (windowSpec) qui comporte des paramêtres géométriques divers et a la responsabilité de dessiner la fenêtre.
  Toutes les interventions sur les divers composants sont à adresser au **builder** : modification d'une étiquette, inhibition ou activation d'un widget...
  Le **Builder** a deux modes. En mode édition, il permet de modifier une spécification dynamiquement. L'outil CANVAS utilise le **Builder** de cette manière.
- **les "Value models" :** instances de `ValueHolder` assurant l'interface avec le modèle, dans l'application. Ce sont des adaptateurs reliant le widget à l'attribut correspondant du modèle. Une modification interne ou externe du modèle se répercute automatiquement par un changement d'état de l'interface.

Le Builder est appelé par la classe `ApplicationModel` pour batir l'interface lors des ouvertures. La construction d'un interface est un processus récursif qui parcours la hiérarchie de composants.

Chaque widget comporte une vue et un contrôleur préférentiel. Les widgets peuvent être des applications complètes réutilisées.

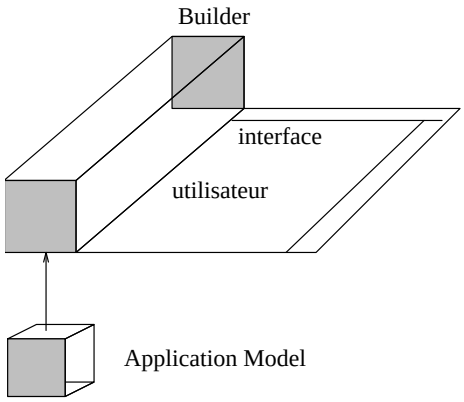


FIGURE 5.6 – Construction de l'interface par le Builder : celui-ci 'peint' l'interface pour le compte de la classe ApplicationModel, et conformément à une spécification litérale (voir les méthodes `windowSpec`).

5.3 Lancement du Canvas

5.3.1 Accès aux outils

Pour utiliser les outils de conception d'interface, on utilise l'icone présentant un chevalet (figure 5.7). Noter en même temps la présence de l'icone de la documentation en ligne (petit livre portant un point d'interrogation) et le menu `Painter` où on trouvera des accès secondaires à l'éditeur de menu, l'éditeur d'icones etc. . .

– Vue sur le Transcript.

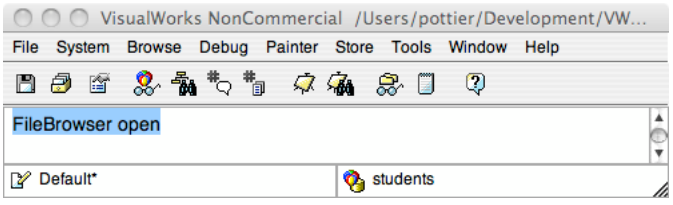


FIGURE 5.7 – Vue sur le Lanceur

5.3.2 Description des outils interactifs

Après le lancement du canvas, on se trouve en présence de trois fenêtres : à gauche, la palette, en haut les outils de contrôle, au centre l'éditeur d'interface.

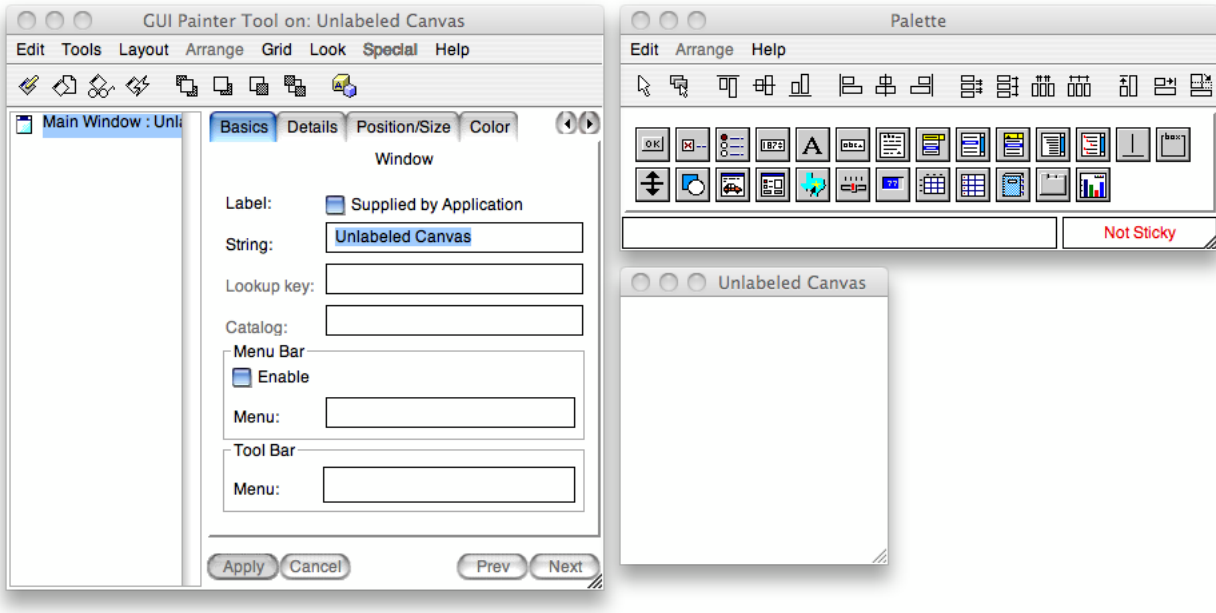


FIGURE 5.8 – Vue sur les outils du Canvas

- **la Palette**
  Elle comporte des composants que vous pouvez déposer sur l'interface en édition. Ces composants sont standards, ou spécifiques à une application. Il est possible d'étendre la palette soit même : l'icône du bas à droite est un outil de présentation de graphes non standard.
  Les principaux composants encapsulent de l'outillage d'interfaçage (table 5.1) :
- **la fenêtre de composition** (Unlabeled Canvas) :
  Elle sert à dessiner l'allure de l'interface, en terme de composants. Pour installer un composant dans l'interface, il suffit simplement de faire glisser son icône à l'endroit voulu de l'éditeur.
  Pour démarrer, on peut par exemple déposer installer une étiquette (Label) au centre de l'éditeur, puis installer un bouton à coté de cette

1 Bouton	2 Boite à cocher	3 Boutons radio
4 Texte étiquette	5 Champs éditable	6 Éditeur de texte
7 Menu déroulant	8 Liste	9 Menu
10 Barres de décors	11 Boite de décors	13 Figures

TABLE 5.1 – Palette : outils de composition courants. La Palette est extensible.

étiquette. Noter la fénêtre de paramétrage (Painter) qui se modifie lors que l'on intervient sur la fenêtre de composition. Dans notre cas il est nécessaire de prévoir une méthode associée au bouton action que l'on a nommé ici #doAction.

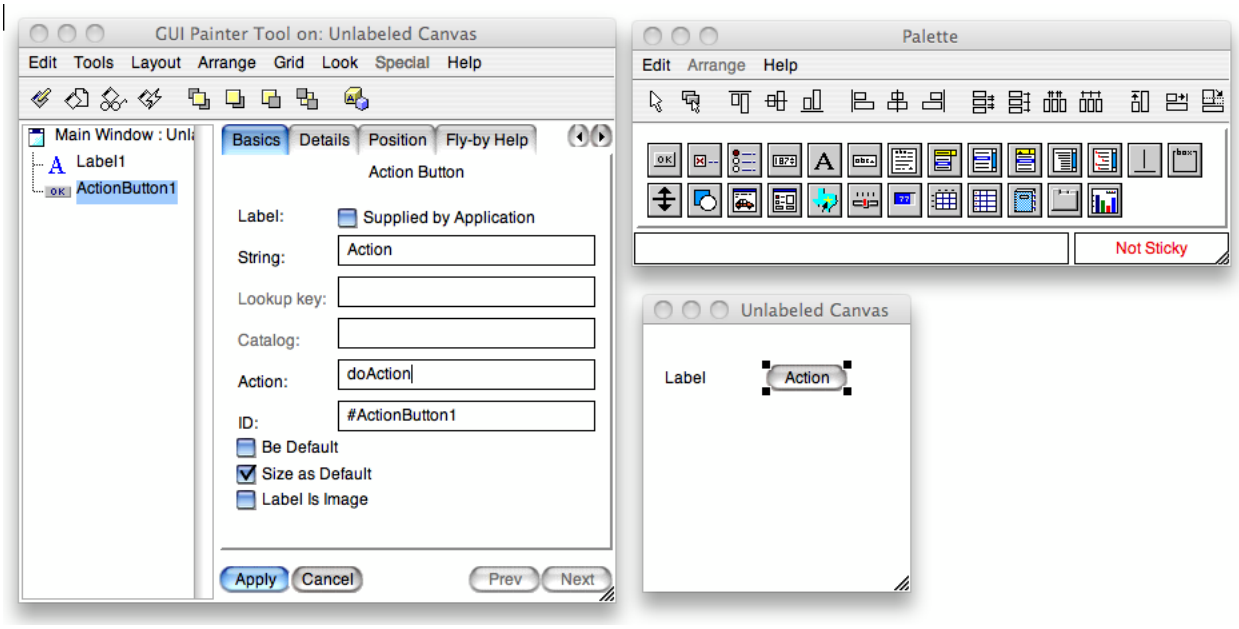


FIGURE 5.9 – Exemple d'un petit interface

– **La fenêtre de contrôle** (Gui Painter Tool).
Cette fenêtre assure le contrôle des outils annexes (éditeur de menu, éditeur d'icônes, alignements divers, groupement des objets et possibilité de mise à l'échelle de ces groupes), et *surtout*, il assure aussi la gestion de l'interface entre les composants de l'interface et le modèle sur lequel l'interface travaille.
On repère le bouton PROPERTIES qui permet de lancer la fenêtre de

paramétrage de propriétés, appliquables aux composants et à la fenêtre d'application elle-même.

5.3.3 Properties

Cette fenêtre *dépend* du composant actif dans la fenetre d'édition. Elle présente un jeu de paramêtres configurables pour le composant, ou les moyens de coupler ce composant à des variables et des méthodes d'un modèle.

Les deux fenêtres de propriétées de la figure 5.10 sont commutées en sélectionnant la fenêtre d'édition et le bouton action. Dans le premier cas on peut changer le nom de la fenêtre, dans le second cas on modifie l'étiquette placée sur le bouton, et le message expédié au modèle lorsque le bouton est enfoncé...

5.4 Couplage d'un interface et d'une classe

La première chose à faire est de disposer d'une classe dite de "*modèle*" sur laquelle l'interface va travailler. Cette classe peut être pratiquement vide, mais il est préférable d'utiliser un petit exemple déjà développé : Passwd (mots de passe unix), Bibliotheque etc...

Pour coupler l'interface au modèle, il est nécessaire de créer une classe qui contiendra la description de l'interface. Ceci se fait une fois et une seule.

5.4.1 Création d'une classe interface

On enfonce le bouton install et obtient la fenêtre de gauche, figure 5.11. On entre dans le champs du haut la classe interface. Généralement on commence les noms de ces classes par UI. Puis on accepte.

Comme la classe n'existe pas, a priori, la première fois que l'installation est effectuée, un second dialogue apparait qui propose de créer automatiquement cette classe (ici `UIUnixPasswd`).

Il faut accepter le choix de `ApplicationModel` en tant que superclasse de `UIUnixPasswd`, et valider la création.

5.4.2 Installation d'un interface

Toutes les autres opérations INSTALL seront des simples regénérations de la méthode #windowSpec dans `UIUnixPasswd`. Ces opérations sont indispensables avant de pouvoir profiter des modifications d'éditions.

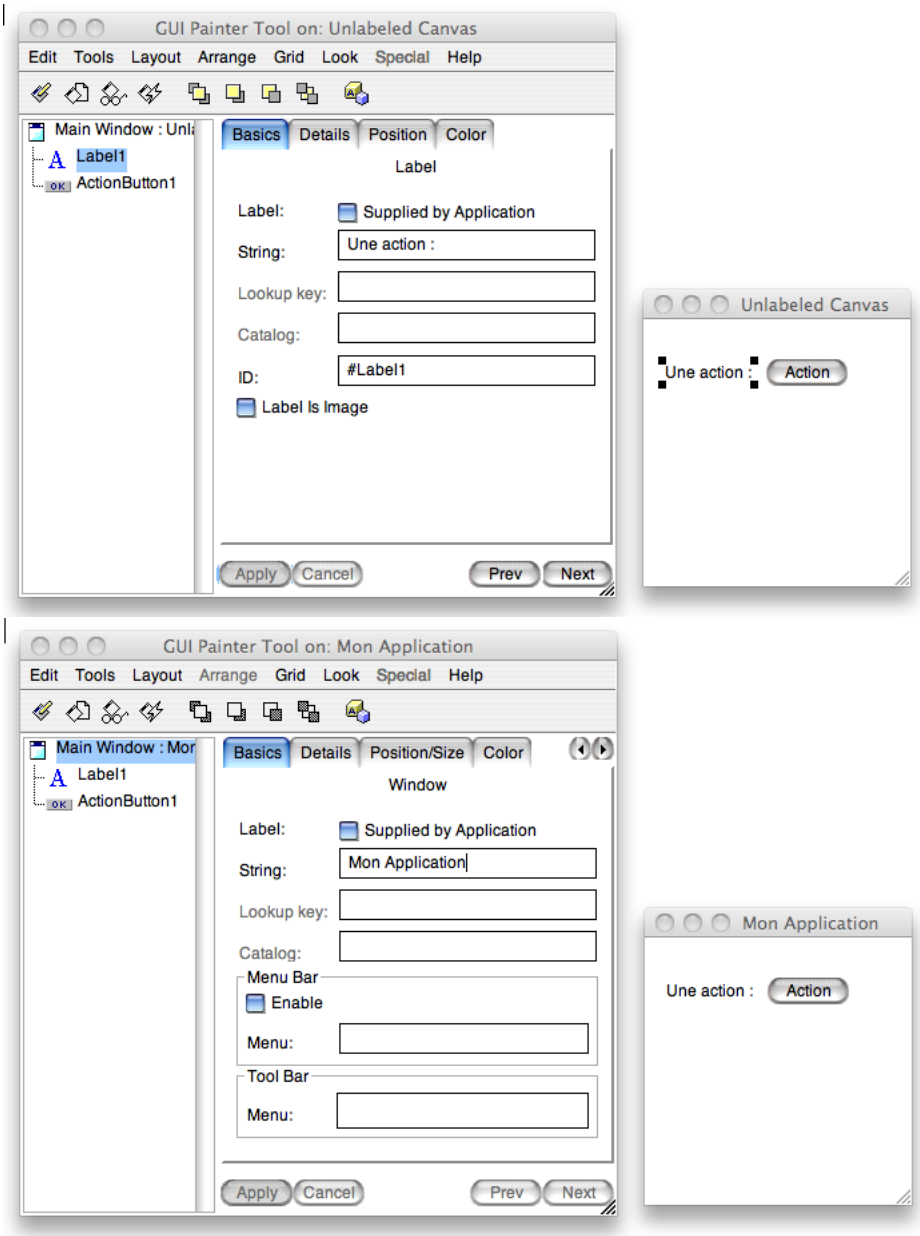


FIGURE 5.10 – Properties change en fonction du composant sélectionné : cas de l'étiquette et cas de la fenêtre elle-même

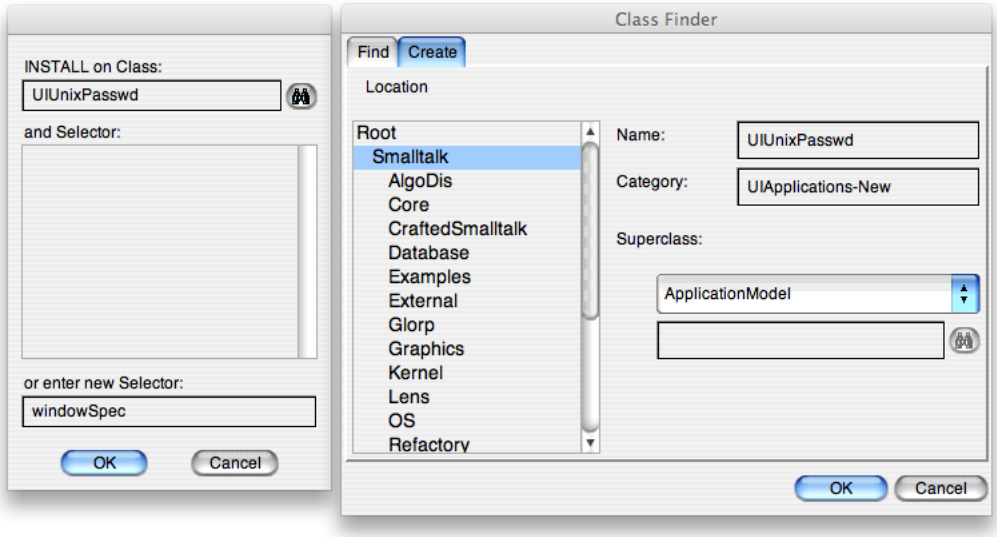


FIGURE 5.11 – Dialogue de création d'une classe interface

On peut lancer un interface via le bouton OPEN de la fenêtre CANVAS TOOL. On obtient un interface pratiquement opérationnel.

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

   <resource: #canvas>
   ^#(#{UI.FullSpec}
      #window:
      #(#{UI.WindowSpec}
         #label: 'Edit Passwd'
         #bounds: #(#{Graphics.Rectangle} 840 525 1286 743 ) )
      #component:
      #(#{UI.SpecCollection}
         #collection: #(
            #(#{UI.SequenceViewSpec}
               #layout: #(#{Graphics.Rectangle} 3 3 171 215 )
               #name: #List1
etc...
```

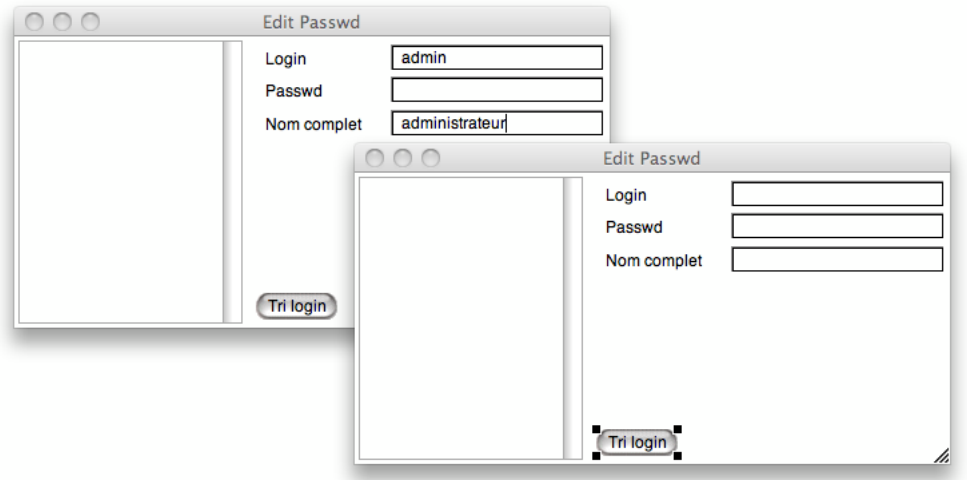


FIGURE 5.12 – L'interface que l'on dessine doit s'ouvrir correctement, avant toute intervention sur le code. La figure montre la même fenêtre en cours de dessin, et activée, avec une saisie effectuée.

5.4.3 Couplage d'un widget et d'une méthode

Dès que la classe interface `UIUnixPasswd` est créée et que le canvas est installé, on s'intéresse à l'assemblage du modèle de données et de l'interface dans le BROWSER. Dans le cas d'un champs éditable, d'une liste, d'un éditeur (...), il s'agit d'une variable d'instance et d'une méthode d'accès portant le même nom, et générant un ValueHolder lors de son premier accès.

- Sélectionner le champs LOGIN dans l'éditeur, et faire un DEFINE (pop-up menu, ou icone). Ceci ouvre un dialogue similaire à celui de la figure 5.13.
- Ouvrir un browser, chercher la classe `UIUnixPasswd`
- Chercher dans la catégorie de méthodes `aspects`, le sélecteur `leLogin`.
- le code par défaut qui vous est présenté est présenté ci-dessous :

```
leLogin
   "This method was generated by UIDefiner.  Any edits made here
   may be lost whenever methods are automatically defined.  The
   initialization provided below may have been preempted by an
   initialize method."

   ^leLogin isNil
      ifTrue:
         [leLogin := String new asValue]
```

```
ifFalse:
   [leLogin]
```

Dans le cas d'un bouton action, define produit une méthode dont l'effet est nul. On y mettra le code utile en lieu et place du *stub* produitr.

- Sélectionner le bouton doTriLogin dans l'éditeur, et faire un DEFINE pour definir la méthode interface. Le dialogue de la figure **??** s'ouvre.
- Ouvrir un browser, chercher la classe `UIUnixPasswd`
- Chercher dans la catégorie de méthodes `action`, le sélecteur `doTriLogin`
- Modifier le code par défaut qui vous est présenté, et tester :

```
doTriLogin

   "This stub method was generated by UIDefiner, then modified"

   Dialog warn: 'C''est doTriLogin'
```

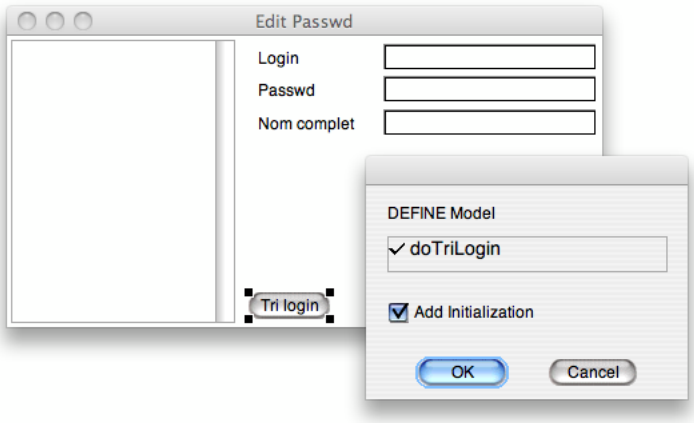


FIGURE 5.13 – Dialogue de définition d'une méthode d'action pour un bouton

5.5 L'exemple de la bibliothèque

5.5.1 modèles de données

On se donne deux classes représentant les fonctionalités d'une bibliothèque :

- la classe `Ouvrage` représente un document ayant un auteur, un titre, une années de parution et un numéro d'enregistrement (variables d'instances de `Ouvrage`)
- la classe `Bibliothèque` représente une collections d'`Ouvrage`.

On sait construire un `Ouvrage` à partir d'une repésentation textuelle, et on sait créer une `Bibliothèque` à partir d'un fichier.

5.5.2 interface sur ces données

L'interface est implanté dans la classe `UIBibliotheque`. En première approche, on y installe les composants suivants :

- `listeDeMots` une liste présentant, par exemple la liste des auteurs.
- `nomDuFichier` un champs éditable (input field) présentant le nom du fichier présenté dans l'interface.
- `nomDuTitre` et `annee` deux champs éditables présentant deux aspects d'un ouvrage.

5.5.3 schéma de dépendances

La figure 5.14 présente une vue d'un interface simple pour Bibliotheque. On reconnait les deux variables d'instances référençant le modèle, `bibli` et `nomBibli`. Le système de dépendances est représenté par des flèches, qui sont essentiellement programmées dans la méthode `initialize` de `UIBibliotheque`.

On peut noter que `bibli` est un `ValueHolder` et que *toute modification externe sur bibli provoque une mise à jour automatique de l'interface.*

Par exemple, on peut :

- ouvrir un inspecteur sur `bibli` à la fin de `initialize`,
- programmer une modification sur un champs d'ouvrage à partir de cet inspecteur,
- remettre à jour le `ValueHolder` par : `self value: self value`,
- constater que l'interface réagit naturellement.

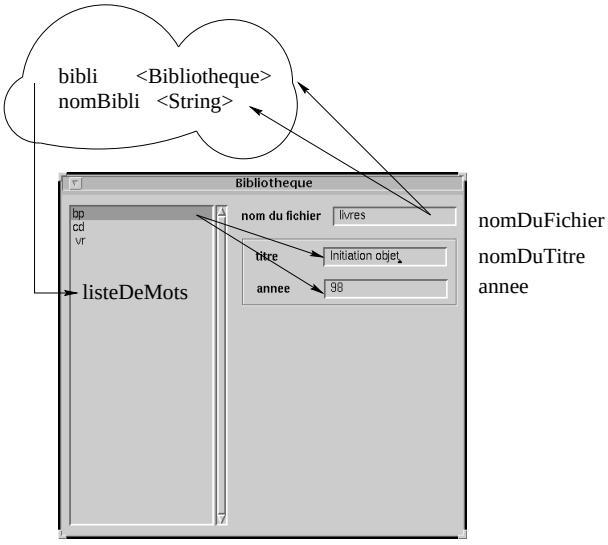


FIGURE 5.14 – Interface simple pour Bibliotheque. Les flèches représentent les dépendances établies dans la méthode `initialize` par des appels à `onChangeSend: to:`

Bibliotheque

class name	**Bibliotheque**
superclass	**Object**
instance variable names	**collection**
class variable names	*none*
pool dictionaries	*none*
category	**Exercices**

Protocol for interrogation

auteurs
```
^self collection collect : [ :ouvrage | ouvrage auteur]
```

parAuteur : unAuteur
```
^self collection select : [ :ouvrage | ouvrage auteur = unAuteur].
```

parTitre : unMot
```
^self collection select : [ :ouvrage | (ouvrage titre tokensBasedOn : $ ) includes :
unMot].
```

Protocol for reading

readFile : aFile
```
| text |
text := UnixProcess cshOne : 'cat ', aFile.
```

```
    self readText : text

readText : text
    | lines ouvrage |
    lines := text tokensBasedOn : Character cr.
    lines := lines reject : [ :line | line size < 4].
    lines
        do :
            [ :line |
            | ligneCassee |
            ligneCassee := line tokensBasedOn : $|.
            ouvrage := Ouvrage
                            auteur : (ligneCassee at : 1)
                            titre : (ligneCassee at : 2)
                            annee : (ligneCassee at : 3)
                            numero : (ligneCassee at : 4).
            self ajouter : ouvrage]
```

Protocol for accessing

collection

```
    ^collection
```

collection : aValue

```
    collection := aValue
```

Protocol for printing

printOn : aStream

```
    self collection
        do :
            [ :livre |
            livre printOn : aStream.
            aStream cr]
```

Protocol for ajout

ajouter : unOuvrage

```
    self collection add : unOuvrage
```

enlever : unOuvrage

```
    self collection remove : unOuvrage
```

Bibliotheque class

class name	**Bibliotheque class**
superclass	**Object class**
instance variable names	*none*
class variable names	*none*
pool dictionaries	*none*
category	**Exercices**

Protocol for instance creation

lireFichier : fichier

```
    "Bibliotheque lireFichier : 'livres"'

    | newBibli nomDuFichier fichierStream |
    newBibli := self new.
    nomDuFichier := fichier asFilename.
    nomDuFichier exists ifFalse : [^self error : fichier , ' introuvable'].
    fichierStream := nomDuFichier readStream.
    [fichierStream atEnd]
        whileFalse :
            [| ligne |
            ligne := fichierStream upTo : Character cr.
              ligne isEmpty ifFalse : [newBibli ajouter : (Ouvrage lireLigne :
ligne)]].
    ^newBibli
```

new

```
    ^self basicNew collection : OrderedCollection new
```

UIBibliotheque

class name	**UIBibliotheque**
superclass	**ApplicationModel**
instance variable names	**bibli nomBibli listeDeMots nomDuFichier nomDuTitre annee**
class variable names	*none*
pool dictionaries	*none*
category	**Exercices**

Protocol for initializing

initialize

"nomDuFichier est un ValueHolder et a une valeur par defaut"

```
nomBibli := 'livres'.
self nomDuFichier value : nomBibli.
self nomDuFichier onChangeSend : #changementDeFichier to : self.
```

"bibli est un ValueHolder initialise avec une Bibliotheque extraite du fichier.
lorsque cette bibliotheque change, l'interface est notifiee par #changedBibli"

```
self bibli : (Bibliotheque lireFichier : self nomBibli) asValue.
self bibli onChangeSend : #changedBibli to : self.
```

"la liste listeDeMots doit presenter les auteurs de la bibliotheque"

```
self listeDeMots list : self bibli value auteurs.
self listeDeMots selectionHolder onChangeSend : #changedAuteur to : self.
```

Protocol for accessing

bibli

```
^bibli
```

bibli : aValue

```
bibli := aValue
```

nomBibli

```
^nomBibli
```

nomBibli : aValue

```
nomBibli := aValue
```

Protocol for changes

changedAuteur

```
| ouvrage |
self listeDeMots selection isNil
    ifTrue :
        [self annee value : ''.
        self nomDuTitre value : '']
    ifFalse :
        [ouvrage := (self bibli value parAuteur : self listeDeMots selection)
first.
        self annee value : ouvrage annee.
        self nomDuTitre value : ouvrage titre]
```

changedBibli

```
self listeDeMots list : self bibli value auteurs
```

changementDeFichier

```
| nom |
nom := self nomDuFichier value.
(nom asFilename exists and : [nom asFilename isReadable])
    ifTrue : [self nomBibli : nom]
    ifFalse : [^self nomDuFichier value : self nomBibli].
self bibli value : (Bibliotheque lireFichier : nom)
```

Protocol for aspects

annee

"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method."

```
^annee isNil
    ifTrue :
        [annee := String new asValue]
    ifFalse :
        [annee]
```

listeDeMots

"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method."

```
^listeDeMots isNil
    ifTrue :
        [listeDeMots := SelectionInList new]
    ifFalse :
        [listeDeMots]
```

nomDuFichier

"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method."

```
^nomDuFichier isNil
    ifTrue :
        [nomDuFichier := String new asValue]
    ifFalse :
        [nomDuFichier]
```

nomDuTitre

"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method."

```
^nomDuTitre isNil
    ifTrue :
        [nomDuTitre := String new asValue]
    ifFalse :
        [nomDuTitre]
```

UIBibliotheque class

class name	**UIBibliotheque class**
superclass	**ApplicationModel class**
instance variable names	*none*
class variable names	*none*
pool dictionaries	*none*
category	**Exercices**

Protocol for interface specs

windowSpec

"UIPainter new openOnClass : self andSelector : #windowSpec"

```
<resource : #canvas>
^#(#FullSpec
    #window :
    #(#WindowSpec
        #label : 'Bibliotheque'
        #bounds : #(#Rectangle 609 495 1108 898 )
        #isEventDriven : true )
    #component :
    #(#SpecCollection
        #collection : #(
            #(#SequenceViewSpec
                #layout : #(#Rectangle 9 12 206 392 )
                #model : #listeDeMots
                #useModifierKeys : true
                #selectionType : #highlight )
            #(#InputFieldSpec
                #layout : #(#Rectangle 337 14 491 37 )
                #model : #nomDuFichier
                #type : #string )
            #(#LabelSpec
                #layout : #(#Point 221 12 )
                #label : 'nom du fichier' )
            #(#InputFieldSpec
                #layout : #(#Rectangle 325 62 479 85 )
                #model : #nomDuTitre
                #type : #string )
            #(#LabelSpec
                #layout : #(#Point 239 60 )
                #label : 'titre' )
            #(#LabelSpec
                #layout : #(#Point 240 98 )
                #label : 'annee' )
            #(#InputFieldSpec
                #layout : #(#Rectangle 326 100 480 123 )
                #model : #annee
```

```
        #type : #string )
    #(#GroupBoxSpec
        #layout : #(#Rectangle 224 52 492 132 ) ) ) ) )
```

Ouvrage

class name	**Ouvrage**
superclass	**Object**
instance variable names	**auteur titre annee numero motcles**
class variable names	*none*
pool dictionaries	*none*
category	**Exercices**

Protocol for accessing

annee

ˆannee

annee : aValue

annee := aValue

auteur

ˆauteur

auteur : aValue

auteur := aValue

motcles

ˆmotcles

motcles : aValue

motcles := aValue

numero

ˆnumero

numero : aValue

numero := aValue

titre

ˆtitre

titre : aValue

titre := aValue

Protocol for printing

printOn : aStream

```
aStream nextPutAll : auteur.
aStream nextPut : $|.
aStream nextPutAll : titre.
aStream nextPut : $|.
aStream nextPutAll : annee.
aStream nextPut : $|.
aStream nextPutAll : numero.
```

Ouvrage class

class name	**Ouvrage class**
superclass	**Object class**
instance variable names	*none*
class variable names	*none*
pool dictionaries	*none*
category	**Exercices**

Protocol for instance creation

auteur : unAuteur titre : leTitre annee : an numero : unNumero
```
    ^(self new) auteur : unAuteur ; titre : leTitre ; annee : an ; numero : unNumero
```

lireLigne : ligne
```
    | flotDeMots |
    flotDeMots := ReadStream on : (ligne tokensBasedOn : $|).
    ^self
        auteur : flotDeMots next
        titre : flotDeMots next
        annee : flotDeMots next
        numero : flotDeMots next
```

Chapitre 6

Usage des images dans MVC

Apprendre à capture, ou lire des images, et apprendre à les utiliser pour composer des présentations graphiques spécifiques dans MVC.
Paquetages : DemoImages et DemoImages2.
Cours : IHM, L2

6.1 Installer et utiliser une image modeste

6.1.1 Capturer une image

1. à partir du canvas, ouvrir l'**Image Editor** (GUI Painter .. Tools .. Image editor)
2. Dans l'**Image Editor**, ouvrir l'outil de capture (Image .. Capture)
3. Sélectionner la zone qui vous intéresse, et cette image sera chargée dans l'éditeur
4. Dans l'**Image Editor**, installer dans votre classe d'application en choisissant un sélecteur (monImage, pour fixer les ides)
5. Vérifier dans un **Browser** que cette image est bien disponible dans les méthodes de classes

6.1.2 Utiliser cette image en tant qu'icone (Label)

1. dans le canvas, installer un composant de type *Label*
2. dans la fenêtre de propriété du label, cocher la case 'Label is image'
3. dans la case *Message*, installer votre sélecteur (*monImage* ici)
4. *Apply*, puis *Install* et vous avez une image visible en tant que *Label*

6.1.3 Dessiner, par programme, une image dans une vue arbitraire

1. Consulter les paquetages de la classe DemoImage donnée en annexe pour fournir un exemple :
   – récupérer l'image en envoyant le message voulu à la classe (*image := MonAppli monImage*)

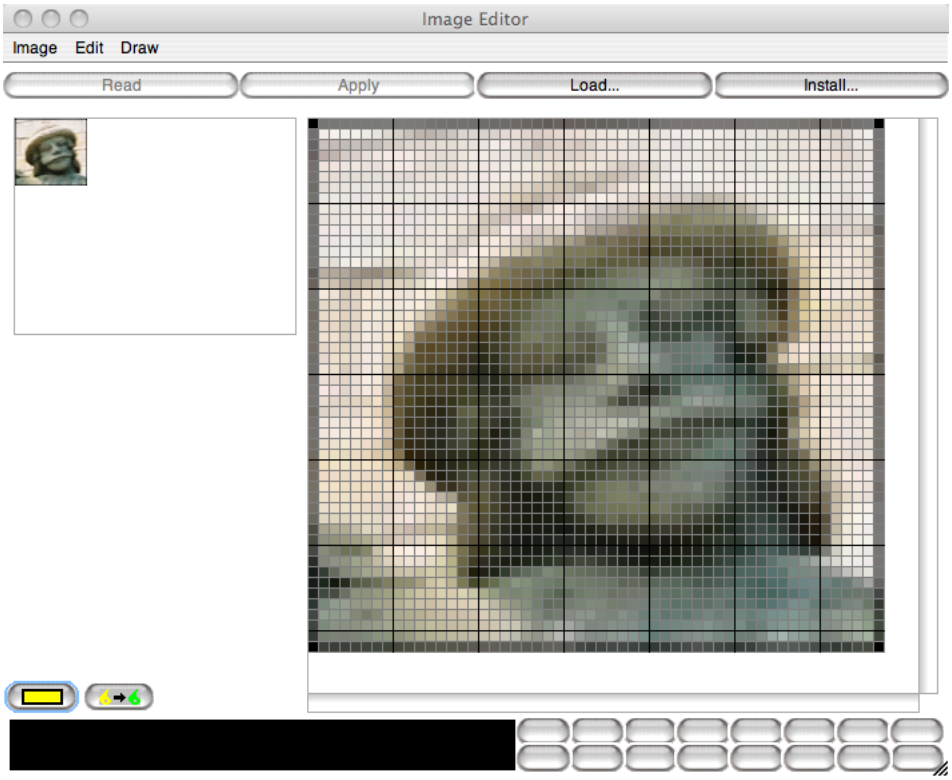


FIGURE 6.1 – Outil de capture d'images

   - dans la méthode *displayOn :* de votre vue, demander un affichage à la position voulue (*image displayOn : aGC at : position*)
2. Ceci fonctionne très bien pour des petites images, genre pièces de jeu, petites photos, etc...

6.2 Lire une image dans un fichier

6.2.1 Charger une image avec ImageReader

- On utilise la classe abstraite *ImageReader*, qui va décider quelle classe concrète utiliser en fonction du fichier. Le résultat est un imageReader (*imr*, ici)

```
fn := Dialog requestFileName: 'Choix fichier d''image'.
imr := ImageReader fromFile: fn.
```

- Cet objet a la capacité de lire le fichier et de restituer une image dans l'environnement Smalltalk, il suffit de lui envoyer le message *image*.

```
image := imr image.
```

- Cette image peut ensuite être affichée dans une vue à l'endroit voulu par *displayOn : aGC at : aPoint*.

6.2.2 Sources des deux démonstrations

Merci de noter que la gestion du modèle associé à la vue est laissé à vos soins,
- dans un cas on passe l'application pour faciliter l'accès aux images,
- dans l'autre on passe un *ValueHolder* qui référence l'image

6.3 Sources : capture et usage d'images

UIDemoImages

class name	**UIDemoImages**
superclass	**ApplicationModel**
instance variable names	**laVue**
class instance variable names	*none*
shared variable names	*none*
imports	*none*
category	**UIApplications-New**

UIDemoImages has a custom view of class DemoImageView. It also has custom screen shots taken from Image editor, used by the view to display faces.

B.Pottier Mars 2009

Instance Variables :
laVue *<View>* *laVue*

Protocol for initialize-release

initialize

```
laVue := DemoImageView new.
laVue model : self
```

Protocol for accessing (has been shortened)

UIDemoImages class

class name	**UIDemoImages class**
superclass	**ApplicationModel class**
instance variable names	*none*
class instance variable names	**thisClass**
shared variable names	*none*
imports	*none*
category	**UIApplications-New**

Protocol for interface specs (has been shortened)

Protocol for resources (has been shortened)

DemoImageView

class name	**DemoImageView**
superclass	**View**
instance variable names	*none*
class instance variable names	*none*
shared variable names	*none*
imports	*none*
category	*none*

DemoImageView shows how to embed arbitrary images in a custom view.
B.Pottier Mars 2009

Protocol for initialize-release

initialize
"Initialize a newly created instance. This method must answer the receiver."

```
super initialize.
" *** Replace this comment with the appropriate initialization code *** "
^self
```

Protocol for displaying

displayOn : aGC

```
| bounds tete1 tete2 pos vPos |
bounds := self container bounds.
tete1 := self model class tete1.
tete2 := self model class tete2.
vPos := tete1 extent y max : tete2 extent y.
pos := 2 @ 2.
[pos y < bounds height]
    whileTrue :
        [[pos x < bounds width]
            whileTrue :
                [tete1 displayOn : aGC at : pos.
                pos x : pos x + tete1 extent x + 2.
                tete2 displayOn : aGC at : pos.
                pos x : pos x + tete2 extent x + 2].
        vPos := tete1 extent y max : tete2 extent y.
        pos := 2 @ (pos y + vPos).
        [pos x < bounds width]
            whileTrue :
                [tete2 displayOn : aGC at : pos.
                pos x : pos x + tete2 extent x + 2.
                tete1 displayOn : aGC at : pos.
                pos x : pos x + tete1 extent x + 2].
        pos := 2 @ (pos y + vPos)]
```

FIGURE 6.2 – Vue présentant un damier d'images capturées

6.4 Sources : lecture et usage d'images externes

UIDemoImageReader

class name	**UIDemoImageReader**
superclass	**ApplicationModel**
instance variable names	**laGrandeVue**
class instance variable names	*none*
shared variable names	*none*
imports	*none*
category	**UIApplications-New**

UIDemoImageReader shows how to display en external image in a view.
Bernard Pottier −− Mars 2009

Instance Variables :
laGrandeVue *<DemoImageReaderView>* *là ou on affiche laGrandeVue*

Protocol for initialize-release

initialize
"On définit notre vue spécifique"

```
self laGrandeVue : DemoImageReaderView new.
"définit un valueholder en tant que modèle et le transfère à la vue"
self laGrandeVue model : nil asValue
```

newBackground

```
| fn imr image |
fn := Dialog requestFileName : 'Choisir un fichier d'image'.
imr := ImageReader fromFile : fn.
image := imr image.
self laGrandeVue model value : image
```

Protocol for accessing (has been shortened)

Protocol for actions

loadFile

```
self newBackground
```

UIDemoImageReader class

class name	**UIDemoImageReader class**
superclass	**ApplicationModel class**
instance variable names	*none*
class instance variable names	**thisClass**
shared variable names	*none*
imports	*none*
category	**UIApplications-New**

Protocol for interface specs (has been shortened)

Protocol for resources (has been shortened)

DemoImageReaderView

class name	**DemoImageReaderView**
superclass	**View**
instance variable names	*none*
class instance variable names	*none*
shared variable names	*none*
imports	*none*
category	*none*

DemoImageReaderView est un simple afficheur sur une image externe chargée par ImageReader.

Protocol for displaying

displayOn : aGC
"rien de plus simple, on demande à l'image de s'afficher"

```
| image |
image := self model value.
image isNil ifFalse : [image displayOn : aGC]
```

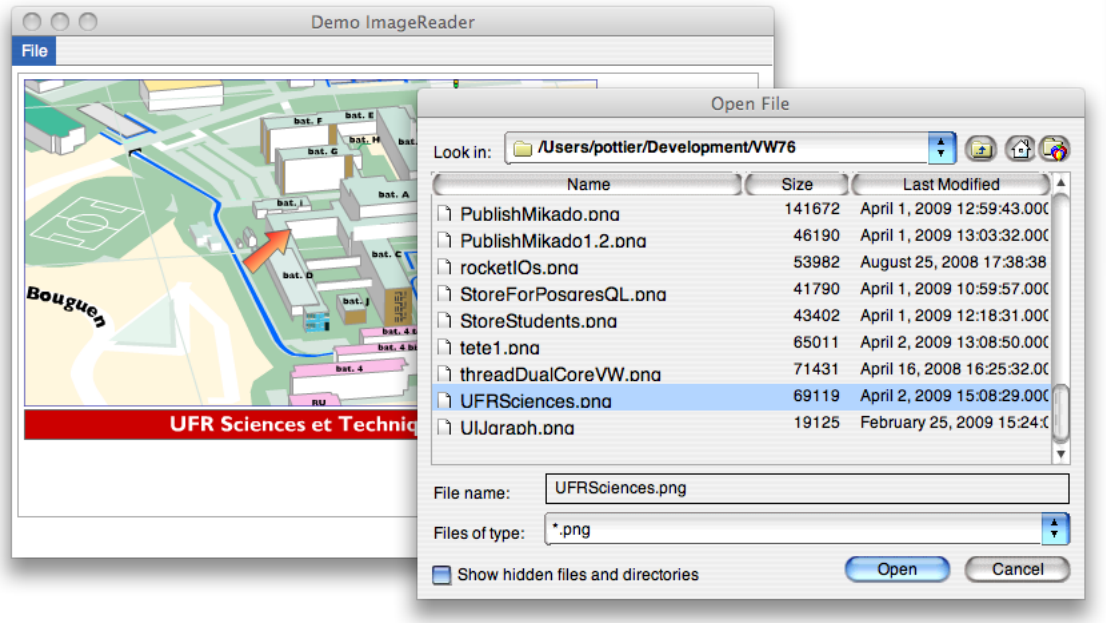


FIGURE 6.3 – UIDemoImageReader : présentation de fichiers images

Troisième partie

Outils de développement

Chapitre 7

Mise au point des programmes

Ce guide se propose d'aider la prise en main du dévermineur (debugger) dans l'environnement Smalltalk VisualWorks. Il décrit la structure et les fonctions de cet outil, puis en montre les effets sur une petite séquence de code. Le dévermineur est très puissant, et incontournable pour l'efficacité des développements ainsi que la compréhension du langage et de ses classes.

Les vues et séquences ont été opérées sur une plateforme MacOSX, mais les explications valent pour les autres plateformes.

7.1 Accès au dévermineur

Cet outil permet de retrouver l'état des objets en cause dans une exécution, de modifier éventuellement leurs valeurs, d'observer les progrès d'un algorithme. Sa manipulation aide donc à la compréhension de l'environnement de programmation, et donc à programmer efficacement.

Les dévermineurs s'ouvrent :

- lors d'une erreur : une division par zéro, un message non compris, un accès illégal dans une collection, etc...
- par décision du programmeur, qui place un point d'arrêt en insérant l'instruction `self halt.` à l'endroit qui l'intéresse.

Dans l'exemple du fragment de code qui suit, il y a une faute de frappe sur le premier message `value`, qui est devenu `valuer`, ce qui amène une erreur (figure 7.1) à l'exécution (message non-compris).

```
changeRadius

| center leRayon cercle |
cercle := self leCercle valuer.
leRayon := self radius value.
center := cercle center.
cercle := Circle center: center radius: leRayon.
self leCercle value: cercle
```

Parmi les options offertes par le dialogue d'erreur, on peut remarquer :

Debug qui va nous permettre d'investiguer le problème,

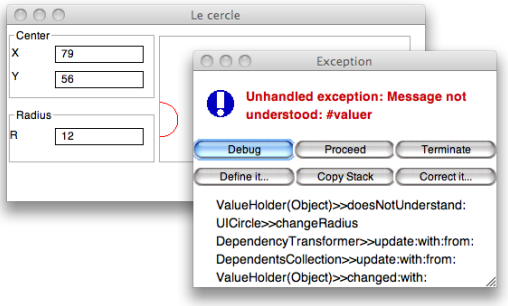


FIGURE 7.1 – Les dialogues d'erreur proposent d'examiner l'état de l'exécution et de corriger une faute de programmation. Ici il s'agit d'une erreur courante dûe à un message non implémenté par un objet.

Proceed qui permet de remettre le programme en activité par exemple en cas d'absence d'intérêt pour un pont d'arrêt.

Terminate pour renoncer à l'exécution courante,

Correct it pour demander au système de proposer une correction à une erreur (un message mal orthographié, probablement).

7.2 Anatomie d'un dévermineur

Prenons la première option *Debug*, et observons la fenêtre qui s'ouvre (figure 7.2).

Cette fenêtre a une structure hiérarchique, dont les blocs élémentaires seront présentés dans les sections qui suivent. La liste du haut (figure 7.3). présente les différents messages suspendus. En général, on trouve à son niveau supérieur le message en erreur, ou un point d'arrêt, et quelque part en dessous, le message initial de lancement de l'exécution.

7.2.1 Présentation des contextes, et contexte courant

Pour visualiser la chaine des appels procéduraux, qui sont ici les messages en cours d'exécution, on déplace le curseur verticalement dans la liste. Il est ainsi possible de trouver le bon niveau de détails pour le problème traité, et de saisir précisément un état de l'exécution en recalant l'observation au bon niveau.

Chaque sélection ou déselection dans la liste des contextes provoque le rafraichissement de la fenêtre présentant le source du message (figure 7.4). On note le surlignement d'un des messages dans ce source, surlignement dont le but est de présenter une approximation du pointeur d'instruction, c'est à dire l'instruction où l'exécution est suspendue.

7.2.2 Inspecteurs liés aux contextes

Lors de la présentation de chaque contexte, deux inspecteurs sont présentés dans le bas du dévermineur. Le premier (figure 7.5) présente les variables d'ins-

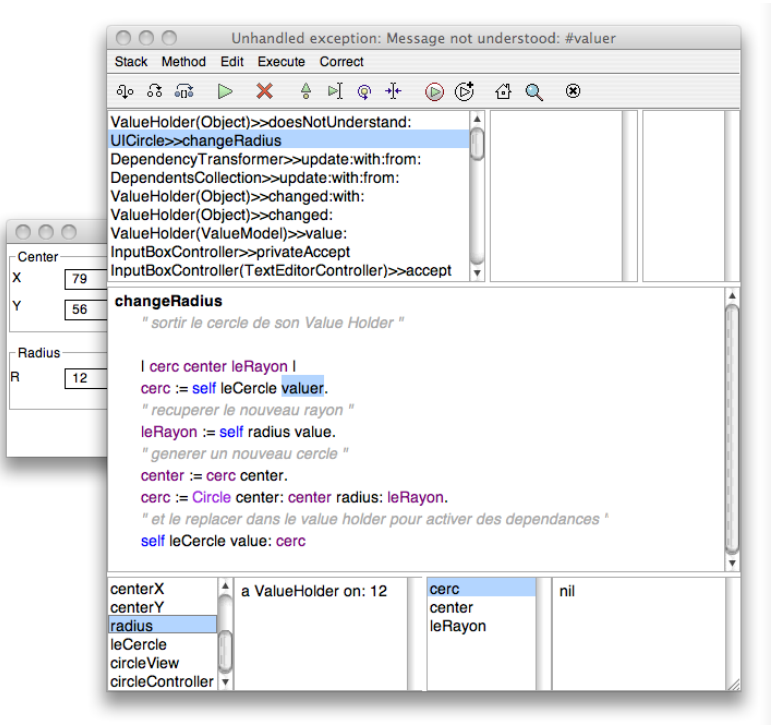


FIGURE 7.2 – Organisation d'un dévermineur. Sous celui-ci on voit l'application en cours de mise au point, l'erreur ayant été introduite dans une méthode appelée par le composant *radius*

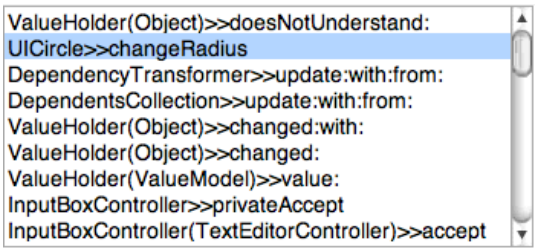


FIGURE 7.3 – Pile des contextes et contexte actif

```
changeRadius
    " sortir le cercle de son Value Holder "

    | cerc center leRayon |
    cerc := self leCercle valuer.
    " recuperer le nouveau rayon "
    leRayon := self radius value.
    " generer un nouveau cercle "
    center := cerc center.
    cerc := Circle center: center radius: leRayon.
    " et le replacer dans le value holder pour activer des dependances "
    self leCercle value: cerc
```

FIGURE 7.4 – Code source du contexte actif et pointeur d'instruction

tances du receveur du message. Le second (figure 7.6) présente les variables locales du contexte. Dans les deux cas, il est possible d'observer une variable en la sélectionnant, et éventuellement de la changer en précisant une nouvelle valeur dans le petit éditeur des inspecteurs, puis en activant lla fonction *accept* du menu.

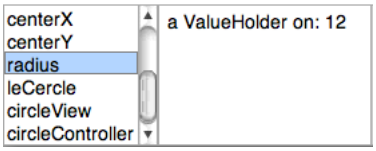


FIGURE 7.5 – Inspecteur sur le receveur d'un message

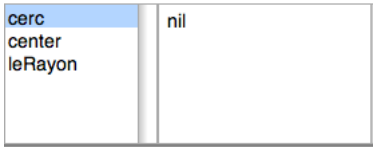


FIGURE 7.6 – Inspecteur sur les paramêtres et variables temporaires dans un contexte

7.2.3 Séquencement de l'exécution

Le dévermineur permet au programmeur d'effectuer l'exécution d'un programme en *pas à pas*. L'intérêt de cette opération est de permettre l'observation de l'exécution des messages sur les variables présentées dans les contextes. La visualisation des changements d'état peut être immédiate. En effet, si on prend soin de sélectionner les variables intéressantes les valeurs présentées dans les inspecteurs sont rafraichies à chaque progrès dans l'exécution.

Dans d'autres cas il peut être souhaitable d'ouvrir des inspecteurs secondaires et de les rafraichir à la main lorsque cela apparaît nécessaire.

Le menu du dévermineur (figure 7.7) propose deux principales options pour le pas à pas, que l'on peut mémoriser par les mnémoniques :

(a) Send : expédition d'un message, et empilement de son contexte. le niveau de la pile augmente.

(b) Step : un pas dans le contexte courant, le niveau de la pile est constant,

Les autres options sont les suivantes :

(c) Step over : pas à pas, sans montrer le détail de l'exécution d'un bloc,

(d) Run : poursuit l'exécution, en abandonnant le dévermineur,

(e) Stop : termine l'exécution, en abandonnant le dévermineur,

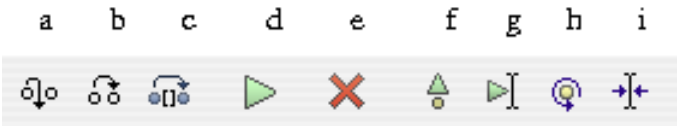


FIGURE 7.7 – Principales options du menu du dévermineur

(f) Return : renvoit une valeur pour ce contexte, en abandonnant son exécution, mais en restaurant l'exécution du message appelant (contexte appelant),

(g) Goto : exécute le code jusqu'au marqueur, sans en montrer les détails,

(h) Redo : recommence l'exécution du contexte courant,

(i) Mark : pose un marqueur dans le code, en vue d'un prochain Goto,

7.3 Illustration : observer et commander une exécution

Le dévermineur est un outil vivant qu'il faut apprendre pratiquement. Il est rare qu'une session de programmation s'effectue sans son usage, et il est fréquent qu'une part significative du temps de développement (20% ?) s'effectue en utilisant ses services.

7.3.1 Compilation, et exécution d'une méthode simple

La section 7.3 présente l'exécution d'une méthode, message par message, et montre comment modifier des valeurs à la volée, ou observer des propriétés.

Afin de disposer d'un exemple, on se donne à comprendre comment une instance de la classe Rectangle peut calculer sa surface. Pour cela on programme un peu de code dans lequel on place un point d'arrêt. On compile et on exécute, et le dévermineur sera ouvert lors de sa rencontre.

```
| monRectangle |
monRectangle := Rectangle origin: 0@0 corner: 20@40.
monRectangle halt.  "ici on s'arretera"
monRectangle area
```

7.3.2 Obtention d'un point d'arrêt, modification de variables

La figure 7.8 montre l'allure du dévermineur lorsque *halt* s'exécute. On est intervenu sur une variable pour changer sa valeur qui passe de 0@0 à 10@10. Chaque contexte contient des valeurs temporaires présentées en haut à droite. Ici il s'agit du rectangle receveur de *halt* dans son état non modifié.

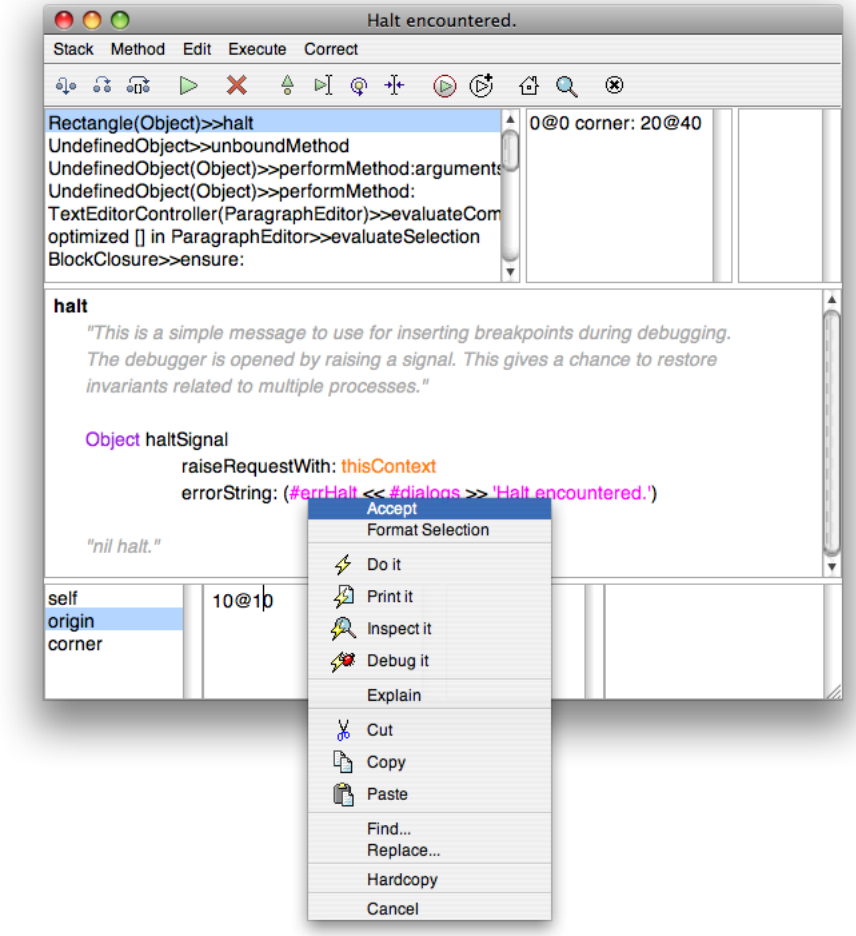


FIGURE 7.8 – Le rectangle a reçu le message halt. L'inspecteur sur self a été utilisé pour modifier l'origine du rectangle et le menu de cet inspecteur permet de valider cette modification avec l'option *accept*.

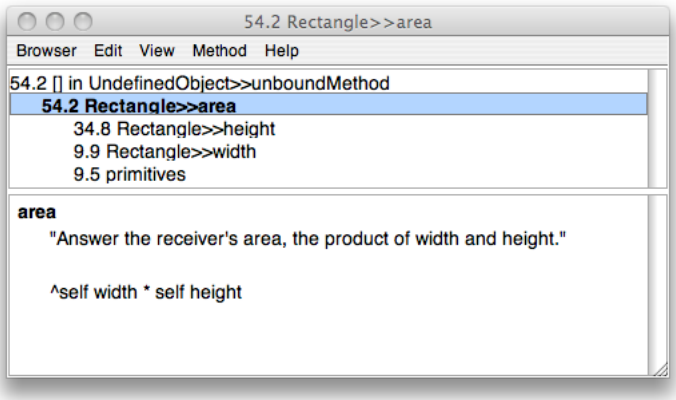


FIGURE 7.9 – Le message *area* est un arbre qui expédie les messages *width* et *height* en effectuant une multiplication au retour. Cette vue est produite par un autre outil : le profileur.

7.3.3 Visite de la pile de contextes

En déplaçant le curseur dans la pile pour visiter le contexte appelant (figure 7.10), on trouve le code source initial : on peut ainsi observer que le rectangle a effectivement été modifié. Une pression sur Step libérera le contexte du point d'arrêt *halt*, préparant l'exécution du message de calcul d'aire *area* (figure 7.11).

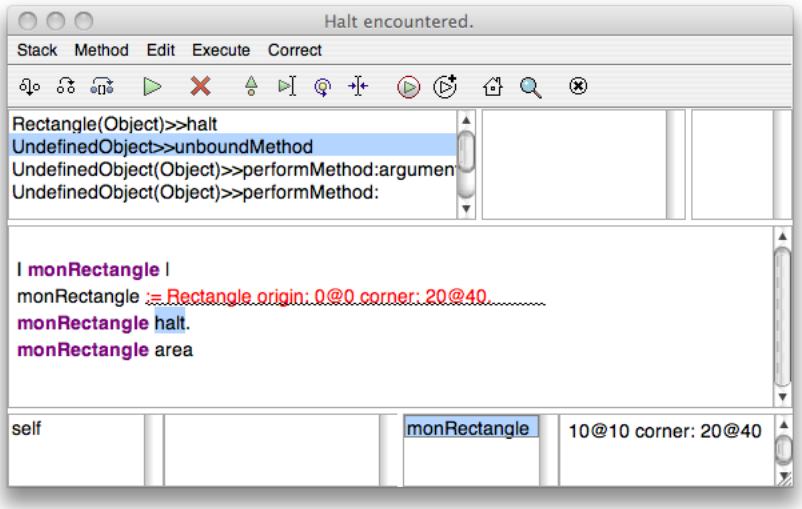


FIGURE 7.10 – Visite de la pile de contexte en déplaçant la sélection courante dans leur liste.

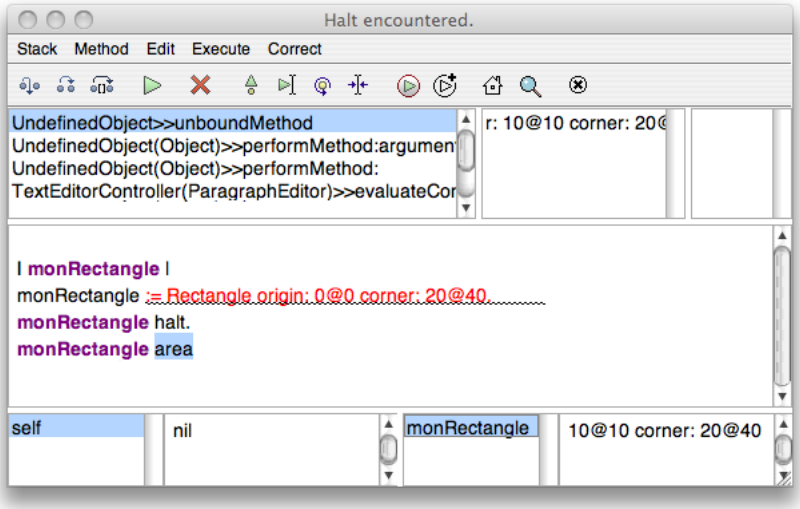


FIGURE 7.11 – Après pression de Step, et avant l'exécution de area qui sera provoquée par un Send. Noter le déplacement du pointeur d'instruction.

7.3.4 Send du message *area*

Un nouveau contexte a été empilé pour la méthode *area* de la classe Rectangle dont la figure 7.12 présente le code. En effectuant un pas à pas (Step), on peut observer l'évaluation de l'expression à niveau constant.

En effectuant un Send sur *width* et *height* lors que le pointeur de programme les visite, on peut aussi observer comment ces messages sont implémentés et comment ils se comportent dans notre cas.

7.3.5 Retour dans une méthode appelante

Après le dernier `Step` dans la méthode `area`, le contexte de cette méthode est dépilé et on revient dans la méthode initiale. Ce pourrait etre l'occasion de réitérer l'exécution en observant sa valeur grace à un *printIt* ou un *inspect*. Ici cette exécution serait inoffensive car ce message ne modifie aucun objet (voir aussi section 7.15).

Il est courant de mener des évaluations dans le débermineur, afin de vérifier des propriétés supposées du programmeur.

7.3.6 Evaluation dans un contexte

La figure 7.15 présente une telle évaluation dont le résultat est en général inspecté. Il est aussi possible de saisir des assertions et de les évaluer, sans modifier pour autant le code original.

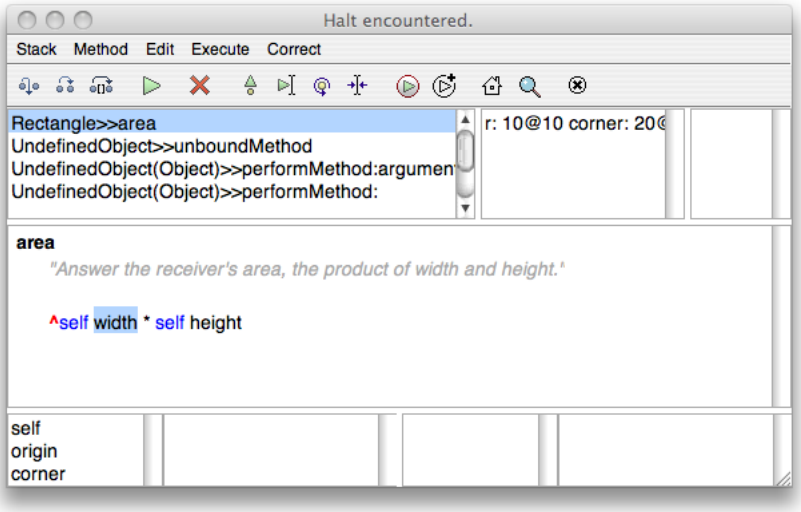


FIGURE 7.12 – Expression de calcul d'une aire : la méthode *area* apparait maintenant en sommet de pile.

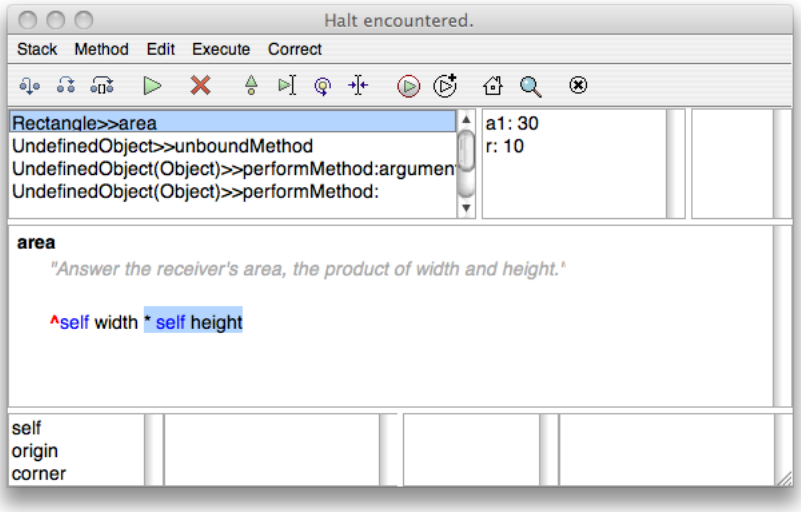


FIGURE 7.13 – Calcul d'une aire. Le message va retourner la valeur sélectionnée.

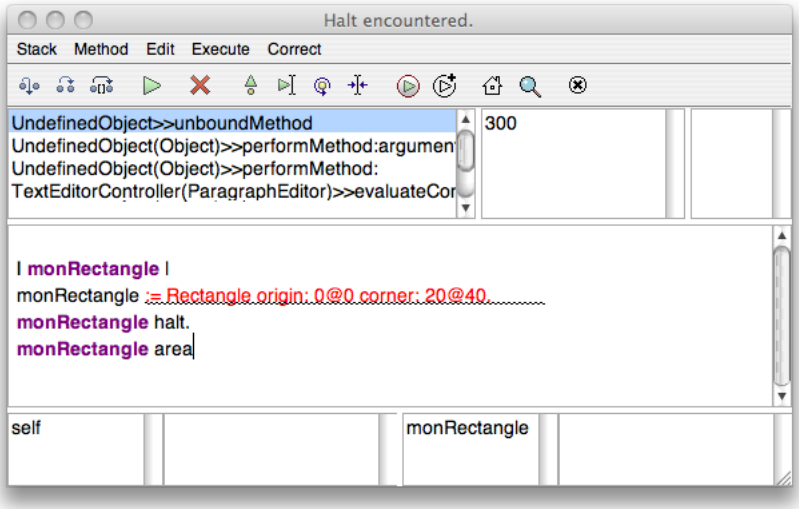


FIGURE 7.14 – Le résultat du calcul de l'aire apparait en sommet de pile en haut à droite. Il s'agit de la surface du rectangle `Rectangle origin: 10@10 corner: 20@40`

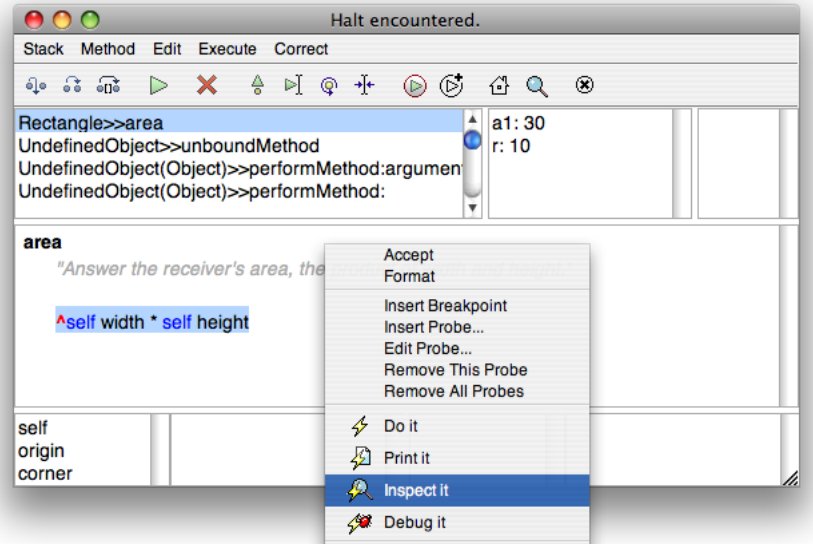


FIGURE 7.15 – Exécution et inspection d'un fragment de code sélectionné dans le dévermineur

7.4 Accès au profileur

Cet outil s'installe avec le paquetage *All advanced tools*. Un profileur (profiler en anglais) est un outil d'analyse du comportement du code. Cet outil est précieux pour améliorer la qualité des algorithmes car il permet de comprendre où un programme passe son temps majoritairement. Cela peut permettre de découvrir que le programme a une mauvaise stratégie algorithmique, où qu'il faut améliorer la qualité du codage de certaines procédures.

Le principe du profileur est d'exécuter le programme avec une interruption périodique. Lors de cette interruption, on récolte l'état du programme :

- pointeur d'instruction : que faisait le programme au moment de l'interruption,
- pile des procédures en exécution : quel était exactement les appels procéduraux

A la fin de l'exécution, les informations récoltées sont placées dans un fichier, ou dans le cas de Smalltalk, dans un objet.

On passe alors à l'analyse de cette récolte en présentant les statistiques d'utilisation. Bien évidemment, la récolte change lorsque les données changent.

Ce processus de développement est accessible avec les compilateurs C, en compilant les programmes avec l'option -p, en exécutant, puis en analysant la trace :

```
cc -p xx.c -o xx
./xx
ls -l gmon.out
gprof xx gmon.out
```

En Smalltalk, il existe plusieurs manière de démarrer un profilage de code.

Chapitre 8

Partager et gérer les développements

> Store est un dépot de sources Smalltalk organisé autour d'une base de données. Nous mettons à votre disposition une base publique qui permet d'accéder à des exemples, ou de partager des développements, directement à partir des images VisualWorks.

8.1 Connexion à Store

Pour accéder aux sources publics, ou ceux de votre projet, vous devez d'abord configurer quelques détails dans votre image.

8.1.1 Configuration de l'image

Il faut ainsi charger un paquetage permettant d'accéder à une base de données Postgres, puisque c'est celle-ci qui est utilisée ici (figure : 8.1).

- Chercher *Load parcels named* dans votre menu System
- Charger *StoreForPostgreSQL*

Dès que vous avez effectué cette opération, sauvez votre image pour être sûr de garder la fonctionalité d'une session sur l'autre. Pour mémoire, la documentation de référence sur Store est disponible dans toutes les installations avec le chemin : $VISUALWORKS/doc/SourceCodeMgmtGuide.pdf.

8.1.2 Connexion à une base de données

Vous devez maintenant voir un menu Store dans votre console Visualworks. En ouvrant ce menu (figure 8.2), il est possible de se connecter à un serveur de logiciels. On vous offre par défaut, celui du distributeur de VisualWorks (figure 8.3).

La fenêtre de connexion définit de manière unique la base sur laquelle travailler :

- nom ou adresse IP du serveur
- port sur le serveur, par défaut celui qui était utilisé par PostgreSQL : 5432
- le nom de la base sur ce serveur.

On trouve ensuite les données de connexion, utilisateur et mot de passe, et enfin, il est possible de sauvegarder la connexion pour les sessions ultérieures.

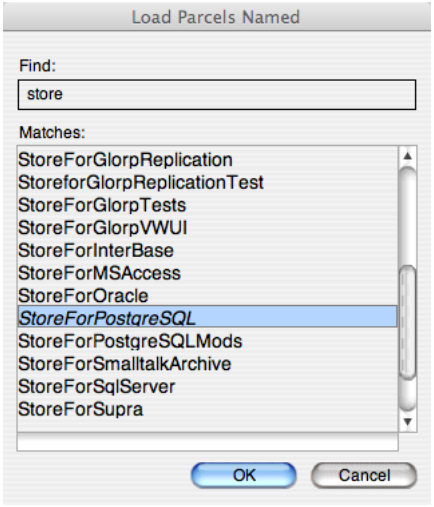


FIGURE 8.1 – chargement de l'interface vers PostgreSQL

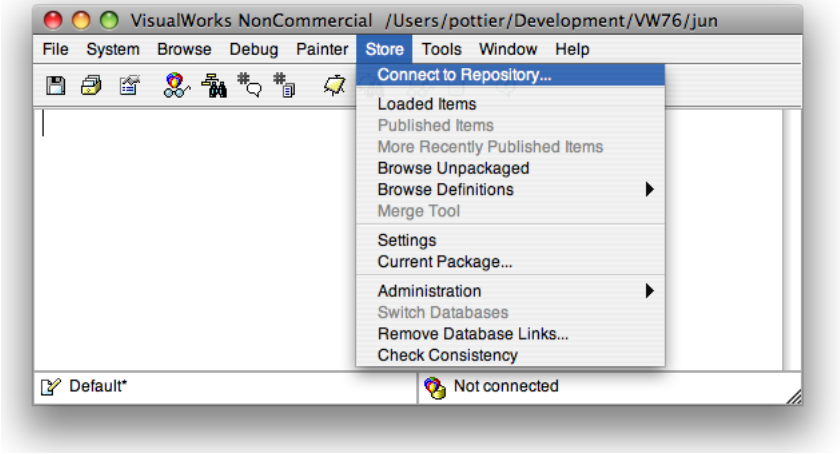


FIGURE 8.2 – Connexion à une base Store

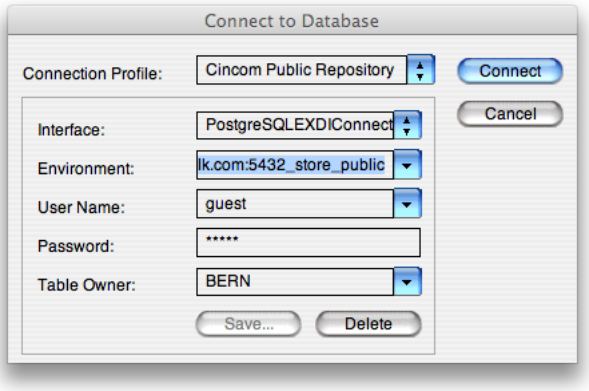


FIGURE 8.3 – Connexion à Cincom. Noter la structure du champ environnement qui est de la forme machineIP :port_nomBase.

8.1.3 Connecter à la base students

Cette base est accessible à l'UBO, et contient les logiciels de la distribution officielle de Visualworks, et les logiciels distribués dans le cadre de cours, de projets, plus ceux que vous y déposez, éventuellement (figure 8.4).

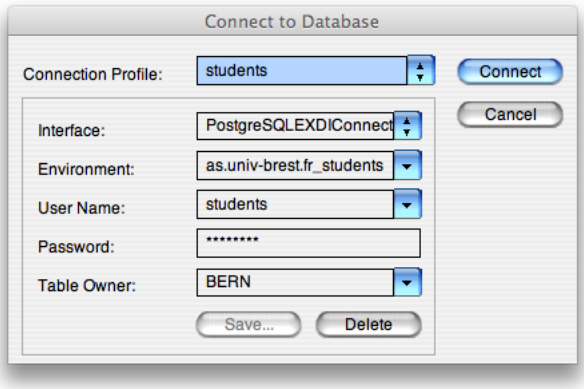


FIGURE 8.4 – Connexion à la base students, sur le port par défaut. La clé d'accès est le nom de l'utilisateur visible dans la fenêtre

8.2 Télécharger et publier

L'accès à Store est en général systématiquement effectué en début et fin de session, la sauvegarde en base de données étant une garantie pour la gestion des évolutions des sources. Les bases sont organisées par fréquence d'accès, il y a ainsi

celles qui supportent les développements, celles qui permettent le partage de distribution de tests, et celles fiabilisées, qui fournissent des logiciels stables, sous le contrôle d'une communauté.

Dans notre cas, un bon projet pourra être publié, ou des développements coopératifs peuvent utiliser les services de Students.

8.2.1 Téléchargements

Après une connexion à Students, il est possible de télécharger un paquetage en le cherchant dans la liste des logiciels publiés (figure 8.5).

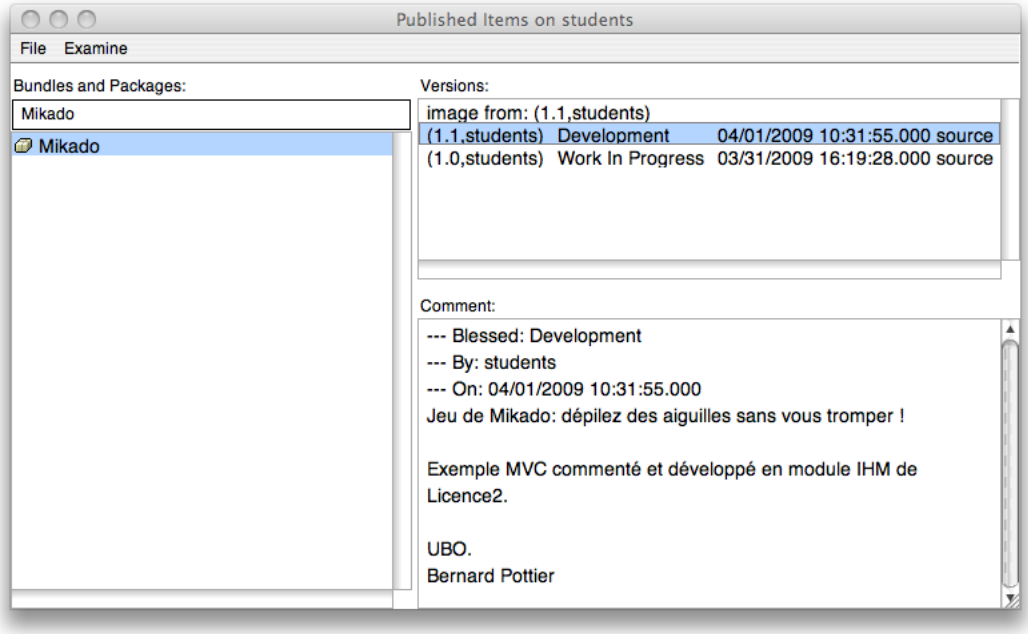


FIGURE 8.5 – Après connexion, et sélection de *Load ..* dans le menu Store, il suffit en général de taper quelques caractères pour repérer le bon paquetage.

Après sélection du paquetage, la fenêtre de droite présente les versions enregistrées. Si l'image est à jour, la première ligne présente la version implantée dans celle-ci. Ici il s'agit du paquetage Mikado, disponible dans l'image, et la version est 1.1.

Il serait possible de revenir en version 1.0, en rechargeant cette ancienne version. Il est aussi possible d'accéder à cette version 1.0, en la brossant en ligne (browse) afin de retrouver des détails d'implémentation abandonnés, par exemple.

8.2.2 Publications

La publication consiste à déposer un paquetage sur le serveur. C'est un acte simple, mais qui requiert une organisation et un effort préalable par rapport à la gestion des sources. Parmi ces efforts, mentionnons :

– l'organisation des classes en paquetages (parcels),
– l'organisation des paquetages en groupes (bundles),
– la spécification des dépendances entre ces catégories, l'ordre de chargement, les pré-requis (autres parcels, code à éxécuter).
– les commentaires, l'identification des progrès effectués.

Dès lors que ces efforts ont été accomplis, la publication et le rechargement des logiciels s'effectuent en quelques clics, avec tous les bénéfices de Store.

La publication de vos développements s'effectue dans le Browser standard qui intègre les fonctions de Store. Il suffit simplement de sélectionner un paquetage et de le publier (*Publish*, figure 8.6), en remplissant soigneusement la fiche de stockage (figure 8.7).

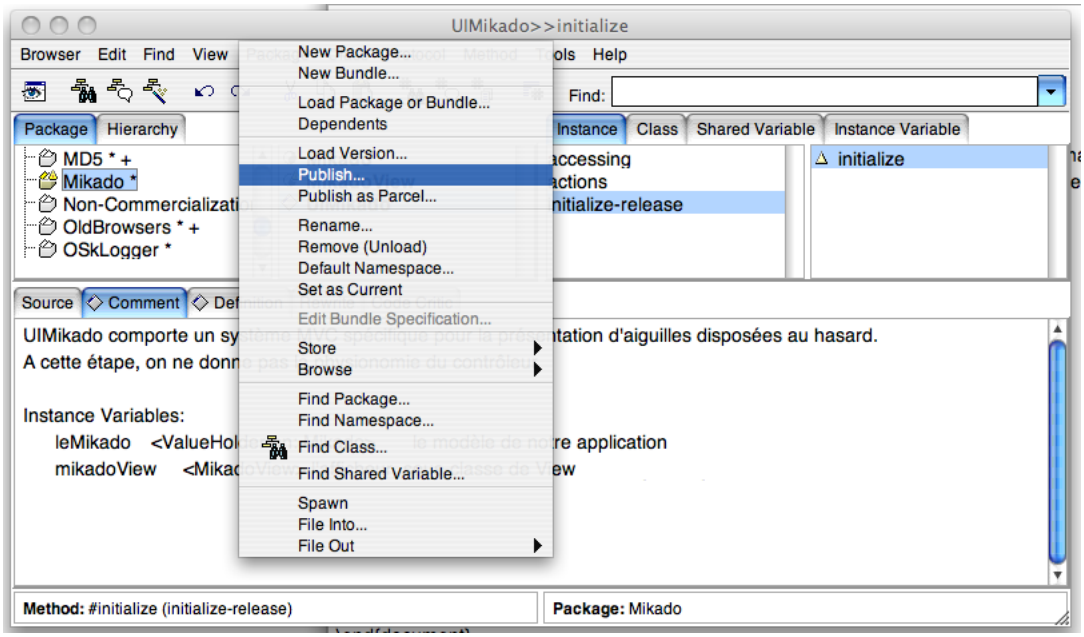


FIGURE 8.6 – Le Browser permet de publier dans la base courante, connexion établie

La publication étant effectuée, il est facile de partager, ou observer les évolutions positives ou négatives du logiciel. En particulier, on peut repartir d'une image vierge conditionnée pour Store à chaque début de session de développement, et l'usage de la base est bien plus facile que celle de fichiers de paquetages (parcels).

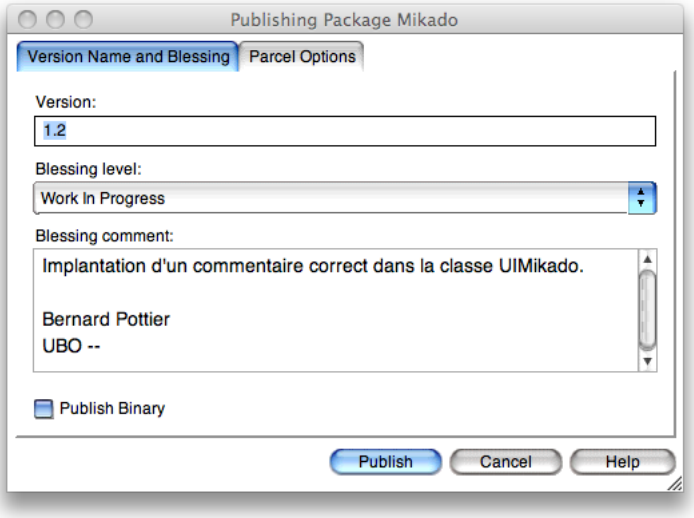


FIGURE 8.7 – L'interface Store incrémente automatiquement le numéro de version et propose de classer le développement selon différents degrés de finalisation.

Chapitre 9

Conditionnement d'une application

9.1 Création d'une image de déploiement

9.1.1 Préparation d'un répertoire de génération

Voir figure 9.1 Il faut commencer par créer un répertoire dans lequel vous placerez les parcels que vous comptez intégrer à vos applications. Exemple : `C:deploiement`. nettoyer ce répertoire si il existe déjà.

Assurez vous que vous avez de la place sur le disque, une trentaine de Megaoctets est un minimum pour etre tranquille, car il faut pouvoir générer deux images.

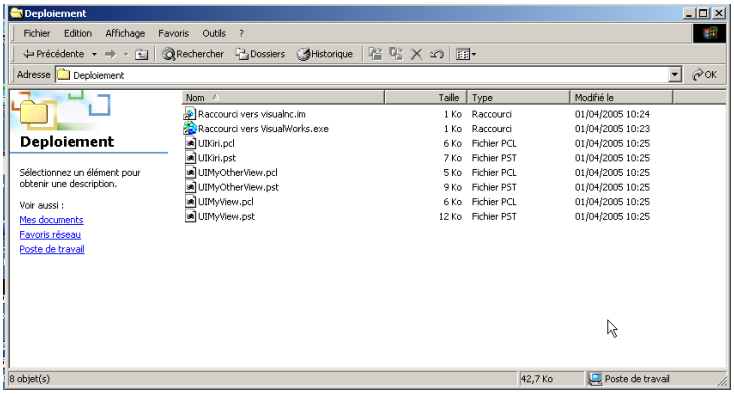


FIGURE 9.1 – (dirInitial-1.eps)

9.1.2 Création d'une image vierge

Voir figure 9.2 Positionnez vous dans votre répertoire de déploiement et démarrez une image d'origine, sans aucun développement, et commencez par la sauvegarder dans le répertoire local. Ici on spécifie le chemin complet afin de s'assurer de la création de l'image runtime.im dans notre répertoire.

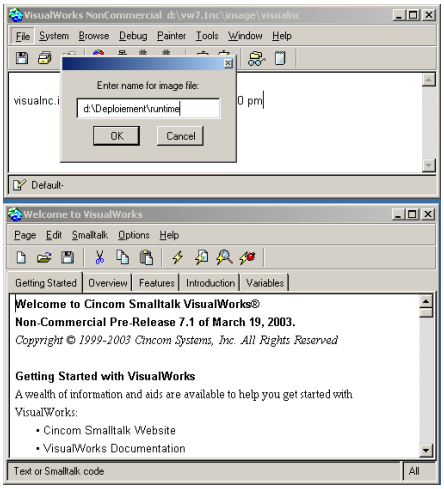


FIGURE 9.2 – (imageLocale-2.eps)

9.1.3 Configuration de l'image

Voir figure 9.3 Modifier les 'settings' de manière à ce que les chemins d'accès aux parcels comportent le répertoire local (saisie, puis add, puis apply).

9.1.4 Chargement des parcels

Voir figure 9.4 Il faut maintenant charger les parcels de votre application (UIMyView, UIKiri...), et celui de l'outil RuntimePackager. On utilise 'load parcel' dans le menu tools, on saisit les noms complets ou on laisse chercher les noms en placant une '*' dans le dialogue de saisie.

Noter les messages de chargements dans la fenetre Visualworks.

9.1.5 Premier test de l'application

Voir figure 9.5 Lancer l'application pour vérifier qu'elle est bien fonctionnelle. Ici, on a chargé UIMyView. Fermer cette fenetre après le test. Fermer toutes les autres fenetres excepté Visualworks

9.1.6 Lancement du RuntimePackager

Voir figure 9.6 A partir du menu 'Tools' ouvrir le RuntimePackager qui est la dernière option proposée. Vous voyez arriver la fenêtre suivante (9.6), qui représente un pas à pas en 8 étapes.

9.1.7 Etapes 'clean up' et 'set options'

Voir figure 9.7 Pour faire avancer le pas à pas on effectue un 'do this step', puis un 'next'.

– La première étape effectue un nattoyage de l'image. Fermer simplement les notifications qui sont présentées.

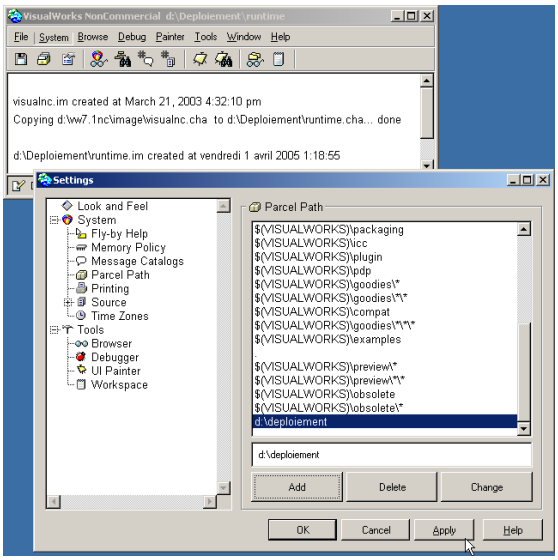


FIGURE 9.3 – (settingsParcelPath-3.eps)

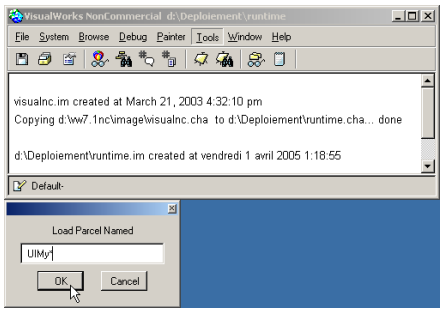


FIGURE 9.4 – (loadParcel-4.eps)

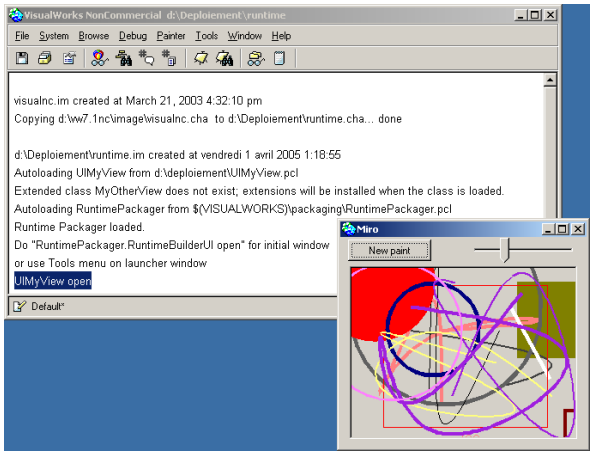


FIGURE 9.5 – (testAppli-5.eps)

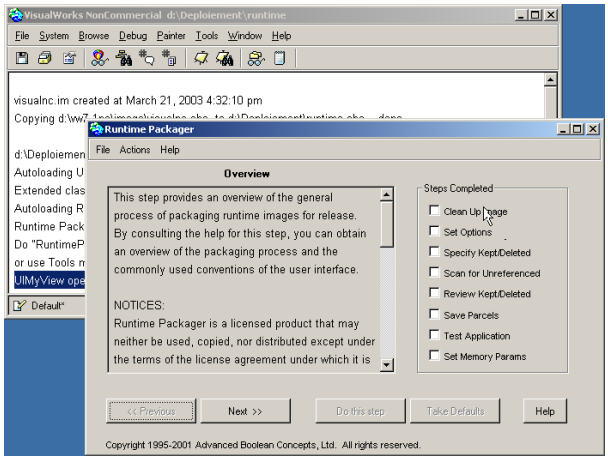


FIGURE 9.6 – (runtimePackager.eps)

– La seconde étape permet de spécifier ce que doit faire l'image produite après le processus d'intégration. Ici on va dire que la Startup Class est UIMyView, et que le message est 'open' pour ouvrir l'application.

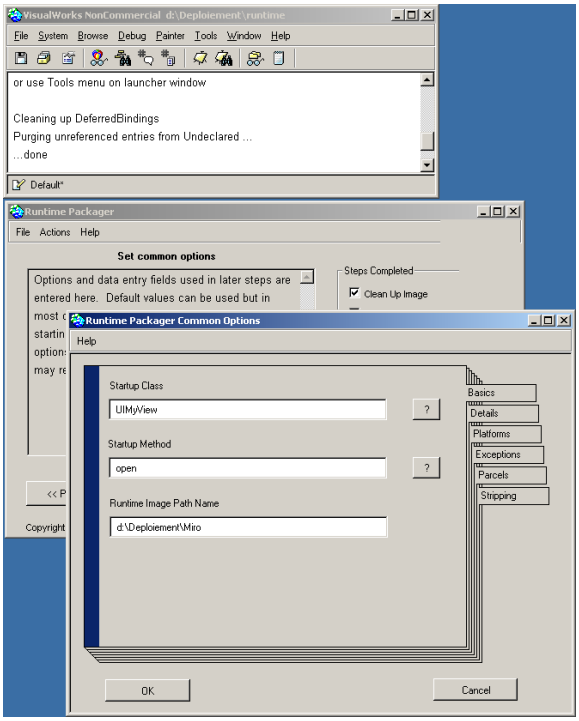


FIGURE 9.7 – (startup-6.eps)

9.1.8 Suite de l'étape 'set options'

Voir figure 9.8 On peut aussi s'intéresser à d'autres aspects (onglet details) tels que la lecture d'une ligne de commande, que l'on supprime ici.

9.1.9 Etape de sélection ou rejet des classes

Voir figure 9.9 Troisième étape : ici on précise simplement ce que l'on veut garder, c'est à dire l'application.

– Selectionner Smalltalk,
– Chercher UIMyView dans la fenetre 'classe' au milieu, et la basculer dans la fenetre classe.

A la suite, on procède à l'étape 'scan'. On peut ensuite sauter à l'étape test.

9.1.10 Test de l'image déployée

Voir figure 9.10 Ici on lance un test durant lequel l'application va etre ouverte. Ne pas oublier de la refermer et d'achever le test dans la fenetre qui est donnée.

A la suite on va positionner les valeurs de la mémoire aux options par défaut.

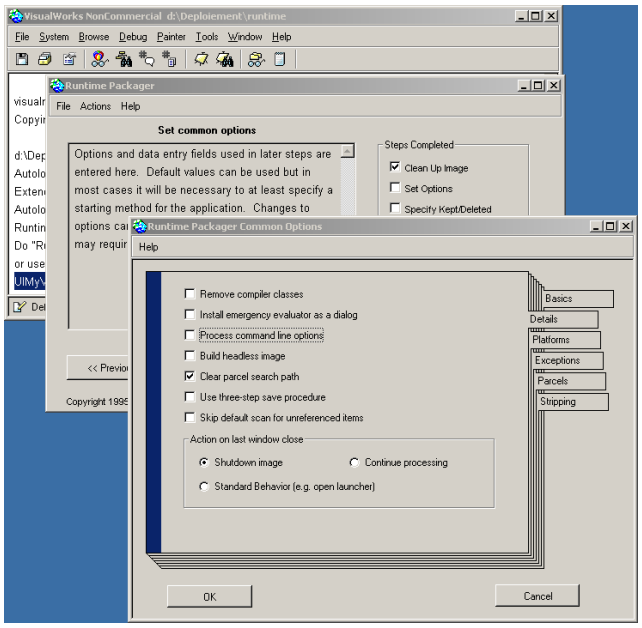


FIGURE 9.8 – (cmdLine-7.eps)

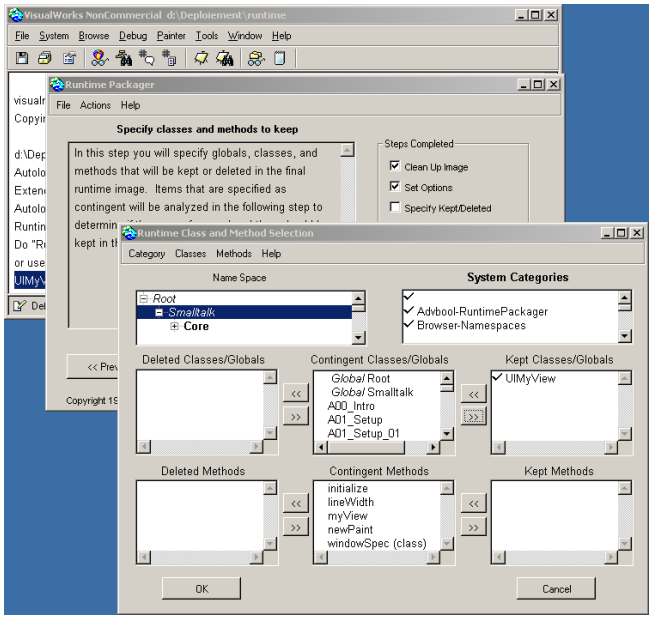


FIGURE 9.9 – (kept-8.eps)

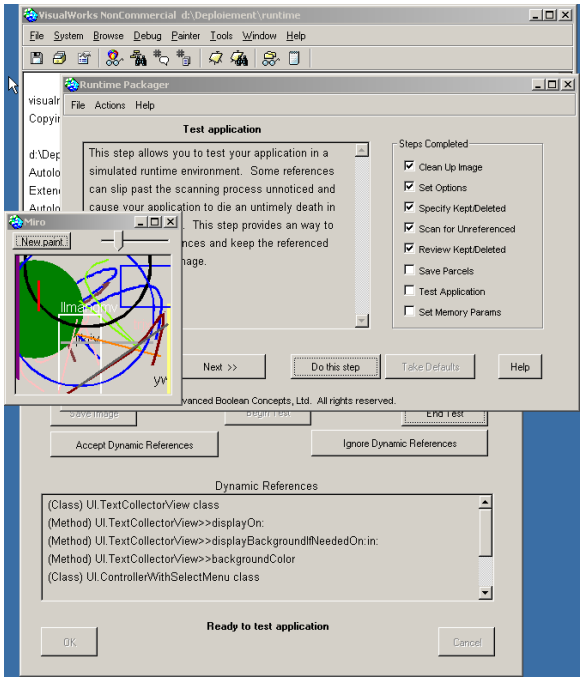


FIGURE 9.10 – (testEmule-9.eps)

9.1.11 Finalisation

Voir figure 9.11 et 9.12 La dernière étape va produire une application Miro dans votre répertoire. Cette étape peut etre assez longue. Elle présente des fenetres de trace telles que 9.11.

9.1.12 Test de l'application

Voir figure 9.13 Cliquer sur l'image miro.im...

9.2 Production d'un exécutable

9.2.1 Installation des utilitaires

Voir figure 9.14 Deziper ces utilaitires dans votre répertoire de deploiement. Vous les trouverez dans le répertoire Visualworks, 'packaging', puis win...

9.2.2 Ouvrir une fenetre de commande

Voir figure 9.15 Dans le menu demarrer chercher executer, puis cmd. Vous avez un terminal de texte qui apparait.

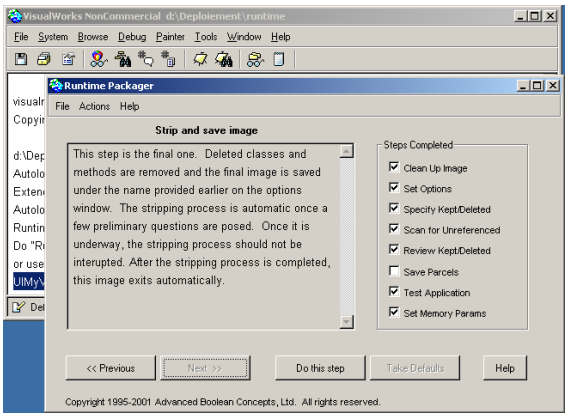


FIGURE 9.11 – (finalStep-10.eps)

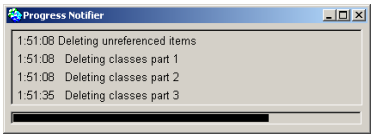


FIGURE 9.12 – (notifier-11.eps)

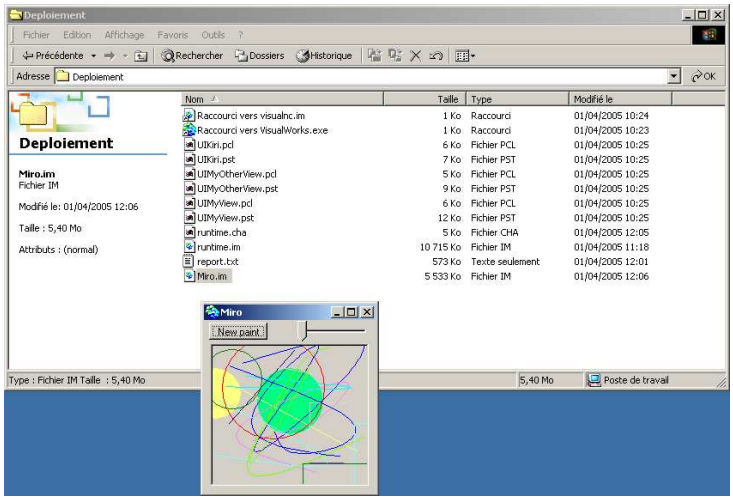


FIGURE 9.13 – (appliFinale-11.eps)

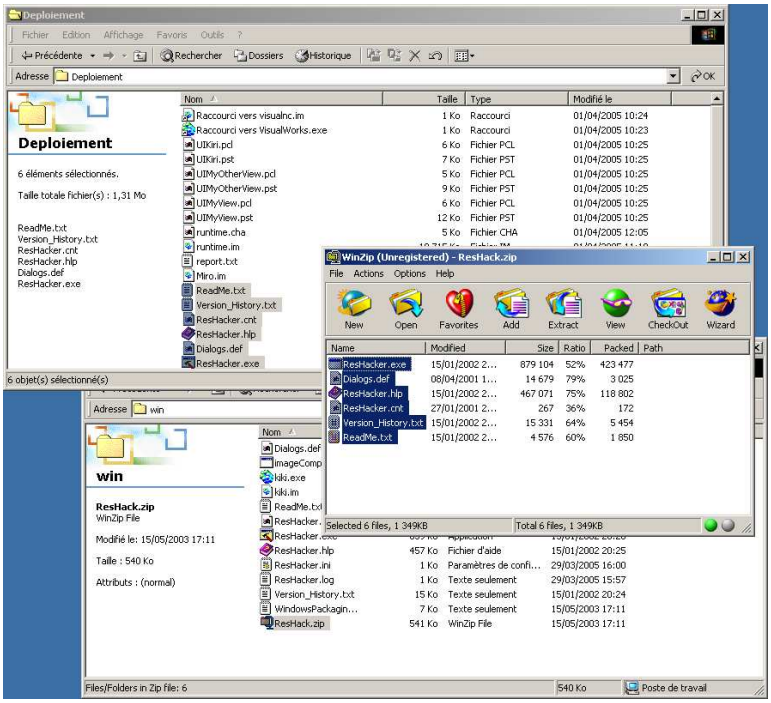


FIGURE 9.14 – (dezipPackager-12.eps)

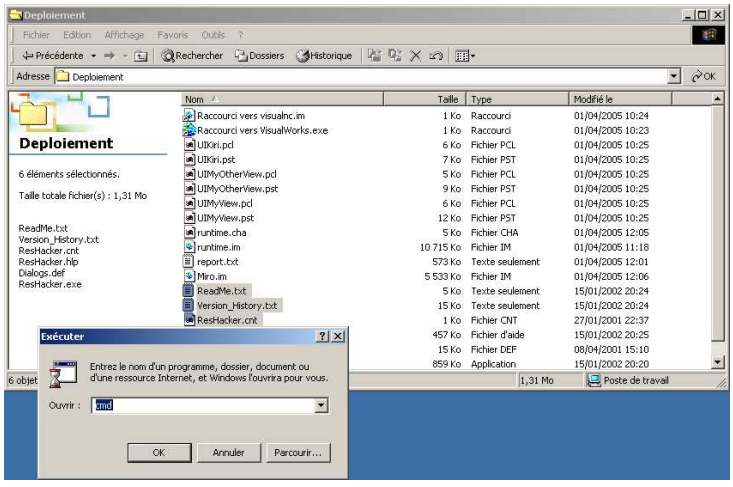


FIGURE 9.15 – (lanceCommande.eps)

9.2.3 Execution de l'éditeur de ressource

Voir figure 9.16 Dans les fichiers dézipés, il y a un fichier qui explique le mode d'emploi de cet éditeur. Il est possible d'associer l'interpréteur et une image, de changer une icone...

Transposer la ligne de la figure 9.16 en changeant le repertoire d'installation de visualworks, ou le nom de votre image, et exécuter.

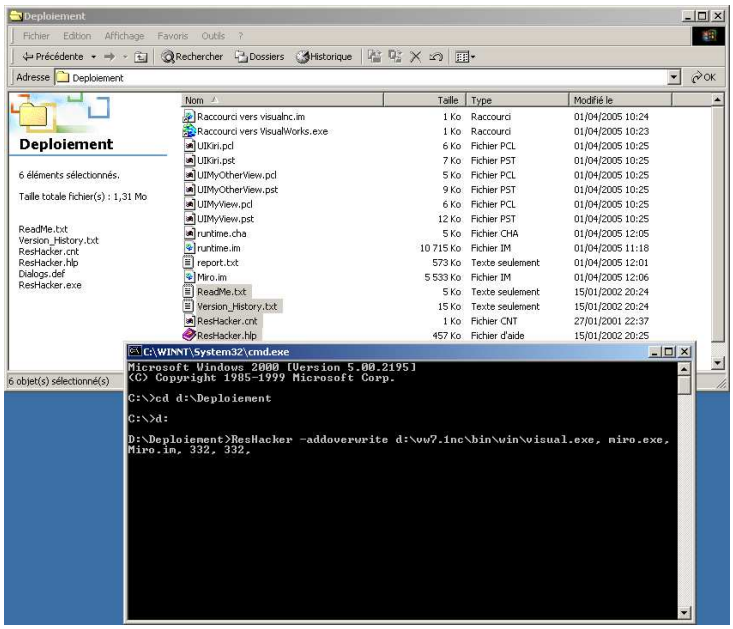


FIGURE 9.16 – (buildAppli-14.eps)

9.2.4 Execution de l'application

Voir figure 9.17 Vous devez voir un fichier .exe dans votre repertoire. L'exécuter pour tester.

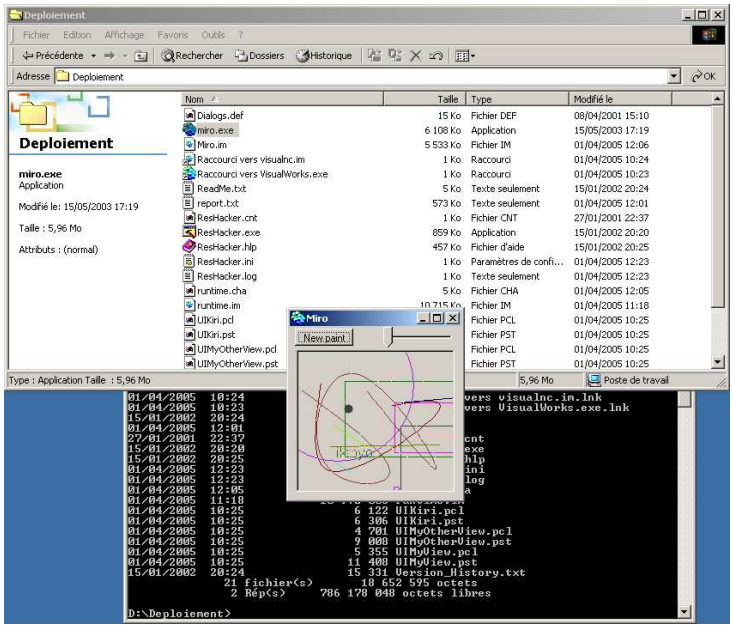


FIGURE 9.17 – (exe-final-15.eps)

Chapitre 10

Interface vers des librairies externes

L'intégration rapide de technologies nouvelles est vitale pour suivre l'évolution des plateformes. Cette intégration consiste à créer de nouvelles classes recouvrant les ressources nouvelles et en donnant l'accès.

Un des outils les plus efficaces est le *DLL&C Connect*. Avec cet outil une intégration procède en plusieurs étapes :

1. écriture d'une librairie C permettant l'accès aux ressources nouvelles (`test.c`),
2. génération d'une librairie dynamique (`test.so`), préparation de fichiers de déclarations (`test.h`),
3. utilisation des outils DLL pour analyser ces fichiers et générer une classe interface (`ExternalInterfaceTest`),
4. ajout de méthodes utilitaires et de test dans (`ExternalInterfaceTest`),
5. création de classes assurant des modèles solides pour la ressource intégrée, en s'appuyant sur les services de la librairie externe et de son interface.

10.1 Pas à pas

10.1.1 Prise en main de l'outil d'intégration DLL

Il s'agit d'un paquetage additionnel que l'on charge à partir du menu système. Ce paquetage dispose d'une documentation fournie, à la hauteur des concepts qu'il amène.

Le chargement produit une icone supplémentaire **C** visible dans la figure 10.1. Le menu *Tools*, est également complété.

Après ces opérations, on ouvre l'outil frontal qui va permettre de lire les fichiers externes. On commence par créer une classe interface qui va s'insérer dans la liste des interfaces externes.

La fenêtre 10.3 présente le brosseur, dans lequel la nouvelle classe est repérable. On commence par localiser les répertoires où se trouve le développement à intégrer. Il s'agit de répertoires locaux ou systèmes, typiquement *xx/include* ou *yy/lib*.

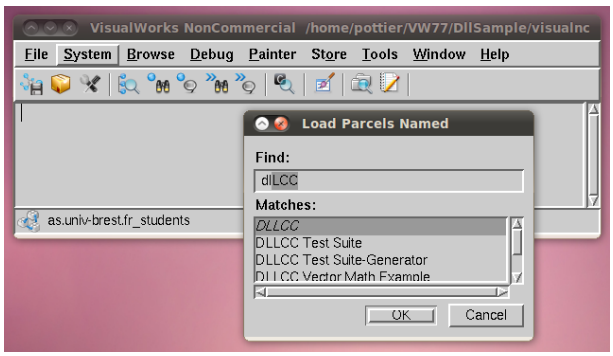


FIGURE 10.1 – Chargement du paquetage DLL and C Connect

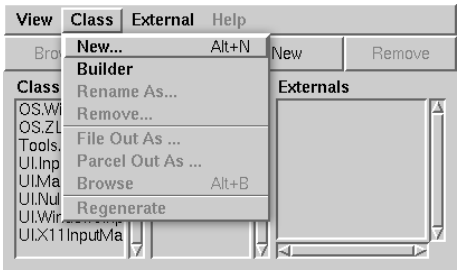


FIGURE 10.2 – Fenêtre de contrôle pour les interfaces externes

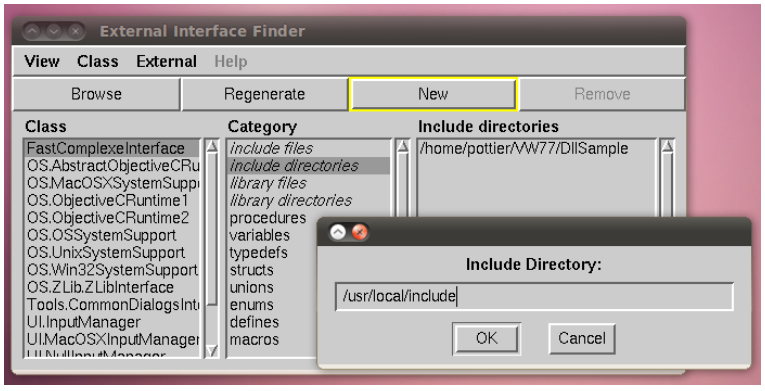


FIGURE 10.3 – Configuration des répertoires où se trouvent les fichiers externes

L'outil va chercher les fichiers externes dans les répertoire fournis. Ultérieurement, l'image Smalltalk utilisera également cette configuration pour chercher les librairies externes.

include files liste des fichiers de déclaration C contenant les entêtes de fonctions, les constantes, etc...

include directories localisation de ces fichiers,

library files liste des librairies dynamiques,

library directories localisation de ces librairies.

10.1.2 Génération des méthodes de l'interface

On ouvre une nouvelle fenêtre intitulée *Builder*, accessible à partir du menu Class (figure 10.4). Ici, on va d'abord analyser les fichiers de déclarations (`fastComplexe.h`), en utilisant la commande (*Parse*). Cette analyse va ramener la totalité des déclarations dans l'interface, ventilée dans les catégories *procédures, variables, structures, ...*

On choisit ensuite les entrées qui doivent apparaitre dans la classe interface, en les biffant, et en exécutant la commande *Add methods*. Cette opération est répétée pour chaque catégorie d'objets.

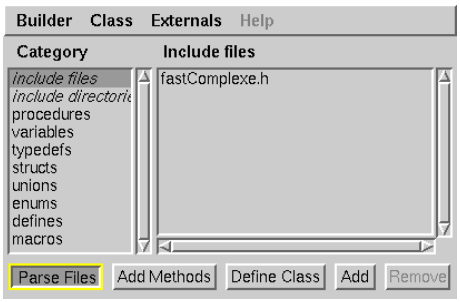


FIGURE 10.4 – Constructeur de classes, de C à Smalltalk

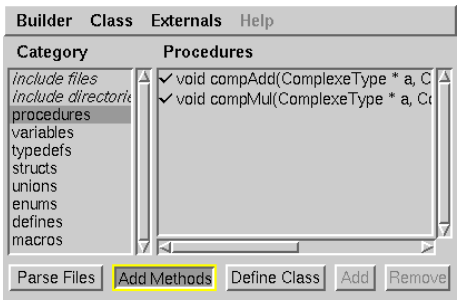


FIGURE 10.5 – Choix des objets C à interfacer

10.1.3 Observation de la classe interface

La génération de code est maintenant observable dans le brosseur. On cherche la classe et on observe l'effet de l'outil *Builder*, à commender par la désignation des fichiers de ressources externes.

```
Smalltalk defineClass: #FastComplexeInterface
    superclass: #{External.ExternalInterface}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            private FastComplexeInterfaceDictionary.*
            '
    category: 'ExternalInterface-New'
    attributes: #(
            #(#includeFiles #('fastComplexe.h'))
            #(#includeDirectories #('/home/vous/dev/DllSample'))
            #(#libraryFiles #('fastComplexe.so'))
            #(#libraryDirectories #('/home/vous/dev/DllSample'))
            #(#beVirtual false)
            #(#optimizationLevel #full))
```

Il est bien entendu possible de programmer normalement cette classe, et d'y ajouter des variables d'instances. On repère les catégories de méthodes que le *Builder* présentait, avec les sources qui y ont été générés.

procédure :
```
compAdd: a with: b with: c
        <C: void compAdd(ComplexeType * a, ComplexeType * b, ComplexeType * c)>
        ^self externalAccessFailedWith: _errorCode
```

type :
```
ComplexeType
        <C: typedef struct {
                        double re, im;
                } ComplexeType>
```

variable :
```
nullComplexe
        <C: ComplexeType nullComplexe>
```

Une des premières chose que l'on souhaite vérifier est le couplage à la librairie externe. La méthode de test qui suit montre comment procéder pour accéder aux ressources :

```
testAdd

        | fc a b c |
        fc := FastComplexeInterface new.
        a := fc ComplexeType gcMalloc.
        b := fc idComplexe.
        c := fc ComplexeType gcMalloc.
        a memberAt: #re put: 1.1d.
        a memberAt: #im put: 1.1d.
        fc compAdd: a with: b with: c.
        c inspect
```

10.2 Ressources C utilisées

Dans cet exemple, on utilise les éléments suivants :
- fastComplexe.c : source de la librairie
- fastComplexe.h : déclarations relatives à la librairie
- testComplexe.c : programme de test validant le fonctionnement de fastComplexe.c
- Makefile

10.2.1 Compilation

La documentation précise les options à prendre dans le Makefile selon les plateformes. La compilation d'une librairie dynamique (DLL, .so, .dylib) exige par exemple une entrée du type qui suit :

```
fastComplexe: testComplexe.o fastComplexe.so
        cc $(CFLAGS) -L /home/vous/dev/DllSample \
        fastComplexe.so testComplexe.o -o fastComplexe

fastComplexe.so: fastComplexe.c fastComplexe.h
        cc $(CFLAGS)  -c -fpic fastComplexe.c
        ld  -o fastComplexe.so -shared fastComplexe.o
```

10.2.2 Déclaration

Le fichier de déclaration est ici `fastComplexe.h`. On y met les directives présentant l'interface usager de la librairie.

Ici, on présente les objets présentés section 10.1.3 dans leur version originale,.

```
typedef struct { double re,im;} ComplexeType;
extern void compAdd(ComplexeType *a,
     ComplexeType *b,
     ComplexeType *c);
extern void compMul(ComplexeType *a,
     ComplexeType *b,
     ComplexeType *c);
extern ComplexeType nullComplexe, idComplexe;
```

10.2.3 Fichiers C

Ici on y met l'implémentation de la librairie, et ses tests. Le fichier de déclaration est lu dans ces sources. Ici, on opère des additions et multiplications sur des nombres complexes passés par adresse.

Noter que les variables définies dans la librairie peuvent être accédées dans l'image Smalltalk-80.

10.3 Les ressources externes vues dans l'image

Il est intéressant de se familiariser avec la représentation des ressources externes. Un exemple simple est celui de la création d'un complexe, que l'on initialise, et que l'on inspecte.

```
testAlloc

        | fc z |
        fc := FastComplexeInterface new.
        z := fc ComplexeType gcMalloc.
        z memberAt: #re put: 1.0d.
        z memberAt: #im put: 1.0d.
        z inspect
```

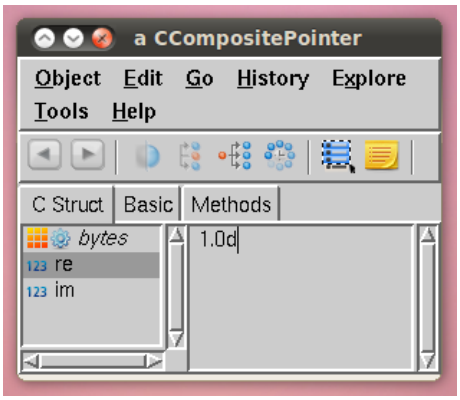


FIGURE 10.6 – Inspecteur sur une variable dynamique externe

Quelques commentaires sur le code de testAlloc. Cette méthode crée une variable dynamique externe à Smalltalk, mais gérée par le ramasse-miette de l'image.

- `fc := FastComplexeInterface new.`.
  fc n'est pas vraiment un objet, mais un moyen de manipuler les ressources externes en désignant l'interface.
- `z := fc ComplexeType gcMalloc.`
  ComplexeType est une structure, et le message gcMalloc alloue une variable conforme à ce type. Le résultat est un pointeur.
- `z memberAt: #re put: 1.0d. z memberAt: #im put: 1.0d.`
  Les structures C s'adressent en désignant simplement les champs par leur nom.
- `z inspect`
  Voir l'allure de z dans son inspecteur, figure 10.6.

De manière similaire on peut également allouer des tables de variables. Par exemple le code qui suit montre comment allouer et adresser ce genre de variable.

10.4 Classe utilisant l'interface

Les classes utilisant l'interface ne doivent pas faire ce concession à l'abstraction. Elles s'appuient sur des ressources, mais représentent des objets classiques, même si ils sont externes. L'ébauche qui suit représente un vecteur de complexes.

```
Smalltalk defineClass: #MesTableauxComplexes
	superclass: #{Core.Number}
	indexedType: #none
	private: false
	instanceVariableNames: 'complexeTable capacity '
	classInstanceVariableNames: 'Interface '
	imports: ''
	category: ''!

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!


!MesTableauxComplexes class methodsFor: 'instance creation'!

new: n

	| mtc table |
	mtc := self basicNew.
	table := MesComplexes interface ComplexeType gcMalloc: n.
	mtc complexeTable: table.
	mtc capacity: n.
	^mtc! !

!MesTableauxComplexes class methodsFor: 'testing'!

testAdd

	| un deux |
	un := self testNewWrite.
	deux := self testNewWrite.
	^un + deux!

testNew

	| table n un |
	n := 100.
	table := self new: n.
	1 to: n
		do:
			[:i |
			un := MesComplexes re: i asDouble im: 0 asDouble.
			table at: i put: un]!

testNewRead

	| table n un ws |
	n := 100.
	table := self testNewWrite.
	ws := Array new writeStream.
	1 to: n
		do:
			[:i |
			un := table at: i.
			ws nextPut: un].
```

```
	^ws contents!

testNewWrite

	| table n un |
	n := 100.
	table := self new: n.
	1 to: n
		do:
			[:i |
			un := MesComplexes re: i asDouble im: 0 asDouble.
			table at: i put: un].
	^table! !

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!


MesTableauxComplexes comment:
'MesTableauxComplexes represente des vecteurs de complexes.
Bernard Pottier. Mars 2010.

Instance Variables
	complexeTable	<Array of: MesComplexes>	table externe
	capacity	<Integer>	taille de la table externe
'!

!MesTableauxComplexes methodsFor: 'accessing'!

at: index

	| tab |
	((1 to: capacity) includes: index) ifFalse: [^self error: 'range error'].
	tab := self complexeTable.
	tab := tab + (index - 1).	"C alignment is on zero"
	^MesComplexes re: (tab memberAt: #re) im: (tab memberAt: #im)!

at: index put: object

	| tab |
	((1 to: capacity) includes: index) ifFalse: [^self error: 'range error'].
	tab := self complexeTable.
	tab := tab +( index - 1). "C alignment is on zero"
	tab memberAt: #re put: object re.
	tab memberAt: #im put: object im!

capacity
	^capacity!

capacity: anObject
	capacity := anObject!

complexeTable
	^complexeTable!
```

```
complexeTable: anObject
        complexeTable := anObject! !

!MesTableauxComplexes methodsFor: 'arithmetic'!

+ aVector

   " a accelerer en implantant la boucle ds la lib externe"
        | result x y |
        self capacity ~= aVector capacity ifTrue: [^self error: 'different vector sizes'].
        result := self class new: self capacity.
        1 to: self capacity
                do:
                        [:i |
                        x := self at: i.
                        y := aVector at: i.
                        result at: i put: x + y].
        ^result! !
```

Table des matières