7

Le noyau système de Squeak

Ce chapitre est consacré à la présentation des éléments techniques de « bas niveau » de Squeak, ceux qui font fonctionner le système. Un système Squeak complet assure les fonctionnalités suivantes :

- gestion des interactions avec le système de gestion de fichiers, avec l'écran, et avec les périphériques d'entrée (clavier-souris) ;
- analyse, compilation et exécution du code, gestion des erreurs lors de l'exécution ;
- gestion des exécutions en temps partagé (threads, processus).

Les classes qui implémentent ces différentes fonctionnalités forment ce que nous appellerons le « noyau système » de Squeak. Les classes de ce noyau sont regroupées dans diverses catégories de classes, parmi lesquelles on distingue en particulier :

- Les entrées-sorties :
  - `System-Files` : les classes de gestion des entrées-sorties sur fichiers ;
  - `Morphic-Events` : les classes de gestion des événements souris et clavier ;
  - `Graphics-Primitives` et `Graphics-Display Objects` : les classes de gestion des affichages sur écran.
- Les objets, classes et méthodes :
  - `Kernel-Objects` : les classes du sommet de la hiérarchie, qui définissent les comportements partagés par tous les objets de Squeak ;
  - `Kernel-Classes` : les classes de gestion des classes ;
  - `Kernel-Methods` : les classes de gestion des méthodes ;
  - `Kernel-Magnitude` et `Kernel-Numbers` : les classes de gestion des dates, des nombres, et la classe `Magnitude`, classe générique des objets dotés d'une relation d'ordre.

- La machine virtuelle, le compilateur :
  - `VMConstruction-Interpreter` ;
  - `VMConstruction-Translation to C` : les classes de définition et de transcodage en C de la machine virtuelle ;
  - `System-Compiler` : le compilateur, qui génère à partir du code Squeak du *bytecode* à destination de la machine virtuelle.
- Le temps partagé :
  - `Kernel-Processes` : les classes de gestion des processus, sémaphores et délais.

Nous ne présenterons dans ce chapitre que les composants essentiels de cet ensemble de classes, qui permettent à Squeak de fonctionner, en commençant par la gestion des entrées-sorties. La gestion des processus sera traitée au chapitre 9.

La gestion des entrées-sorties

Cette section concerne toutes les opérations qui se situent en dehors du domaine interne de Squeak et de sa machine virtuelle pour toucher les périphériques de stockage de masse (fichiers), d'entrée (clavier, souris) et d'affichage (écran).

Entrées-sorties sur fichier

Pour interagir avec un fichier physique stocké sur un périphérique magnétique quelconque, Squeak utilise un objet interne qui représente ce fichier. Techniquement parlant, ces objets sont des sortes de `Stream` (voir chapitre 6, consacré aux classes de collections), dont la collection support correspond au contenu du fichier. Toutes les opérations disponibles sur les `Stream` le sont donc pour les fichiers.

Les classes spécifiques aux fichiers, sous-classes de la classe `FileStream`, sont regroupées dans la catégorie de classe `System-Files`. Elles héritent des méthodes d'accès et d'ajout des `Stream` : `next`, `nextPut:`, `atEnd...` auxquelles s'ajoutent des méthodes spécifiques aux fichiers : `contentsOfEntireFile`, `close`, `flush`, `localName`…

Voici la position des classes de gestion de fichiers dans la hiérarchie des `Stream` :

```
Stream #()
   PositionableStream #('collection' 'position' 'readLimit')
      WriteStream #('writeLimit')
         ReadWriteStream #()
            FileStream #('rwmode')
               StandardFileStream #('name' 'fileID' 'buffer1')
               CrLfFileStream #('lineEndConvention')
                     BDFFontReader #('properties')
                     HtmlFileStream #('prevPreamble')
```

Les objets qui représentent les répertoires appartiennent, quant à eux, aux sous-classes de `FileDirectory`, qui se spécialisent en fonction du système d'exploitation.

La hiérarchie des classes de gestion des répertoires est la suivante :

```
Object #()
   FileDirectory #('pathName')
      AcornFileDirectory #()
      DosFileDirectory #()
      MacFileDirectory #()
      UnixFileDirectory #()
```

La classe qui doit être utilisée est dynamiquement déterminée par le système lui-même.

L'exécution de `FileDirectory on: 'C:\Xavier Briffault\Eyrolles\Squeak\SQUEAK 3\Images de travail\1'` renvoie, sur une plate-forme Windows par exemple, une instance de `DosFileDirectory`. `FileDirectory default` retourne le répertoire par défaut. Celui-ci est en fait stocké dans la variable de classe `DefaultDirectory`, et peut être modifié à cet endroit.

Les spécificités de la syntaxe des noms de fichiers sur les différentes plates-formes (par exemple, l'usage de `/`, `\`, `:` pour séparer les éléments du chemin d'accès) sont gérées par les méthodes du protocole `platform specific`. Les méthodes `dot`, `extensionDelimiter`, `isCaseSensitive`, `maxFileNameLength`, `pathNameDelimiter`, `slash`, sont redéfinies dans les sous-classes de façon à renvoyer les valeurs adéquates.

Voici les principales méthodes qui sont définies sur les classes de gestion des répertoires :

- les méthodes d'accès aux répertoires contenants et contenus (`containingDirectory`, `directoryNamed:`, `directoryNames`), aux noms de fichiers (`fileAndDirectoryNames`, `fileNames`, `fullNamesOfAllFilesInSubtree`), aux fichiers eux-mêmes (`fileNamed: uneString`, `readOnlyFileNamed: localFileName`, qui renvoient une instance de `FileStream` sur le fichier correspondant à l'argument) ;
- les opérations sur les fichiers : `copyFile:toFile:`, `copyFileNamed:toFileNamed:`, `createDirectory:`, `deleteFileNamed:`, etc. ;
- différentes méthodes utilitaires : `fileNamesMatching:`, `isLegalFileName:`, `url`, `filesContaining:caseSensitive:`, etc.

L'exemple suivant illustre le fonctionnement de quelques-unes de ces méthodes :

```
|fic|
fic := (FileDirectory on: 'C:\Xavier Briffault\Eyrolles\Squeak\1')
   fileNamed: 'test.txt'.
fic nextPutAll: 'chaîne de test'.
fic close.
fic := (FileDirectory on: 'C:\Xavier Briffault\Eyrolles\Squeak\1')
   readOnlyFileNamed: 'test.txt'.
fic contentsOfEntireFile. "Renvoie 'chaîne de test'"
```

Plusieurs classes utilisées pour la compression/ décompression de données sont disponibles (catégorie `System-Compression`), et permettent de gérer les formats Gzip, Zip, PKZip :

```
InflateStream #('state' 'bitBuf' 'bitPos' 'source' 'sourcePos'
      'sourceLimit' 'litTable' 'distTable' 'sourceStream')
   FastInflateStream #()
   GZipReadStream #()
   ZLibReadStream #()
```

Entrées-sorties sur le port série

La gestion des entrées-sorties sur le port série est assurée par la classe `SerialPort`, qui propose les méthodes d'accès habituelles (`baudRate`, `parityType`…). Une interface d'accès aux ports MIDI est fournie avec la classe `SimpleMIDIPort` (catégorie `Sounds-Scores`).

Gestion de la souris et du clavier

Les événements engendrés par les actions de l'utilisateur sur le clavier et la souris sont définis par les sous-classes de `UserInputEvent`, `KeyboardEvent`, `MouseButtonEvent` et `MouseMoveEvent`. Voici la hiérarchie des classes de gestion d'événements clavier et souris :

```
Object #()
   MorphicEvent #('timeStamp' 'source')
      UserInputEvent #('type' 'buttons' 'position' 'handler' 'wasHandled')
         KeyboardEvent #('keyValue')
         MouseEvent #()
            MouseButtonEvent #('whichButton')
            MouseMoveEvent #('startPoint' 'trail')
```

Lorsqu'une action de l'utilisateur a lieu sur le clavier ou la souris, l'événement de bas niveau engendré par le système d'exploitation est pris en charge par l'objet chargé de la gestion des événements dans l'espace Morphic courant. Cet objet est une instance de la classe `HandMorph`. Une instance de la classe `WorldState` est chargée d'assurer la boucle de prise en charge des événements (méthode `WorldState>>doOneCycleNowFor:`), et d'appeler la méthode `HandMorph>>processEvents` pour traiter l'événement :

```
WorldState>>doOneCycleNowFor: aWorld
   self flag: #bob.
   LastCycleTime := Time millisecondClockValue.
   self
      handsDo: [:h |
         activeHand := h.
         h processEvents.
         activeHand := nil].
   aWorld runStepMethods.
   self displayWorldSafely: aWorld
```

```
HandMorph>>processEvents
   | evt evtBuf type hadAny |
   hadAny := false.
   [(evtBuf := Sensor nextEvent) == nil]
      whileFalse: [evt := nil.
         type := evtBuf at: 1.
         type = EventTypeMouse
            ifTrue: [evt := self generateMouseEvent: evtBuf].
         type = EventTypeKeyboard
            ifTrue: [evt := self generateKeyboardEvent: evtBuf].
         type = EventTypeDragDropFiles
            ifTrue: [evt := self generateDropFilesEvent: evtBuf].
         evt == nil
            ifFalse: [
               self handleEvent: evt.
               hadAny := true.
               evt isMouse
                  ifTrue: [^ self]]].
   mouseClickState notNil
      ifTrue: [
         mouseClickState handleEvent: lastMouseEvent asMouseMove
            from: self].
   hadAny
      ifFalse: [
         self mouseOverHandler processMouseOver: lastMouseEvent]
```

L'événement Squeak généré (par les méthodes `HandMorph>>generateMouseEvent:` et `HandMorph>>generateKeyboardEvent`) est alors traité par la méthode `HandMorph>>handleEvent:`, méthode assez longue, qui a principalement pour objet de transmettre l'événement au morph qui a le focus courant. À cet effet, c'est la méthode `HandMorph>>sendEvent:focus:` qui est appelée :

```
HandMorph>>sendEvent: anEvent focus: focusHolder
   focusHolder
      ifNil: [^ owner processEvent: anEvent].
   ^ self sendFocusEvent: anEvent to: focusHolder
```

Dans cette méthode, `owner` est le morph qui est concerné par l'événement.

L'événement sera finalement traité par un dispatcheur d'événements, une instance de la classe `EventHandler`, appelé par la méthode `Morph>>processEvent:using: :`

```
Morph>>processEvent: anEvent using: defaultDispatcher
   (self rejectsEvent: anEvent)
      ifTrue: [^ #rejected].
   ^ defaultDispatcher dispatchEvent: anEvent with: self
```

Gestion de l'écran

Afin d'assurer une portabilité complète sur tout type d'ordinateur et de système d'exploitation, la gestion de l'affichage graphique est entièrement prise en charge par Squeak. Seules les opérations de très bas niveau, relatives à la manipulation des pixels, sont en fait déléguées à la machine virtuelle.

Les principales classes de gestion des affichages sont regroupées dans les catégories `Graphics-Display Objects` et `Graphics-Primitives`. On y trouve en particulier la classe `DisplayScreen`, dont l'unique instance, référencée par la variable globale `Display`, représente la zone d'affichage courante complète.

Voici la hiérarchie de la classe `DisplayScreen` :

```
Object #()
   DisplayObject #()
      DisplayMedium #()
         Form #('bits' 'width' 'height' 'depth' 'offset')
            DisplayScreen #('clippingBox')
               ExternalScreen #('argbMap' 'allocatedForms')
                     B3DDisplayScreen #()
```

La plupart des traitements sont en fait délégués à des instances de la classe `BitBlt` (pour Bit Block Transfert), qui assure les affichages primitifs des pixels en liaison avec la machine virtuelle. À titre d'exemple, voici deux méthodes significatives de `DisplayScreen` et de `BitBlt` qui se chargent de l'affichage de blocs de pixels :

```
DisplayScreen>>copyBits: rect from: sf at: destOrigin clippingBox: clipRect
rule: cr fillColor: hf
   (BitBlt current
      destForm: self
      sourceForm: sf
      fillColor: hf
      combinationRule: cr
      destOrigin: destOrigin
      sourceOrigin: rect origin
      extent: rect extent
      clipRect: (clipRect intersect: clippingBox)) copyBits

BitBlt>>copyBits
   <primitive: 'primitiveCopyBits' module:'BitBltPlugin'>
      "primitiveExternalCall"
   "Check for compressed source, destination or halftone forms"
   (combinationRule >= 30
         and: [combinationRule <= 31])
      ifTrue: ["No alpha specified -- re-run with alpha = 1.0"
         ^ self copyBitsTranslucent: 255].
   ((sourceForm isKindOf: Form)
         and: [sourceForm unhibernate])
```

```
      ifTrue: [^ self copyBits].
   ((destForm isKindOf: Form)
         and: [destForm unhibernate])
      ifTrue: [^ self copyBits].
   ((halftoneForm isKindOf: Form)
         and: [halftoneForm unhibernate])
      ifTrue: [^ self copyBits].
   "Check for unimplmented rules"
   combinationRule = Form oldPaint
      ifTrue: [^ self paintBits].
   combinationRule = Form oldErase1bitShape
      ifTrue: [^ self eraseBits].
   self error: 'Bad BitBlt arg (Fraction?); proceed to convert.'.
   "Convert all numeric parameters to integers and try again."
   destX := destX asInteger.
   destY := destY asInteger.
   width := width asInteger.
   height := height asInteger.
   sourceX := sourceX asInteger.
   sourceY := sourceY asInteger.
   clipX := clipX asInteger.
   clipY := clipY asInteger.
   clipWidth := clipWidth asInteger.
   clipHeight := clipHeight asInteger.
   ^ self copyBitsAgain
```

La gestion des objets graphiques de base est par ailleurs assurée par les classes `Color`, `ColorMap`, `Pen`, `Point`, `Quadrangle`, `Rectangle`, `WarpBlt`.

Traitements définis au sommet de la hiérarchie des classes

Après les mécanismes d'entrées-sorties, nous allons maintenant nous intéresser aux classes fondamentales du sommet de la hiérarchie de Squeak. Les principes généraux de l'organisation de ces classes ont été présentés dans le chapitre 4, consacré au modèle objet de Squeak. Nous allons aborder ici des éléments relatifs aux aspects techniques.

La catégorie `Kernel-Objects` comprend douze classes (`Boolean`, `EventModel`, `False`, `MessageSend`, `Model`, `MorphObjectOut`, `Object`, `ObjectOut`, `ObjectTracer`, `ObjectViewer`, `ProtoObject`, `True`, `UndefinedObject`) qui définissent les objets les plus « primitifs » de Squeak.

Les classes ProtoObject et Object

La racine de la hiérarchie des classes est constituée par la classe `ProtoObject`. `ProtoObject` est une classe qui définit les comportements partagés par tous les objets, sans exception, de

Squeak, soit l'ensemble minimal de ce que doit savoir faire un objet Squeak. On y trouve en particulier == (égalité de pointeurs) et la méthode réciproque ~~ (différence de pointeurs).

La méthode `doesNotUnderstand:` y est également définie. Elle est déclenchée par la machine virtuelle en envoyant le message correspondant à un objet lorsque celui-ci reçoit un message qu'il ne sait pas traiter. Cette méthode a un comportement par défaut qui consiste à ouvrir un débogueur sur le message qui ne peut être traité. Ce comportement peut parfaitement être redéfini dans les sous-classes, par exemple pour personnaliser le message d'erreur, enregistrer les messages erronés dans un journal d'erreurs, comme nous le verrons au chapitre 8. Nous verrons un peu plus loin, dans la section « La gestion des erreurs », que cette méthode est redéfinie dans `Object`.

On trouve enfin la méthode `become:`, qui permute les pointeurs du receveur et de l'argument. Ainsi, après l'exécution de `objet1 become: objet2`, toutes les variables qui pointaient sur `objet1` pointent désormais sur `objet2`, et réciproquement. Cette technique est par exemple utilisée pour faire grossir les collections : lorsqu'une collection devient trop petite pour le nombre d'éléments qu'elle contient, une nouvelle collection plus importante est créée, les éléments de la première y sont transférés, et tous les pointeurs vers la première collection sont redirigés vers la deuxième (voir la méthode `grow` de `MethodDictionary`, par exemple).

Un autre excellent exemple d'utilisation de cette technique de permutation de pointeurs réside dans la création de *wrappers* (encapsulateurs ajoutant des comportements à des objets) et de *proxies* (représentants locaux d'objets distants), que nous détaillerons dans le chapitre 8 consacré à la réflexivité.

Comme nous pouvons le constater, `ProtoObject` n'implémente que fort peu de méthodes, et de nouvelles sous-classes directes ne lui sont ajoutées que dans des cas très précis, tels que ceux que nous présenterons dans le chapitre sur la réflexivité.

La plupart des comportements partagés sont en fait définis sur la classe `Object`, sous-classe directe de `ProtoObject`. Les méthodes de comparaison d'objets = et ~= sont définies à ce niveau. Elles sont ici équivalentes à == et ~~, et doivent donc être redéfinies dans les sous-classes. Les méthodes de copie sont également définies sur la classe `Object` et sont aux nombres de trois :

- `copy` (et `shallowCopy`, strictement identique) effectue une copie de premier niveau du receveur : les variables de la copie du receveur pointent sur les mêmes objets que l'original ;
- `deepCopy` : la copie se fait à deux niveaux ; les variables d'instance de la copie pointent sur des copies des objets sur lesquels pointaient les variables du receveur ;
- `veryDeepCopy` : la copie se fait à tous les niveaux ; l'ensemble du graphe représenté par les relations entre les objets sur lesquels pointent les variables du receveur est dupliqué.

Gestion des classes et des méthodes

Nous avons jusqu'à présent abordé l'outillage logiciel utilisé par Squeak pour gérer les objets de toutes sortes. Nous abordons à présent le cas spécifique des objets qui sont des classes. Les classes de gestion… des classes sont regroupées dans la catégorie `Kernel-Classes`. Cette catégorie comprend les classes `Behavior`, `Class`, `ClassBuilder`, `ClassCategoryReader`, `ClassCommentReader`, `ClassDescription`, `ClassOrganizer` et `MetaClass`.

> **Remarque**
>
> Il n'est pas nécessaire de comprendre les détails internes de la modélisation des classes en Squeak pour programmer. Nous levons ici le voile pour les lecteurs intrépides.

Nous allons nous intéresser tout d'abord à la hiérarchie de `Behavior`, super-classe de plus haut niveau des classes de gestion de classes. Cette hiérarchie est la suivante :

```
ProtoObject #()
   Object #()
      Behavior #('superclass' 'methodDict' 'format')
         ClassDescription #('instanceVariables' 'organization')
            Class #('subclasses' 'name' 'classPool' 'sharedPools'
                     'environment' 'category')
               [ ... all the Metaclasses ... ]
            Metaclass #('thisClass')
         Oop #()
         Unsigned #()
```

Rappelons que les fondements conceptuels de cette organisation ont été présentés dans le chapitre 4, consacré au modèle objet de Squeak. Nous en abordons ici les aspects techniques.

Au niveau des métaclasses

La classe `Behavior`

La classe `Behavior` implémente le comportement minimal qui permet de compiler des méthodes, et de créer et de faire fonctionner des instances.

Structure

Elle définit les variables suivantes :

- `superclass`, qui fait partie de la structure de toute classe (rappelons que les instances des sous-classes de `Behavior` sont des classes) ;
- `methodDict`, qui contient le dictionnaire des méthodes (associations entre les sélecteurs des méthodes et le code compilé de la méthode), et la variable ;
- `format`, un entier qui représente le type et le nombre de variables d'instances.

Gestion des méthodes

Les méthodes de gestion des méthodes se trouvent dans les protocoles `creating methods dictionary`, `accessing method dictionary` et `testing method dictionary`. On réalise l'ajout d'une nouvelle méthode (déclenché par exemple par un ajout à partir d'un browser, ou à partir d'un ajout programmatique) au moyen de la méthode `addSelector: withMethod:`.

On obtient le code compilé d'une méthode (une `CompiledMethod`) par l'envoi du message `compile:` à une classe, avec le code source de la méthode à compiler. Toute classe doit pouvoir fournir un compilateur qui puisse être utilisé pour produire du code compilé (du *bytecode* Squeak, voir page 152 la section consacrée à la machine virtuelle pour plus de détails). Ce compilateur, qui est renvoyé par la méthode `compilerClass`, est par défaut supporté par la classe `Compiler` (catégorie `System-compiler`), qui accepte la syntaxe Squeak en entrée.

Il est important de noter qu'un autre compilateur peut parfaitement être utilisé. L'environnement Squeak n'est pas dépendant de la syntaxe du langage Squeak. Rien n'empêche de définir des classes dont les méthodes sont écrites selon la *syntaxe* de Java, de C++, ou de tout autre langage dont la *sémantique* reste compatible avec celle de Squeak. Cette facilité est intéressante lorsque des micro-langages bien adaptés à des domaines fonctionnels spécifiques peuvent être définis, et qu'il se révèle plus rapide et/ou plus simple de programmer en utilisant la syntaxe de ces micro-langages, tout en continuant à bénéficier de la librairie de classes et de l'environnement de développement.

Un décompilateur est également disponible (classe `Decompiler`), qui permet de rétro-construire le code source à partir du code compilé.

Autres langages en Squeak

Ces facilités permettent effectivement de mettre à la disposition du développeur Squeak les possibilités offertes par d'autres langages. Il existe ainsi des Prolog, des LISP…, écrits en Squeak. Voir par exemple www.cc.gatech.edu/projects/squeakers/25.html et www.sra.co.jp/people/nishis/smalltalk/Squeak/goodies/.

Nombre de méthodes d'accès aux méthodes d'une classe sont définies.

- `allSelectors` renvoie les sélecteurs de toutes les méthodes définies par la classe et ses super-classes (`selectors` se limite à la classe elle-même). Il est possible d'accéder par le sélecteur d'une méthode à son code compilé (`compiledMethodAt:`), ou à son code source (`sourceCodeAt:`). Rappelons que le sélecteur d'une méthode est un symbole (par exemple, `#sourceCodeAt:`), et pas une chaîne de caractères (`'sourceCodeAt:'`).
- `canUnderstand:` permet de savoir si la classe peut traiter le sélecteur passé en argument (et prend donc en considération les méthodes héritées), tandis que `includesSelector:` se limite à tester la présence du sélecteur dans le dictionnaire des méthodes de la classe (sans prendre en considération les méthodes héritées). Certaines des fonctionnalités relatives à la recherche des envoyeurs, des implémenteurs, des méthodes référençant une classe…, accessibles à partir des browsers, sont également implémentées à ce niveau (méthodes `classThatUnderstands:`, `whichSelectorAccess:...`).

Gestion des instances et de la hiérarchie de classes

La gestion des instances et de la hiérarchie des classes est également assurée à ce niveau, avec la définition des méthodes de création d'instance, `basicNew` et `new`, des méthodes d'accès aux instances, `allInstances` (toutes les instances de la classe), `allSubInstances` (toutes les instances de la classe et de ses sous-classes, des méthodes de création des liens hiérarchiques entre classes (`superclass:`), d'accès à la hiérarchie (`superclass`, `allSuperclasses`, `subclasses`…).

La classe ClassDescription

La classe abstraite `ClassDescription`, sous-classe de `Behavior`, ajoute à ces comportements de nombreuses fonctionnalités. Parmi ces fonctionnalités on distingue les notions de variables d'instances nommées (protocole `instance variables`), de protocoles pour l'organisation des méthodes (protocoles `method dictionary` et `organization`), de gestion du nom de la classe, de gestion du « change set » (suivi des ajouts/ modifications de code effectués pour un projet donné) et du fichier « change » (le fichier qui contient le code source associé à une image particulière), ainsi que le mécanisme nécessaire pour le « file out » (exportation d'une méthode/classe/application sous forme de code source Squeak, protocole `fileIn/Out`).

La classe Class

La classe concrète à partir de laquelle sont définies les métaclasses est `Class`. `Class` ajoute aux fonctionnalités définies par ses superclasses les notions de variables de classe et de variables de pool, et redéfinit certaines méthodes d'accès et de création de sous-classes.

Au niveau des objets

Le protocole `class membership` de la classe `Object` contient cinq méthodes qui permettent d'obtenir des informations sur la relation de l'objet à sa classe.

- `class` renvoie la classe à laquelle appartient le receveur.
- `isKindOf: uneClasse` (et `isKindOf:orOf:`) renvoie(nt) vrai si le receveur appartient à la (à l'une des deux) classe(s) passée(s) en argument ou à une superclasse.
- `isMemberOf:` se limite à vérifier l'appartenance directe du receveur à l'argument.

Par exemple, `2 isMemberOf: SmallInteger` renvoie vrai, ainsi que `2 isKindOf: Number`. `respondsTo: unSymbole` permet de savoir si un objet est capable de répondre à un message. Par exemple : `#(1 2) respondsTo: #at:put:` renvoie vrai, tandis que `#(1 2) respondsTo: #sin` renvoie faux.

Dans le protocole `message handling` on trouve toutes les méthodes qui permettent de faire exécuter un message dont le sélecteur et les arguments sont fournis en argument.

Par exemple, `2 perform: #sin` retourne le sinus de 2. `(Array new: 2) perform: #at:put: withArguments: #(1 4); yourself` renvoie le tableau `#(4 nil)`.

Rappel

La méthode `yourself` est intéressante en ceci que, lorsqu'un objet reçoit le message `yourself`, il se renvoie lui-même. Cette méthode est le plus souvent utilisée dans les situations similaires à celle que présente l'exemple précédent, où l'on souhaite accéder au receveur initial d'une cascade de messages, qui n'est pas renvoyé par le dernier message de la cascade.

La méthode `perform:` est très fréquemment utilisée. Elle permet par exemple d'associer aux options d'un menu les sélecteurs des méthodes qui doivent être déclenchés lorsque cette option est sélectionnée, et de faire exécuter la méthode. Elle peut également permettre d'éviter de longs enchaînements de tests, du type :

```
Case valeur1 : traitement 1
Case valeur2 : traitement 2
Case valeur3 : traitement 3
…
```

Il suffit en effet pour gérer ce type de situation de définir un dictionnaire qui associe les valeurs testées aux sélecteurs correspondants qu'il convient de déclencher, et de faire un accès direct par la valeur au dictionnaire, ce qui est à la fois plus rapide et plus lisible.

Un mécanisme similaire, mais utilisant des blocs, permet d'ailleurs l'implémentation du `case` : un `case` Squeak (méthodes `caseOf:` et `caseOf:otherwise:`) est en fait un ensemble d'associations (un dictionnaire donc) entre les valeurs du test `case` et celles des blocs qui doivent être exécutés lorsque cette valeur est rencontrée. Ce mécanisme ne devrait pas être utilisé en conception objet.

La gestion des erreurs

Les méthodes de gestion des erreurs (et de contrôle du déroulement de l'exécution) sont contenues dans le protocole `error handling` de `Object`.

On y trouve tout d'abord une définition plus complète que celle de `ProtoObject` de la méthode `doesNotUnderstand:`, qui provoque l'ouverture d'un débogueur. Par exemple, l'exécution de `25 asArray` provoque l'ouverture d'un débogueur (voir figure 7-1).

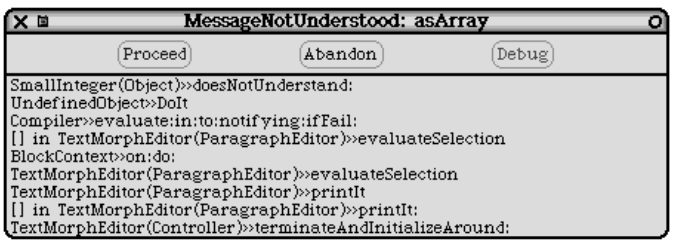


Figure 7–1

Ouverture d'un débogueur par la méthode `doesNotUnderstand:`

`Object>>doesNotUnderstand:` est définie de la façon suivante :

```
1  doesNotUnderstand: aMessage
2     (Preferences autoAccessors
3           and: [self tryToDefineVariableAccess: aMessage])
4        ifTrue: [^ aMessage sentTo: self].
5     MessageNotUnderstood new message: aMessage; signal.
6     ^ aMessage sentTo: self
```

La ligne 3 vérifie tout d'abord que la méthode incomprise ne correspond pas à un accesseur sur une variable d'instance du receveur. Une confirmation est demandée à l'utilisateur, et la méthode est créée si nécessaire.

La méthode `tryToDefineVariableAccess:`, qui effectue cette création, est intéressante à étudier, car elle illustre parfaitement les possibilités offertes par les capacités d'évaluation et de compilation dynamique de code de Squeak. Voici son code :

```
1  Object>>tryToDefineVariableAccess: aMessage
2  | ask newMessage sel |
3  aMessage arguments size > 1 ifTrue: [^ false].
4  sel := aMessage selector asString.
5  aMessage arguments size = 1
6     ifTrue: [sel last = $: ifFalse: [^ false].
7           sel := sel copyWithout: $:].
8  (self class instVarNames includes: sel)
9     ifFalse: [(self class classVarNames includes: sel asSymbol)
           ifFalse: [^ false]].
10 ask := self confirm: 'A ' , thisContext sender sender receiver class
      printString , ' wants to '
11                   , (aMessage arguments size = 1
12                            ifTrue: ['write into']
13                            ifFalse: ['read from']) , '
14 ' , sel , ' in class ' , self class printString , '.
15 Define a this access message?'.
16 ask
17       ifTrue: [
18         aMessage arguments size = 1
19             ifTrue: [newMessage := aMessage selector , ' anObject' ,
                     sel , ' anObject']
20             ifFalse: [newMessage := aMessage selector , '^' ,
                     aMessage selector].
21         self class
22             compile: newMessage
23             classified: 'accessing'
24             notifying: nil].
25 ^ ask
```

Les lignes 3 à 9 vérifient que le message est bien susceptible de correspondre à un accès à une variable d'instance. Si c'est le cas, les lignes 10 à 15 demandent à l'utilisateur de confirmer qu'il souhaite bien faire créer une nouvelle méthode pour la variable d'instance identi-

fiée. Dans l'affirmative, les lignes 19-20 créent le code de la méthode, qu'il s'agisse d'un mutateur (ligne 19), ou d'un simple accesseur (ligne 20). Les lignes 21 à 24 se chargent alors de compiler la méthode créée, et de l'ajouter à la classe idoine.

Remarque

Ce mécanisme qui permet la recompilation, voire la création, dynamique d'une méthode (ou d'une classe, puisque la création d'une classe résulte elle aussi d'un envoi de message), est très puissant. Il peut être utilisé par exemple pour mettre à jour une application avec de nouvelles versions sans arrêter l'application, pour assurer les mises à jour simultanées sur plusieurs postes de travail... Le code utilisé peut être téléchargé sur un serveur de code en permettant la mise à jour dynamique. Ce mécanisme est d'ailleurs utilisé pour la mise à jour de Squeak à partir des modifications validées par le « Squeak Central ». (Voir le bouton « load code update » de la barre d'outils Squeak.) Bien évidemment, l'utilisation de ce type de mécanisme nécessite de s'employer à un suivi des versions et d'y procéder avec cohérence, si on souhaite éviter les catastrophes !

La méthode `error:` provoque la levée d'une exception générique (appartenant à la classe `Error`), qui peut être récupérée par un gestionnaire d'erreur adéquat. La méthode `halt` est utilisée pour spécifier des points d'arrêt explicites dans le code. Elle provoque l'ouverture du débogueur, à partir duquel il est possible de tracer l'exécution.

La gestion des assertions (vérification en tête d'une méthode de conditions autorisant le déroulement du corps de la méthode) est assurée par la méthode `assert:`. Elle évalue le bloc passé en argument. Si l'évaluation renvoie faux, une exception de type `AssertionFailure` est levée. On peut en voir un exemple d'utilisation parmi d'autres dans la méthode `mergeSortFrom:to:by:` de la classe `ArrayedCollection` (lignes 5 et 6) :

```
1  ArrayedCollection>>mergeSortFrom: startIndex to: stopIndex by: aBlock
2     self size <= 1
3        ifTrue: [^ self].
4     startIndex = stopIndex ifTrue: [^ self].
5     self assert: [startIndex >= 1 and: [startIndex < stopIndex]].
6     self assert: [stopIndex <= self size].
7     self
8        mergeSortFrom: startIndex
9        to: stopIndex
10       src: self clone
11       dst: self
12       by: aBlock
```

La gestion des dépendances entre objets

Le mécanisme dit de gestion des dépendances a pour fonction de gérer les situations dans lesquelles un objet X qui dépend d'un autre objet Y doit être informé des changements d'état de Y, afin qu'il puisse s'adapter à ces changements et modifier son propre état interne en conséquence.

Les changements d'état d'un objet pourraient être notifiés à ses dépendants en insérant dans Y des méthodes qui informeraient X des changements survenus. Cette méthode n'est toutefois pas adéquate car elle oblige les concepteurs de Y à tenir compte de tous les objets qui peuvent en dépendre (ce qui, en outre, peut varier dynamiquement) et crée un couplage fort entre Y et ses dépendants. Cela rend la maintenance extrêmement problématique : des modifications importantes provoquent des effets en cascade difficiles à prévoir et à gérer, en particulier lorsque plusieurs types de propagation des modifications ont été utilisés.

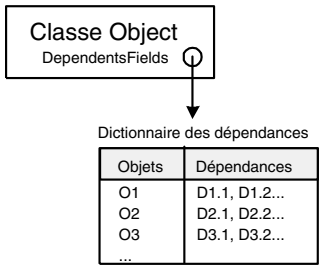
Une meilleure solution consiste à laisser à un mécanisme général le soin d'informer les dépendants des changements survenus, l'objet modifié ayant pour seul rôle d'informer ce mécanisme de tout changement de son état interne. Une utilisation « prototypique » de ce mécanisme se rencontre dans la relation entre un objet et les différentes vues qui présentent cet objet à l'écran : chaque vue est informée des changements de l'objet et se modifie en conséquence. Comme cela a déjà été mentionné, dans ce cas, on appelle *modèle* l'objet dont dépendent les différentes vues. Ce mécanisme est en particulier utilisé dans le paradigme de développement d'interfaces connu sous le nom de MVC, pour Modèle-Vue-Contrôleur.

Plusieurs mises en œuvre de ce mécanisme peuvent être notées en Squeak. L'une notamment implique l'utilisation d'une variable de classe sur `Object`, une autre s'appuie sur la classe `Model`.

Implémentation utilisant une variable de classe sur la classe Object

Dans la première de ces mises en œuvre (historiquement, la mise en œuvre initiale), la classe `Object` maintient dans une variable de classe, nommée `DependentsFields`, un dictionnaire de dépendances, nommé `DependentsFields` (voir figure 7–2).

Figure 7–2
La gestion des dépendances sur la classe Object



Dans cet exemple, `D1.1`, `D1.2` ... se déclarent dépendants de `O1`, ce qu'indique le dictionnaire d'association dans la variable de classe `DependentsFields` de `Object`.

Ce mécanisme permet à n'importe quel objet de se déclarer dépendant d'un autre objet (par la méthode `Object>>addDependent:`, protocole `dependents access`). `Object` définit également une méthode d'instance `changed` (protocole `updating`). Cette méthode, lorsqu'elle est déclenchée, envoie automatiquement un message `update:` à tous les dépendants de l'objet qui reçoit le message `changed`. Les objets dépendants doivent alors simplement définir une méthode `update:` qui effectue les actions correspondantes, actions qui consistent en général à demander la valeur des variables d'instances de leur modèle et à effectuer les opérations

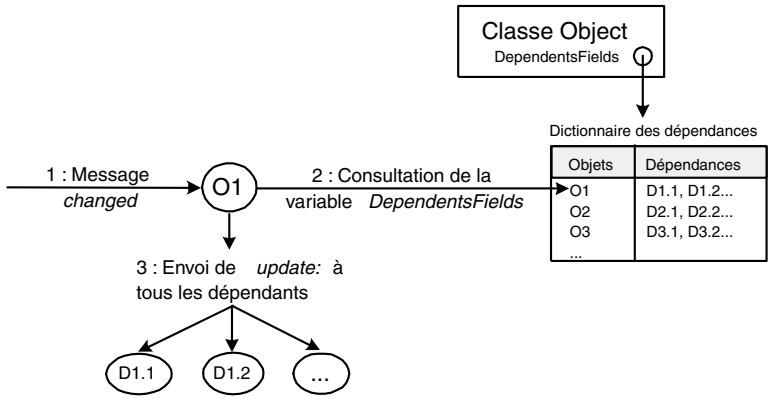


Figure 7–3
Propagation des notifications de changement sur les dépendants

permettant de mettre à jour leur état interne en fonction des modifications survenues. Bien entendu, dans la plupart des cas, il est souhaitable d'informer les dépendants de ce qui a changé, plutôt que de les informer simplement d'un changement indéfini. Pour ce faire, on utilise la méthode `changed:` ; elle reçoit en paramètre un objet qui indique au receveur ce qui a changé chez l'envoyeur.

Erreur fréquente

Les méthodes `changed`... envoyées à un objet lui indiquent qu'il *a changé* (et *non pas* qu'il doit ou *va changer*), et qu'en conséquence il doit en informer ses dépendants.

La méthode `myDependents`, définie au niveau de la classe `Object`, va chercher dans la variable de classe `DependentsFields` l'`IdentityDictionary` qui contient la liste des dépendants associés à l'objet receveur, à laquelle elle envoie le message `update`.

Les méthodes `update:`... sont par défaut définies comme… ne faisant rien, et doivent donc être redéfinies dans les sous-classes concernées. En d'autres termes, ces définitions impliquent le fonctionnement suivant :

Implémentation

Si on envoie... `changed`
alors ...le système envoie `self changed: self`
alors ...le système envoie à tous les dépendants `update: self`
Si on envoie avec argument... `changed: unAspect`
alors ...le mécanisme de dépendances envoie...
`update: unAspect` ...qui, par défaut, ne fait rien !"

La mise en œuvre des dépendances dans la classe `Object` a l'avantage de la généralité : tout objet du système peut dépendre de tout autre objet, ou avoir lui-même des dépendances. Elle présente pourtant plusieurs inconvénients :

- L'indirection engendrée par l'utilisation d'une variable de classe et d'un dictionnaire peut pénaliser les performances en cas de recours intensif aux dépendances.

> **Rappel**
>
> Les variables de classe sont stockées dans la variable d'instance de métaclasse `classPool` de la classe qui les définit. Ce stockage se fait sous la forme d'un dictionnaire associant le nom des variables à leurs valeurs. Tout accès à une variable de classe se fait donc de façon indirecte, par un accès à ce dictionnaire. Cette indirection, consommatrice de temps, peut pénaliser les performances.

- La référencement des dépendances par une variable de classe ne permet plus leur récupération par le ramasse-miettes. Ainsi, des objets qui ne sont référencés que dans une liste de dépendances ne seront pas récupérés même si l'objet dont ils dépendent n'existe plus en mémoire. On doit donc procéder à la gestion de la désallocation explicitement, ce qui est lourd et souvent très peu fiable, en particulier durant les premières étapes du développement d'un logiciel où les accidents qui pourraient laisser le système dans un état indéfini sont fréquents.

Implémentation utilisant la classe Model et une variable d'instance

Une deuxième mise en œuvre est donc proposée, qui utilise la classe `Model`. `Model` ajoute à `Object` la variable d'instance `dependents`, qui permet à chaque objet de stocker ses propres dépendances, plutôt que d'en passer par la variable de classe d'`Object`. Toute instance de la classe `Model` ou de l'une de ses sous-classes gère donc directement sa liste de dépendances, avec un fonctionnement tout à fait analogue à celui que nous venons de voir. Cette solution est préférable lorsque de nombreuses dépendances sont prévisibles sur un objet. Ainsi, dans la figure suivante, `D1`, `D2` ... se déclarent dépendants de `unModel`, directement à son niveau. À l'exception de cette modification technique, le mécanisme de gestion est identique.

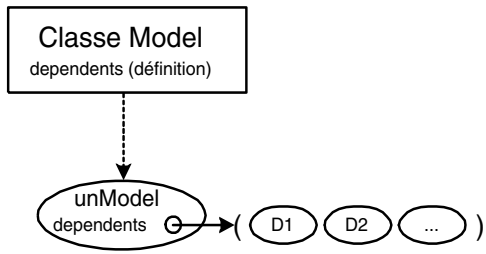


Figure 7–4
Implémentation de la gestion des dépendances sur la classe `Model`

La machine virtuelle, le compilateur

Un système Squeak comprend, nous l'avons vu, deux composants majeurs : l'image virtuelle et la machine virtuelle. L'image virtuelle contient l'ensemble des objets (y compris les classes et les méthodes compilées) du système. La machine virtuelle est l'exécutable qui permet de faire fonctionner cet ensemble d'objets et de méthodes sur une machine physique donnée.

Le compilateur

Le compilateur (classe `Compiler`) est un objet qui produit à partir d'une méthode contenant du code source Squeak une méthode compilée (`CompiledMethod`) qui renferme un ensemble d'instructions en assembleur pour la machine virtuelle. Ces instructions sont appelées *bytecodes*.

Par exemple, la méthode suivante

```
center
   ^origin + corner / 2
```

a pour *bytecodes* `0, 1, 176, 119, 185, 124`.

Voici la signification de ces valeurs numériques :

`0`	Empile la valeur de la première variable d'instance (`origin`) du receveur sur la pile
`1`	Empile la valeur de la seconde variable d'instance du receveur (`corner`) sur la pile
`176`	Envoie un message binaire avec le sélecteur `+`
`119`	Empile le `SmallInteger` 2
`185`	Envoie un message binaire avec le sélecteur `/`
`124`	Renvoie la valeur qui se trouve en sommet de pile en tant que résultat de la méthode (`center`).

Cet ensemble de *bytecodes* a été initialement défini dans le livre « bleu » (livre d'origine de Smalltalk, de Adèle Goldberg). Des extraits de ce livre fondateur sont disponibles sur le Web à l'adresse http://users.ipa.net/~dwighth/smalltalk/bluebook/bluebook_imp_toc.html.

Une fois engendré par le compilateur, le *bytecode* peut être interprété par la machine virtuelle. Un point intéressant est que la machine virtuelle Squeak est écrite… en Squeak puis traduite en C.

La machine virtuelle

Une implémentation complète, en Squeak, de cette machine virtuelle est en effet disponible dans l'environnement, dans l'application `VMConstruction-Interpreter`. S'y trouvent en particulier les classes `Interpreter`, laquelle contient l'ensemble des méthodes d'interpréta-

tion du *bytecode* (environ 450 méthodes), et la classe `ObjectMemory`, qui assure la gestion de la mémoire. C'est en particulier à ce niveau que sont définies les méthodes du `garbage collector`.

Comme le code complet de la machine virtuelle est écrit en Squeak, on peut la développer en utilisant toutes les facilités de l'environnement Squeak. Pour obtenir un exécutable réel qui puisse être utilisé sur une machine physique, un générateur de code C est fourni dans l'application `VMConstruction-Translation to C`. Ce générateur permet d'obtenir l'équivalent C d'un code Squeak. Par exemple, `Interpreter translate: 'interp.c' doInlining: true` permet de transformer en C le code de `Interpreter` et de ses super-classes. Le résultat (533 ko de code C) peut alors être compilé pour obtenir une machine virtuelle exécutable.

Attention

Seul un sous-ensemble de Squeak est directement traduisible en C. C'est avec ce sous-ensemble que l'implantation Squeak de la machine virtuelle est écrite.

À titre d'exemple, voici un extrait de la fonction `interpret()`, issue du code C généré par l'opération précédente. Il s'agit de la fonction principale d'exécution des *bytecodes*. Elle est constituée, en fait, d'une longue suite de `case`, chaque `case` correspondant à l'exécution d'un *bytecode*. L'extrait présenté est celui du *bytecode* 176 (correspondant au traitement du sélecteur binaire +) :

```
case 176:
         /* bytecodePrimAdd */
         t2 = longAt(localSP - (1 * 4));
         t3 = longAt(localSP - (0 * 4));
         if (((t2 & t3) & 1) != 0) {
            t1 = ((t2 >> 1)) + ((t3 >> 1));
            if ((t1 ^ (t1 << 1)) >= 0) {
               /* begin internalPop:thenPush: */
               longAtput(localSP -= (2 - 1) * 4, ((t1 << 1) | 1));
               /* begin fetchNextBytecode */
               currentBytecode = byteAt(++localIP);
               goto l45;
            }
         } else {
            successFlag = 1;
            /* begin externalizeIPandSP */
            instructionPointer = ((int) localIP);
            stackPointer = ((int) localSP);
            theHomeContext = localHomeContext;
            primitiveFloatAddtoArg(t2, t3);
            /* begin internalizeIPandSP */
            localIP = ((char *) instructionPointer);
            localSP = ((char *) stackPointer);
```

```
            localHomeContext = theHomeContext;
            if (successFlag) {
               /* begin fetchNextBytecode */
               currentBytecode = byteAt(++localIP);
               goto l45;
            }
         }
         messageSelector = longAt(((((char *) (longAt(((((char *)
            specialObjectsOop)) + 4) + (23 << 2))))) + 4) + ((0 * 2) << 2));
         argumentCount = 1;
         /* begin normalSend */
         goto commonSend;
```

Comme on peut le constater, Squeak est vraiment un langage totalement ouvert. Tous les éléments du langage, de l'environnement et de la machine virtuelle sont accessibles, et peuvent être consultés, modifiés, améliorés… C'est cette base exceptionnelle qui est offerte à tout développeur suffisamment motivé pour en acquérir la maîtrise, ce qui est évidemment loin d'être trivial, mais reste à la portée de tout informaticien ! C'est, entre autres atouts, ce qui permet à Squeak d'être porté sur un nombre considérable de plates-formes, allant de la station de travail Unix au PDA, en passant par les PC et les Macintosh. Il est important de rappeler qu'une image Squeak est totalement portable (y compris l'interface graphique), sans aucune modification, sur toute plate-forme qui dispose d'une machine virtuelle Squeak.

En résumé

Le « noyau système » est la partie la plus technique de Squeak. Incluant les classes du sommet de la hiérarchie, la machine virtuelle, la gestion des processus, le compilateur, le `garbage collector`, la gestion des entrées-sorties, c'est cette machinerie qui fait fonctionner les classes et les méthodes de plus haut niveau.

Après une rapide présentation des entrées-sorties, essentiellement fondées sur la gestion des flots de données, les mécanismes fondamentaux de gestion des classes, des objets et des pointeurs, assurés par les classes du sommet de la hiérarchie, ont ensuite été détaillés.

Nous avons ensuite abordé la gestion des dépendances entre objets, la gestion des erreurs, la création de classes, ainsi que les fondements de l'exécution d'un programme Squeak, la machine virtuelle et le compilateur, dont nous avons montré que bien qu'écrits en Squeak (et donc aisément modifiables et compréhensibles), ils pouvaient très aisément être transcodés en C, et recompilés sur toute plate-forme.

Dans les chapitres suivants, nous complétons la cartographie du noyau système de Squeak en présentant les applications de la réflexivité et la gestion des processus.