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

Ce chapitre concerne toutes les opérations qui dépassent le 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`…

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

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:\Documents and Settings\Xavier Brifffault\Mes documents\Projets\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 := (FileDirectory on: 'C:\Documents and Settings\Xavier Brifffault\Mes
documents\Projets\Eyrolles\Squeak\SQUEAK 3\Images de travail\1') fileNamed: 'test.txt'.
 fic nextPutAll: 'chaîne de test'.
 fic close.
 fic := (FileDirectory on: 'C:\Documents and Settings\Xavier Brifffault\Mes
documents\Projets\Eyrolles\Squeak\SQUEAK 3\Images de travail\1') readOnlyFileNamed:
'test.txt'.
 fic contentsOfEntireFile.  «Renvoie 'chaîne de test'»
```

Plusieurs classes utilisées pour la compression/ décompression de données sont également disponibles (catégorie `System-Compression`), qui 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 également 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`.

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 le monde 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.

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` 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).

Y est également définie la méthode `doesNotUnderstand:`, qui 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…

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 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`.

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 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: selector withMethod: compiledMethod`.

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 plus bas la section consacrée à la machine virtuelle pour plus de détails). Ce compilateur accessible, 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

http://www.cc.gatech.edu/projects/squeakers/25.html,
http://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
Case valeur4 : traitement 4
…
```

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. Par exemple, la séquence de code suivante renvoie `4` (l'évaluation du bloc associé à `#b`) :

```
#b caseOf: {[#a]->[1+1]. ['b' asSymbol]->[2+2]. [#c]->[3+3]}
```

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` fait apparaître la fenêtre de la figure suivante :

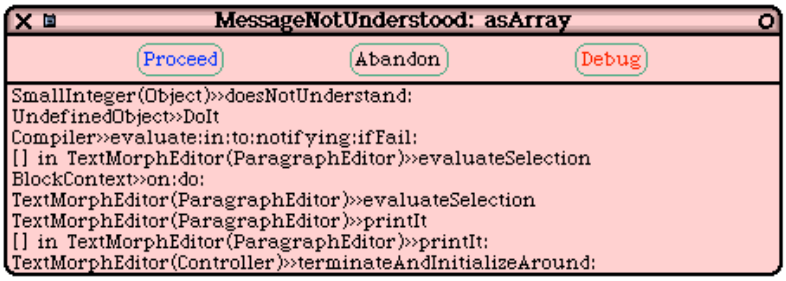

Figure 7-1. Ouverture d'un débogueur par la méthode doesNotUnderstand

`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 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 identifiée. 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 qu'il doit utiliser 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:`.

`assert:` 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) :

```
mergeSortFrom: startIndex to: stopIndex by: aBlock
     self size <= 1
        ifTrue: [^ self].
     startIndex = stopIndex ifTrue: [^ self].
     self assert: [startIndex >= 1 and: [startIndex < stopIndex]].
     self assert: [stopIndex <= self size].
     self
        mergeSortFrom: startIndex
        to: stopIndex
        src: self clone
        dst: self
        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 suivante).

Dans cet exemple, `X1`, `X2`, et `X3` se déclarent dépendants de `Y`, ce qu'indique l'association entre `Y` et `X1`, `X2`, `X3` 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`.

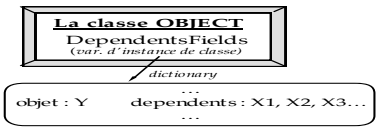

Figure 7-2. La gestion des dépendances sur la classe Object

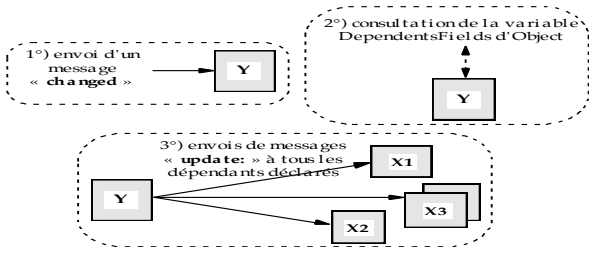

Figure 7-3. Propagation des notifications de changement sur les dépendants

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 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, `X1`, `X2`, `X3` se déclarent dépendants de `Y`, directement à son niveau. À l'exception de cette modification technique, le mécanisme de gestion est identique.

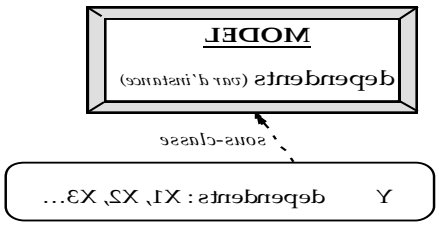

Figure 7-4. Implémentation de la gestion des dépendances sur la classe Model

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 aux adresses suivantes :

http://users.ipa.net/%7Edwighth/smalltalk/bluebook/bluebook_chapter26.html et
http://users.ipa.net/%7Edwighth/smalltalk/bluebook/bluebook_chapter26.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.

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'interprétation 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.

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 présente

en fait 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.

8

Notions avancées : application de la réflexivité en Squeak

Les éléments techniques de Squeak que nous avons présentés dans les chapitres 4 et 7 en font un langage *réflexif*. Cela signifie que ce langage permet d'une part *l'introspection*, c'est-à-dire l'analyse des structures de données qui définissent le langage lui-même, et d'autre part *l'intercession*, c'est-à-dire la modification depuis le langage lui-même de sa sémantique et de son comportement.

On notera ainsi que, bien qu'il définisse une interface réflexive, Java n'est pas réflexif et ne permet qu'une introspection limitée. Nous allons montrer dans cette section quelques exemples des aspects réflexifs de Squeak, sur la base d'exemples pratiques, susceptibles d'aider les développeurs dans leurs développements quotidiens.

Accès à la structure d'un objet par un autre objet : le cas de l'inspecteur

Dans les chapitres précédents, nous avons abordé la notion d'inspecteur ; un inspecteur est un petit outil de développement qui permet de visualiser et de modifier les valeurs des variables d'instances d'un objet, et de lui envoyer des messages.

Exemple de fonctionnement

En évaluant le code suivant, voici l'inspecteur que l'on obtient sur un objet `Workspace` (voir figure 8-1) :

```
|inst |
inst := Workspace new.
inst openLabel: 'Workspace'.
inst inspect
```

Figure 8-1. Un inspecteur sur un Workspace

L'inspecteur de la figure 8-1 présente dans sa partie gauche les variables d'instances (ici, `dependents`, `contents` et `bindings`) de l'objet inspecté (ici, un `Workspace`), et la valeur de la variable sélectionnée dans la partie droite (ici, une chaîne de caractères). Si vous sélectionnez la variable `contents` qui représente le texte contenu dans le `Workspace`, vous obtenez une chaîne vide ; le `Workspace` que nous avons ouvert ne contient en effet pas de texte.

Saisissez à présent `'1+2'` à la place de la chaîne vide, puis faites **<**accept**>**. La valeur de la variable `contents` a changé. Pour voir l'effet de cette modification dans le `workspace`, évaluer (<do it>) `self contentsChanged` dans la partie basse de l'inspecteur, pour notifier les changements à la fenêtre. En effet, l'accesseur `contents:` ne propage pas automatiquement ces changements.

On peut se demander comment un tel inspecteur fonctionne.

En effet, en Squeak, les variables d'instances sont protégées. Si une classe ne définit pas d'accesseurs, il est théoriquement impossible d'accéder à ses variables. Pourtant, l'inspecteur permet d'accéder à n'importe quelle variable d'instance de n'importe quel objet.

Analyse du fonctionnement d'un inspecteur

Il utilise pour cela les possibilités réflexives de Squeak. Les méthodes `instVarAt:`, `instVarAt:put:`, `instVarNamed:`, `instVarNamed:put:`, définies sur la classe `Object`, permettent en effet d'accéder à la valeur d'une variable d'instance. `instVarAt: unEntier` permet d'accéder à la variable d'instance de position `unEntier`. `instVarAt: unEntier put: uneValeur` permet de modifier la variable d'instance de position `unEntier`. `instVarNamed: uneChaine` permet d'accéder à la variable d'instance représentée par `uneChaine`. `instVarNamed: uneChaine put: uneValeur` permet de changer la valeur de la variable d'instance représentée par `uneChaine`.

On notera que ces méthodes sont utiles pour construire des outils de développement ou de débogage, mais qu'il vaut mieux ne pas les utiliser lors du développement d'applications.

Le code suivant montre comment on peut afficher dans le `Transcript` (menu <open> / <Transcript>) les valeurs des variables d'instances d'une instance de la classe `SystemWindow` sélectionnée au hasard (`someInstance`). La méthode `allInstVarNames` permet d'obtenir tous les noms des variables d'instances d'une classe.

```
| instance |
instance :=  SystemWindow someInstance.
instance class allInstVarNames
    do: [:each | Transcript
          nextPutAll: each ;
          nextPutAll: '  -> ';
          nextPutAll: (instance instVarNamed: each) asString; cr.].
Transcript flush
```

Dans le même esprit, l'expression suivante permet d'obtenir toutes les instances de la classe `Browser` dont la variable d'instance `systemOrganizer` n'est pas `nil`.

```
Browser allInstances select: [ :inst | (inst instVarNamed: 'systemOrganizer') notNil. ]
```

Examinons à présent l'implantation de la méthode `instanceVariableValues` définie sur la classe `Object`. `instanceVariableValues` rend un tableau dont les éléments sont les valeurs des variables d'instances définies exclusivement par la classe de l'objet receveur (à l'exclusion de celles des variables héritées). Cet exemple montre combien il est utile de pouvoir accéder aux variables par leur position. La méthode `intSize` renvoie le nombre de variables d'instance d'une classe.

L'évaluation de `(1@2) instanceVariableValues` rend `anOrderedCollection(1 2)`. La méthode `instanceVariableValues` est définie comme :

```
Object>>instanceVariableValues
| c |
c := OrderedCollection new.
self class superclass instSize + 1
1 to: self class instSize do: [:i | c add: (self instVarAt: i)].
^ c
```

Précisons une nouvelle fois que toutes ces fonctionnalités sont intéressantes pour la définition d'outils de programmation, mais que leur utilisation est sujette à caution pour le développement d'applications finales. Le respect des principes d'encapsulation est impératif. Il ne saurait être question dans une application finale d'essayer de contourner ces règles en utilisant les possibilités offertes par le langage.

Squeak offre également d'autres possibilités d'introspection qui ne seront pas détaillées ici, sur des objets tels que les méthodes, les piles d'exécution réifiées, les processus, le gestionnaire de mémoire...

Utilisation de la réflexivité sur les classes

Comme cela a été montré dans les exemples précédents, il est possible d'obtenir une instance quelconque d'une classe (`someInstance`). Il est aussi possible d'obtenir toutes les instances d'une classe (`allInstances`), ainsi que le nombre d'instances en mémoire (`instanceCount`).

```
SystemWindow instanceCount
```

Examinons maintenant les nombreuses possibilités de références croisées et de navigation entre méthodes et classes offertes par les classes Squeak. L'exemple suivant retourne la liste des classes dont le commentaire contient la chaîne de caractères `'URL'`.

```
Squeak allClasses select: [:aClass | aClass comment asString includesSubString: 'URL']
```

Voici quelques exemples qui illustrent la facilité avec laquelle il est possible de construire des outils de navigation. Essayez les expressions suivantes :

- `Point whichSelectorsAccess: 'x'` rend la liste des méthodes de la classe `Point` qui accèdent à la variable d'instance `x`.
- `Point whichSelectorsStoreInto: 'x'` rend la liste des méthodes qui accèdent en écriture à la variable d'instance `x`.
- `Rectangle whichSelectorsReferTo: #+` rend la liste des méthodes définies sur la classe `Rectangle` qui invoquent la méthode +.

- `Point crossReference` rend un tableau dont les lignes représentent tous les messages invoqués par la méthode.
- `Rectangle whichClassIncludesSelector: #+` rend les classes qui définissent la méthode `+`.
- `Rectangle unreferencedInstanceVariables` rend la liste de variables d'instances qui ne sont pas référencées dans la classe ou ses sous-classes, et qui sont donc potentiellement inutiles.

La classe `SystemDictionary`, qui représente l'espace de nommage dans lequel les classes et les variables globales sont définies, propose un grand nombre de fonctionnalités de références croisées et navigation. `Smalltalk` est une variable globale qui référence l'espace de nom de base (Squeak n'utilise dans la version 3.0 qu'un seul espace de noms).

Essayez par exemple les expressions suivantes :

- `Squeak allClassesImplementing: #+`
- `Squeak allSentMessages`
- `Squeak allUnSentMessages`
- `Squeak allUnimplementedCalls`
- `Squeak allCallsOn: (Squeak associationAt: Point name)`

Cette dernière expression rend toutes les références à la classe `Point`.

Comment calculer automatiquement des métriques sur le code ?

Il est possible d'utiliser les capacités introspectives de Squeak pour définir rapidement des métriques.

Lorsqu'on aborde une application inconnue, le calcul des métriques permet d'obtenir une première idée de cette application. Les métriques peuvent également donner des indices quantitatifs qui permettent de juger de la rapidité d'avancement d'un projet, mais aussi fournir des indices de qualité ou de respect de normes de développement.

Voici quelques métriques simples que l'on peut calculer : le niveau d'héritage, le nombre de méthodes, le nombre de variables d'instances, le nombre de méthodes définies localement, le nombre de variables d'instances ajoutées localement, le nombre de classes, ainsi que le total des sous-classes.

Les expressions suivantes illustrent comment ces métriques peuvent être calculées.

```
niveauDHeritage := Browser allSuperclasses size.
nbMethodes := Browser allSelectors size.
nbInstances := Browser allInstVarNames size.
nbMethodesAjoutees := Browser selectors size.
nbInstanceAjoutees := Browser instVarNames size.
nbSouclasses := Browser subclasses size.
nbTotalSouclasses := Browser allSubclasses size.
```

Un des métriques intéressantes dans le domaine des langages est celle qui détermine le nombre de méthodes qui étendent des méthodes héritées de leur superclasse, c'est-à-dire qui invoquent une méthode masquée à l'aide de la pseudo-variable `super`. Cette opération permet d'avoir une meilleure compréhension de la relation qui existe entre une classe et ses superclasses.

Voici un exemple de code qui montre comment on peut identifier les méthodes de la classe `Browser` qui font un appel à une méthode cachée, c'est-à-dire une invocation *via* `super` ayant la forme suivante :

```
Browser>>xx
     ...
     super xx
```

Le code renvoie sur cet exemple les méthodes: `#systemOrganizer:` et `#veryDeepInner:`. Le nombre d'invocation à des méthodes masquées est donc simplement la taille de la collection rendue, ici 2.

```
      Browser selectors select: [:selector || method | method := Browser compiledMethodAt:
selector. method sendsToSuper]
```

Détection automatique de bogues possibles

Invoquer une méthode masquée *via* la pseudo-variable `super` en utilisant un nom de méthode différent de celui de la méthode qui contient `super` peut créer des problèmes lorsque la classe est spécialisée. Il est préférable d'éviter ce genre de code.

Une recherche automatique de méthodes qui ne respecteraient pas cette règle peut être réalisée très simplement : il suffit en effet de trouver les méthodes effectuant un appel *via* `super` qui ne contiennent pas le sélecteur de la méthode analysée :

```
 Browser selectors
select: [:selector |
| method |
method := Browser compiledMethodAt: selector.
(method sendsToSuper and:
[(method messages includes: selector) not])]
```

En exécutant cet exemple, vous devez obtenir la méthode `systemOrganizer`, qui devrait effectivement invoquer la méthode `initialize` *via* `self`, et non *via* `super`. On notera que ce code ne détecte pas toutes les méthodes problématiques. En particulier, les méthodes qui contiennent des boucles ou des récursions, telle celle qui est présentée ci-après, ne sont pas détectées.

```
xxx
  super yyy
  self xxx "boucle"
```

Le code suivant permet de trouver toutes les méthodes qui ne sont pas nécessaires, car elles sont identiques à une méthode déjà définie dans une des super-classes `ProtoObject`, `allSubclassesDo:$$$`

```
[:class |
     class selectors do:
      [:selector |
           | currentMethod superMethod |
           currentMethod := class compiledMethodAt:selector.
           superMethod := class superclass lookupSelector:selector.

           currentMethod = superMethod
```

```
            ifTrue:[Transcript
                        show: class;
                        show: '>>';
                        show: selector;
                        cr]]]
```

Mise en œuvre de l'intercession en Squeak

Squeak étant un langage *réflexif*, il permet donc aussi *l'intercession*, à savoir la modification par le langage lui-même de sa sémantique et de son comportement. C'est ce dernier point que nous allons maintenant aborder.

Nous l'avons souligné à maintes reprises dans les précédents chapitres, Squeak est entièrement développé en langage Squeak. De nombreux aspects du système peuvent ainsi être aisément modifiés, entre autres l'envoi de message, la façon dont le ramasse-miettes fonctionne, les processus, l'ordonnanceur de tâches, la pile d'exécution...

Dans ce chapitre, nous nous limiterons à des exemples relatifs au contrôle du comportement des objets et qui peuvent faciliter le développement en phase de prototypage. Nous montrerons en particulier comment la réflexivité est utilisée pour construire des fonctionnalités qui aident au prototypage d'applications et à la création d'outils d'espionnage.

Avertissement

Les interventions et modifications sur les couches « basses » de Squeak (envoi de message, héritage, gestion des classes...) peuvent produire des résultats très puissants. Elles peuvent aussi totalement détruire votre environnement Squeak ! Il est donc important lors des développements à ce niveau de prendre soin des sauvegardes intermédiaires du code et de l'image. Rappelons néanmoins que tant que l'image n'a pas été sauvegardée, il suffit de quitter l'environnement sans procéder à une sauvegarder pour le retrouver tel quel.

Quelques notions techniques préliminaires

Nous allons tout d'abord illustrer trois notions que nous utiliserons par la suite : le changement de référence (ou d'identité), la récupération d'erreur et l'invocation explicite de méthodes.

Changement d'identité d'un objet

En Squeak, il est possible de substituer toutes les références qui pointent sur un objet à celles pointant sur un autre. C'est la tâche que réalise la méthode `become:`.

On peut ainsi substituer un objet à un autre. On dit aussi alors qu'un objet en devient un autre ; en effet, du point de vue du programme, une variable pointant initialement sur un objet (`pt1` dans l'exemple) pointe sur un autre (`pt2` dans l'exemple), après exécution du code de substitution (`pt1 become: pt2`).

```
|pt1 pt2|
pt1 := 0@0.
pt2 := 10@10.
Transcript show: 'pt1 ' ; show: pt1 printString; cr.
Transcript show: 'pt2 ' ; show: pt2 printString; cr.
pt1 become: pt2. "à partir d'ici, pt1 pointe sur pt2 et pt2 sur pt1"
```

```
Transcript show: 'pt1 ' ; show: pt1 printString; cr.
Transcript show: 'pt2 ' ; show: pt2 printString; cr.
```

Récupération d'erreur

On dispose de différentes techniques pour contrôler les messages envoyés à un objet, et pour créer des messages asynchrones, distribués, ou des outils d'espionnage. Nous expliquons ici la méthode la plus simple qui est fondée sur la récupération d'erreur.

Lorsqu'un message `m` est envoyé à un objet, si la classe ou les super-classes de cet objet n'implantent pas de méthode `m`, la méthode `doesNotUnderstand:` est invoquée par la machine virtuelle sur l'objet receveur du message. Par défaut, la méthode `doesNotUnderstand:` lève une exception qui aboutit à l'ouverture d'un débogueur.

Figure 8-2. Mécanisme d'invocation de la méthode `doesNotUnderstand:`

En rendant cette méthode spécifique, on peut donc contrôler les messages qui ne sont pas compris par un objet. Pour contrôler les méthodes existantes, une des techniques consiste alors à créer un objet minimal qui n'implante que les méthodes vitales et provoque donc l'invocation de la méthode `doesNotUnderstand:` pour toutes les autres. En Squeak, la classe `ProtoObject` est une classe minimale. L'implantation d'un espion est fondée sur cette technique.

Invocation explicite de méthodes

Il est également possible d'invoquer l'exécution d'une méthode à l'aide de la méthode `perform:withArguments:` définie sur la classe `Object`.

En invoquant cette méthode sur un objet, avec pour argument le sélecteur de la méthode à invoquer et un tableau qui représente les arguments, la méthode est recherchée dans l'arbre d'héritage, puis exécutée, comme cela serait le cas avec une invocation directe. `1+2` peut être donc représentée comme `1 perform: #+ withArguments: #(2).`

Squeak permet également d'exécuter directement une méthode sans que l'on doive la rechercher. La classe `CompiledMethod` définit en effet la méthode `#valueWithReceiver:arguments:`. Cette méthode doit être invoquée directement sur une instance de la classe `CompiledMethod` qui représente les méthodes, à laquelle on doit préciser le receveur et les arguments de l'appel. Exécuter `1+2` revient alors à trouver la méthode `#+`, puis à l'invoquer sur `1` avec pour argument `2`,comme suit :

```
(1 class compiledMethodAt: #+)
     valueWithReceiver: 1
        arguments: #(2)
```

Des accesseurs invisibles pour l'aide à la mise au point

Lors des premières phases d'un développement, il est fréquent que la structure de l'application développée ne soit pas vraiment stable. Il est alors souvent nécessaire d'éditer la définition des

classes, de changer les accesseurs à chaque fois que l'on se rend compte que l'on a besoin de nouvelles variables.

Pour éviter de devoir effectuer trop fréquemment ces modifications, une solution simple consiste à utiliser un dictionnaire afin de stocker les variables d'instances et les variables de ces dernières.

Redirection automatique des messages d'accès

Dans l'exemple suivant, nous définissons la classe `Stuff` comme suit, avec la variable `variables` que nous initialiserons avec un dictionnaire :

```
Object subclass: #Stuff
   instanceVariableNames: 'variables'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Scaffolding'
```

Définissez ensuite la méthode `initialize`, ainsi que la méthode de classe `new` comme ceci :

```
Stuff>>initialize
   variables := IdentityDictionary new.
```

```
Stuff class>>new
   ^ super new initialize
```

On pourrait alors définir un accesseur, par exemple `widget`, de la façon suivante :

```
Stuff>>widget
     ^ variables at: #widget
```

```
Stuff>>widget: aWidget
     variables at: #widget put: aWidget
```

Cependant, cette façon de procéder implique que l'on définisse d'une façon manuelle des accesseurs, ou que l'on utilise explicitement le dictionnaire, ce qui est pour le moins ennuyeux. Ce que nous souhaiterions ici, c'est pouvoir utiliser un accesseur sans devoir le créer, et que l'accesseur accède au dictionnaire automatiquement et de façon transparente. Nous souhaiterions donc que `widget` soit automatiquement transformé en `variables at: #widget`.

L'une des solutions est de spécialiser la méthode `doesNotUnderstand:` afin de récupérer les messages incompris (correspondant aux accesseurs non définis) et de les convertir automatiquement en accès à notre dictionnaire. Pour ce faire, définissez la méthode `doesNotUnderstand:` comme suit :

```
Stuff>>doesNotUnderstand: aMessage
| key |
aMessage selector isUnary
   ifFalse: [
   key := (aMessage selector copyWithout: $:) asSymbol.
       variables at: key put: aMessage arguments first]
ifTrue: [^ variables at: aMessage selector ifAbsent: [nil]]
```

L'argument de cette méthode, `aMessage`, est une représentation objet d'un message. Il contient le receveur (`aMessage receiver`), le sélecteur du message (`aMessage selector`) et la liste des arguments (`aMessage arguments`). Notons que cette transformation en objet du message

(réification) n'est effectuée qu'en cas d'erreur, et ce afin de ne pas pénaliser l'exécution normale d'un programme.

La méthode de transformation vérifie tout d'abord si la méthode a un argument, ou pas. Lorsque c'est le cas, elle extrait la variable du nom du sélecteur de la méthode, puis met dans le dictionnaire la valeur associée à cette variable. On notera que cette implantation ne vérifie pas si la méthode est bien un accesseur. En effet, `aStuff xxx: 12 yyy: 13` va mettre `13` dans la variable `xxx:yyy`. La résolution de ce problème est laissée à l'initiative du lecteur à titre d'exercice. Pour ce faire, il faut bien considérer qu'il est possible d'insérer un `self halt` à l'intérieur de la méthode et d'inspecter son argument.

Génération automatique d'accesseurs

Une fois que la phase de prototypage est terminée, nous souhaiterions pouvoir analyser le dictionnaire de toutes les instances disponibles et générer automatiquement la définition de la classe et les accesseurs. Dans un premier temps, nous allons vous proposer de générer automatiquement le code des accesseurs. Nous vous laisserons à titre d'exercice le soin d'imaginer des solutions pour changer la définition de la classe afin de remplacer le dictionnaire par les variables effectivement utilisées, en mettant en œuvre les techniques présentées plus haut.

Pour générer automatiquement les accesseurs, nous remplaçons la méthode `doesNotUnderstand:` afin qu'elle compile des accesseurs d'une manière automatique.

```
Stuff>>doesNotUnderstand: aMessage
| key |
key := aMessage selector.
key isUnary
     ifTrue: [
         self class compile: (self textOfGetterFor: key asString) classified:
'accessors'.
         ^ self  perform: key withArguments: aMessage arguments].
   (key isKeyword and: [ key numArgs = 1 ])
     ifTrue: [
        self class compile: (self textOfSetterFor: key asString) classified:
'accessors'.
        ^ self perform: key withArguments: aMessage arguments].
^ super doesNotUnderstand: aMessage
```

La méthode compile les accesseurs puis redemande l'exécution de l'accesseur originellement invoqué. Cette méthode est plus sûre que la précédente : en effet, nous vérifions que le message non compris est bien un message unaire ou un message à mots-clés avec un seul argument. Ainsi, l'invocation de la méthode `xxx:yyy:` provoque bien une erreur, au lieu d'invoquer une méthode compilée à tort.

Les deux méthodes suivantes génèrent le code des méthodes qui seront compilées.

```
Stuff>>textOfGetterFor: aString
|stream|
stream := ReadWriteStream on: (String new: 16).
stream nextPutAll: aString .
stream nextPut: Character cr ; nextPut: Character tab.
stream nextPutAll: '^ variables at: #', aString.
^ stream contents
```

```
Stuff>>textOfSetterFor: aString
|stream|
stream := ReadWriteStream on: (String new: 16).
stream nextPutAll: aString, ': anObject'  .
stream nextPut: Character cr ; nextPut: Character tab.
stream nextPutAll: ' variables at: #', aString, ' put: anObject'.
^ stream contents
```

Ainsi, lorsque le message `widget` est envoyé à une instance de la classe `Stuff`, l'accesseur

```
widget
    ^ variables at: #widget
```

est compilé.

Mécanismes d'espionnage

Un des exemples typiques d'utilisation du contrôle de l'envoi de messages est l'espionnage des objets à des fins d'optimisation ou de compréhension des applications. Un espion est un objet que l'on substitue à l'objet que l'on souhaite espionner.

Création d'une classe d'espions

Pour créer un tel espion, nous allons définir une sous-classe `Spy` de `ProtoObject`, ayant comme variable d'instance `spyedObject`, qui représente l'objet espionné. Rappelons que `ProtoObject` est une classe minimale qui ne sait répondre qu'à un nombre extrêmement limité de messages, et génère des erreurs, que nous allons capturer, pour tous les autres messages.

```
ProtoObject subclass: #Spy
    instanceVariableNames: 'spyedObject '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Scaffolding'
```

Définissons tout d'abord les méthodes d'instance `on:` et `doesNotUnderstand:` de la façon suivante :

```
Spy>>on: anObject
        spyedObject := anObject
```

```
Spy>>doesNotUnderstand: aMessage
      Transcript
            nextPutAll: 'sel>> '; nextPutAll: aMessage selector printString;
            nextPutAll: ' args>> '; nextPutAll: aMessage arguments printString; cr; flush.
      ^ spyedObject perform: aMessage selector withArguments: aMessage arguments
```

Nous définirons ensuite la méthode de classe `on:` qui installe un espion sur un objet donné.

```
Spy class>>on: anObject
    | spy |
    spy := self new.
    spy become: anObject.
    anObject on: spy.
    ^ anObject
```

Une fois que l'expression `spy become: anObject` est exécutée, la variable `spy` pointe sur l'objet et la variable `anObject` pointe l'instance de `Spy` nouvellement créée. Ouvrez alors un `Transcript` (menu open... Transcript) et évaluez l'expression suivante dans un `Workspace` (menu open... Workspace).

```
|t |
t := OrderedCollection new.
Spy on: t.
t add: 3.
t add: 5
```

Une trace des messages envoyés apparaît alors dans le `Transcript`.

Quelques limites au mécanisme d'espionnage

Cette technique présente les limitations suivantes :

- On ne peut pas espionner les classes car elles ne supportent pas de `become:`.
- Si un objet envoie un message à l'espion qui le renvoie à l'objet espionné et que celui-ci retourne `self`, on peut alors envoyer des messages à ce résultat et donc court-circuiter l'espion.
- De la même façon, si un objet espionné est stocké dans une collection (par exemple, un dictionnaire), la différence entre les clés de hachage que présentent l'espion et l'objet espionné peut provoquer des comportements très étranges.
- Cette technique peut également avoir des résultats imprévisibles avec les morphs.

Une version plus sophistiquée de la méthode `doesNotUnderstand:` peut aussi afficher l'objet qui a envoyé le message espionné, grâce à l'utilisation de la pseudo-variable `thisContext` qui réifie à la demande la pile d'exécution.

```
Spy>>doesNotUnderstand: aMessage
      Transcript nextPutAll: 'sel>> ' ;  nextPutAll: aMessage selector printString ;
               nextPutAll: ' args>> '; nextPutAll: aMessage arguments printString.
      Transcript nextPutAll: ' sent by ', thisContext sender printString ; cr; flush.
      ^ spyedObject perform: aMessage selector withArguments: aMessage arguments
```

Wrappers et Proxies : `ObjectOut`, `ObjectTracer` et `ObjectViewer`

Squeak utilise cette technique (permutation de pointeurs) dans la création de *wrappers* (encapsulateurs qui ajoutent des comportements à des objets) et de *proxies* (représentants locaux d'objets distants).

Wrappers

La classe `ObjectOut` (sous-classe directe de `ProtoObject`, et non d'`Object`) implémente un tel mécanisme. Un `ObjectOut` est un représentant (que nous appellerons `Oo`) en mémoire d'un objet (que nous appellerons `Oi`, pour objet initial) qui a été stocké sur disque, ou sur une machine distante (par exemple, pour libérer de la mémoire), et qui n'est donc plus disponible pour recevoir les messages qui lui sont adressés. Lorsque `Oi` doit être déchargé de la mémoire sur le disque, en utilisant le format de stockage linéarisé proposé par Squeak (voir remarque ci-après), `Oi` est permuté avec `Oo`, et l'URL où a été stocké `Oi` est affectée à `Oo`. Toutes les variables qui pointaient

sur `Oi` pointent désormais sur `Oo`. `Oi` n'étant plus référencé, il sera détruit lors du prochain passage du `garbage collector`.

Implémentation : stockage des objets sur le disque

Squeak dispose d'un format binaire pour stocker les objets, mais aussi d'un format linéaire, qui recrée l'enchaînement de méthodes nécessaire pour recréer l'objet stocké à partir de sa représentation. Ce format est obtenu à partir de la méthode `storeString`.

Par exemple, à partir d'une `OrderedCollection` créée par `oc := OrderedCollection new add: 4; add: 5; remove: 4; yourself` (et qui contient donc uniquement le nombre 5), le format linéaire de stockage (`oc storeString`) est `'((OrderedCollection new) add: 5; yourself)'`, dont l'évaluation reconstruit effectivement l'objet d'origine.

Lorsque les accointances (objets collaborateurs) de `Oi`, ignorant sa disparition de la mémoire, lui enverront un message, ce message parviendra en fait à `Oo`. `Oo` n'implémentant pas la méthode correspondante (le sous-classement de `ObjectOut` sous `ProtoObject` garantissant que `Oo` ne « comprend » vraiment rien), la machine virtuelle déclenchera la méthode `doesNotUnderstand:`, qui est redéfinie à ce niveau pour prendre en charge le rechargement de `Oi` depuis le disque :

```
ObjectOut>>doesNotUnderstand: aMessage
| realObject oldFlag response |
oldFlag := recursionFlag.
recursionFlag := true.
realObject := self xxxFetch.
oldFlag == true
ifTrue: [response := (PopUpMenu labels: 'proceed normally\debug' withCRs)
startUpWithCaption: 'Object being fetched for a second time. Should not happen,
and needs to be fixed later.'.
response = 2 ifTrue: [self halt]].
^ realObject perform: aMessage selector withArguments: aMessage arguments
```

L'opération essentielle se passe à la ligne 5. La méthode `xxxFetch` reconstruit l'objet depuis le disque, et effectue les permutations de pointeurs adéquates (par l'intermédiaire de la méthode `xxxFix`). Le message d'origine (`aMessage`, en argument de `doesNotUnderstand:`) est alors exécuté par l'objet reconstruit (ligne 10).

Cette technique peut être utilisée pour créer des objets espions qui analysent le fonctionnement d'autres objets sans le perturber. Ce type de système est souvent appelé système épiphyte[1]. Un framework complet nommé Epitalk a d'ailleurs été proposé sur ce thème par François Pachet (http://www-poleia.lip6.fr/~fdp/EpiTalk-papers.html), et est très utile pour tracer le fonctionnement d'un système complexe, faire de l'analyse de performances, d'architectures…

L'embryon d'un tel framework existe d'ailleurs en Squeak avec les classes `ObjectTracer` et `ObjectViewer`. `ObjectTracer` ouvre un inspecteur sur chaque message reçu par un objet, tout en permettant à ce message de s'exécuter normalement. Il s'agit d'un wrapper, qui ne remplace pas l'objet encapsulé (encapsulation effectuée par le message `on: objetAEncapsuler` envoyé à la classe `ObjectTracer`), mais lui retransmet les messages (dernière ligne) :

```
ObjectTracer>>doesNotUnderstand: aMessage
```

[1] **Épiphyte :** adjectif et nom masculin (gr. *epi*, sur, et *phuton*, plante). Se dit d'un végétal (telles certaines orchidées équatoriales) qui vit fixé sur des plantes, mais sans les parasiter.

```
"Ceci est la définition sur ObjectTracer"
Debugger
   openContext: thisContext
   label: 'About to perform: ' , aMessage selector
   contents: thisContext shortStack.
^ aMessage sentTo: tracedObject
```

`ObjectViewer`, au contraire, vient remplacer l'objet qu'il encapsule, grâce au `become:`. Nous allons illustrer par un exemple la puissance de ce concept.

Considérons une application de plusieurs centaines de milliers de lignes, dans laquelle un tableau de trois éléments est créé, où deux éléments sont insérés (`t := Array new: 3. t at: 1 put: 4. t at: 2 put: 5.`). Nous souhaiterions être informé de l'ajout du troisième élément dans le tableau, afin d'effectuer certaines opérations, ou simplement tracer les utilisations du tableau, mais nous ne savons pas où cet ajout va avoir lieu dans les centaines de milliers de lignes de code de l'application (cas – hélas – extrêmement courant dans une application réelle). La méthode `evaluate: unBloc wheneverChangeIn: unAutreBloc`, définie sur `Object`, nous permet d'effectuer cette surveillance : après l'exécution de `t evaluate: [Transcript cr; show: (t at: 3)] wheneverChangeIn: [t at: 3]`, toute modification de la valeur de la troisième cellule de t (par exemple, `t at: 3 put: 7`) provoquera l'évaluation du premier bloc, qui effectuera l'affichage de la nouvelle valeur dans le `Transcript`.

Ce mécanisme d'apparence quelque peu magique fonctionne en fait assez simplement : la méthode `evaluate:wheneverChangeIn:` est définie de la façon suivante (notez l'utilisation du `become:` en ligne 5) :

```
Object>>evaluate: actionBlock wheneverChangeIn: aspectBlock
  | viewerThenObject objectThenViewer |
  objectThenViewer := self.
  ViewerThenObject := ObjectViewer on: objectThenViewer.
  objectThenViewer become: viewerThenObject.
  objectThenViewer
     xxxViewedObject: viewerThenObject
     evaluate: actionBlock
     wheneverChangeIn: aspectBlock
```

Lorsque le tableau `t`, devenu un `ObjectViewer`, reçoit le message `at: 3 put: 7`, il ne le comprend pas, et la méthode `doesNotUnderstand:` est déclenchée.

```
ObjectViewer>>doesNotUnderstand: aMessage
  | returnValue newValue |
  aMessage lookupClass: tracedObject class.
  recursionFlag
     ifTrue: [^ aMessage sentTo: tracedObject].
  recursionFlag := true.
  returnValue := aMessage sentTo: tracedObject.
  newValue := valueBlock value.
  newValue = lastValue
     ifFalse: [changeBlock value.
        lastValue := newValue].
recursionFlag := false.
^ returnValue
```

Le message à exécuter est transmis à l'objet encapsulé (lignes 5-7). Le bloc de test de changement est évalué (ligne 8). S'il y a un changement, le bloc de changement est évalué à son tour (ligne 10).

Bogue !

Pour tester cet exemple, il a été nécessaire de corriger ce qui semble être un bogue dans la livraison Squeak utilisée. La ligne 3 a dû être ajoutée. En effet, la méthode `sentTo:` teste la présence de la `lookupClass` (la classe à partir de laquelle va commencer la recherche de la méthode à utiliser) du message (qui se trouve ici être `ObjectViewer`) dans la hiérarchie du receveur avant d'autoriser l'exécution du message. Il est donc nécessaire de réaffecter la classe de l'objet encapsulé dans la variable `lookupClass`.

Pour annuler l'espionnage d'un objet, il suffit de lui envoyer le message `xxxUnTrace`.

Proxies

Un proxy est un objet qui représente un objet complexe auquel il délègue les messages le cas échéant. Son implémentation utilise les deux techniques présentées (changement de référence et récupération d'erreur). Dans le cadre de son implantation en Squeak, l'idée est que le proxy est un objet minimal qui génère des erreurs qui sont capturées.

Lors de la capture d'une erreur, l'objet que le proxy représente va être créé, puis substitué *via* `become:` au proxy. Ainsi toutes les références au proxy sont-elles mises à jour seulement lorsqu'une des fonctionnalités du proxy est devenue nécessaire. On évite ainsi de continuellement déléguer les messages.

Supposons par exemple que l'on dispose d'une part d'une classe `Image` et d'autre part d'une classe `ImageProxy` incluant certaines informations nécessaires pour créer une image (un chemin de fichier par exemple).

Dans ce cas, la méthode `doesNotUnderstand:` ressemblerait à la méthode suivante :

```
ImageProxy>>doesNotUnderstand: aMessage
  |image|
  image := Image from: self path.
  self become: image.
  ^self perform: aMessage selector
       withArguments: aMessage arguments
```

En résumé

Dans ce chapitre, nous avons présenté les applications de la réflexivité (introspection et intercession) en Squeak. Cette technique puissante peut être utilisée pour analyser la structure interne de tout objet, analyser son fonctionnement par espionnage sans le perturber, mais aussi analyser automatiquement le code d'une application pour y détecter des bogues possibles, ou encore récupérer et traiter les erreurs à l'exécution.

9

Les processus

Ainsi que nous avons pu le mentionner, un système Squeak comprend une machine virtuelle, qui met en œuvre un processeur virtuel ; ce processus exécute les instructions (exprimées en *bytecode*) qui correspondent au code des méthodes Squeak.

Dans les mécanismes dont nous avons traités jusqu'ici, il n'existait qu'un seul flux de programme, s'exécutant en mémoire. Une nouvelle notion va permettre d'introduire un contrôle spécifique sur le flux des exécutions au sein du système : une portion de code peut être interrompue (par le message `suspend`) et reprise plus tard (`resume`). Si plusieurs portions de code sont en attente en mémoire à un instant donné, le choix quant à celle qu'il faut reprendre est effectué par un objet gestionnaire, principalement sur la base de priorités.

Ces portions de code s'appellent des processus (instances de la classe `Process`, appartenant à la catégorie `Kernel-Processes`) et permettent de simuler le fonctionnement simultané de plusieurs programmes en mémoire.

Qu'est ce qu'un processus ?

Les processus sont utilisés en particulier dans les applications multi-utilisateurs (serveurs de bases de données, serveurs Web…) pour traiter « simultanément » plusieurs demandes ; dans le domaine de la simulation, ce sera pour étudier le fonctionnement d'un système concurrent (chaîne de production, systèmes multi-agents…), tandis que dans tout système d'exploitation ils seront mis à profit pour permettre l'utilisation simultanée de plusieurs applications par l'utilisateur, ainsi que la gestion des différents périphériques.

Dans la mesure où il n'existe qu'un seul processeur matériel (du moins sur les machines mono-processeur), la simultanéité ne sera pas réelle et il s'agira nécessairement de temps partagé : chaque processus dispose du processeur pendant un certaine période, déterminée par l'attribution d'un quantum de temps prédéfini pour chaque processus, ou par l'exécution d'une instruction redonnant explicitement le contrôle au gestionnaire de processus ; ce dernier, implémenté par la

classe `ProcessorScheduler` et son unique instance `Processor` peut alors donner la main à un autre processus(voir figure 9-1). Cette seconde solution est utilisée en standard dans Squeak, bien que la première puisse être mise en œuvre sans trop de difficultés.

Remarque

Il suffit de créer un processus de très haut niveau, dont le fonctionnement est contrôlé par l'horloge avec un délai d'attente fixé (*time slice,* qui donne son nom à ce mécanisme de *time slicing*). Puisque ce processus est de très haute priorité, il aura nécessairement la main lorsque son délai d'attente viendra à expiration. Il lui suffit alors d'envoyer un message qui contraigne le processus « courant » (celui qui était actif avant lui) à rendre la main (message `yield`) pour permettre aux autres processus de s'exécuter.

Techniquement parlant, les processus Squeak sont des processus légers (threads), préemptifs d'un niveau de priorité à l'autre (un processus de niveau supérieur peut prendre la main « de force » sur un processus de niveau inférieur), et coopératifs entre processus de niveau de priorité identiques (un processus doit rendre la main volontairement pour que les autres aient une chance de s'exécuter).

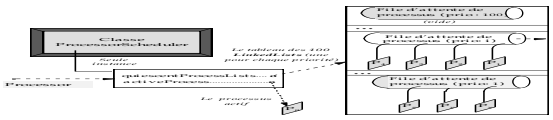

Figure 9-1. Le gestionnaire de processus (seule instance de ProcessorScheduler) et ses listes de processus en attente (P0 est de priorité i, i étant supposée être ici la priorité la plus élevée correspondant à une liste non vide)

Un processus est une instance de `Process`, sous-classe de `Link` (abordée au chapitre 6 traitant des collections). Nous verrons en effet que la gestion des processus implique qu'ils puissent être stockés (alternativement) dans différents types de listes chaînées, et que ces listes ont pour éléments des `Link`.

Un processus est pour l'essentiel caractérisé par ses variables d'instance :

- `priority` : une valeur entière utilisée pour ordonner les processus par niveaux de priorité ;
- `suspendedContext` : le contexte actif au moment de la suspension, c'est-à-dire un ensemble d'informations pertinentes au moment de l'exécution du processus, relatives au contexte qui est à l'origine du contexte courant, du receveur de ce contexte, des arguments et informations temporaires sur la pile de travail… ;
- `myList` : la liste de processus où se trouve le processus suspendu (il peut y avoir plusieurs listes de ce genre, et ce en liaison également avec la notion de sémaphore…).

Remarque terminologique

Le terme de *processus* est employé en Squeak en raison du nom de la classe `Process`. En toute rigueur, il ne s'agit pas de processus, mais de threads, ou processus légers. Un processus, au sens par exemple des processus Unix (créés par `fork()`), s'exécute en effet dans un segment de données totalement autonome. Il ne peut donc communiquer avec les autres processus que par des « tubes » (pipes), ou par des segments de mémoire partagés qui doivent être explicitement définis. Au contraire, les threads créés à partir d'un même processus partagent l'espace de données dont ils sont issus, et peuvent donc communiquer par son intermédiaire. C'est le cas en Squeak, où les processus, qui sont donc des threads, peuvent accéder aux objets de l'espace de nommage à partir duquel ils ont été créés (variables globales, classes…).

Création de processus

On crée un processus en envoyant un message à un bloc (une instance de la classe `BlockContext`, par exemple `[2+4] fork`). Plusieurs messages de création sont définis :

- `newProcess` crée un processus dont le code est celui du bloc receveur, mais qu'il n'insère dans aucune file des processus en attente. La priorité qui est affectée au nouveau processus est celle du processus actif ;
- `newProcessWith:` permet de créer un nouveau processus en transmettant des arguments au bloc receveur ; cela permet de définir le processus de façon générale, sans référence aux arguments qui sont définis à l'extérieur de ce bloc (plus précisément, il est ainsi possible de *ne pas* considérer sous forme d'un objet le contexte extérieur au bloc, ce qui améliore les performances) ;
- `fork` permet de créer un processus (avec `newProcess`) et de le « planifier » (*schedule*). Un message `resume` lui est envoyé,ce qui le positionne dans la file d'attente correspondant à sa priorité et le rend susceptible d'être activé ;
- `forkAt:` effectue la même opération que `fork`, mais permet de préciser le niveau de priorité à affecter au processus créé.

Priorités des processus

À un moment donné, plusieurs processus peuvent donc se trouver en mémoire, l'un d'eux seulement étant actif. Le contrôle de l'exécution des processus est la tâche de l'unique instance de la classe `ProcessorScheduler`, stockée dans une variable globale du système, la variable `Processor`. Pour déterminer la séquence d'activation des processus, et en permettre une gestion fine, une priorité est affectée à chacun d'eux, qui peut varier de 1 à 10.

Certains de ces niveaux de priorité ont été dénommés et correspondent à des situations répertoriées (`BackgroundProcess`, `HighIOPriority`, `LowIOPriority`, `SystemBackgroundPriority`, `SystemRockBottomPriority`, `TimingPriority`, `UserBackgroundPriority`, `UserInterruptPriority`, `UserSchedulingPriority`).

Tableau 9-1. Rôles des différentes priorités des processus et méthodes d'accès

Valeur de la priorité	Méthode	But
8	`timingPriority`	Processus devant fonctionner en temps réel
7	`highIOPriority`	Entrées-sorties critiques (par exemple réseau)
6	`lowIOPriority`	Entrées-sorties normales (par exemple clavier)
5	`userInterruptPriority`	Interaction utilisateur de haut niveau de priorité
4	`userSchedulingPriority`	Interaction utilisateur normale
3	`userBackgroundPriority`	Processus utilisateurs en tâche de fond

2	`systemBackgroundPriority`	Processus système en tâche de fond
1	`systemRockBottomPriority`	Processus système de plus bas niveau de priorité

On obtient la valeur numérique de la priorité correspondant à un nom en envoyant à l'instance `Processor` le message indiqué dans la deuxième colonne du tableau 9-1. Par exemple, `Processor lowIOPriority` renvoie `6`.

Les différents processus qui doivent s'exécuter (par exemple, ceux qui ont reçu un message `resume`, mais qui n'ont pas une priorité suffisante pour être *le* processus actif) sont stockés dans des listes, correspondant à leur priorité, où ils attendent que le gestionnaire les active réellement. Ils sont alors qualifiés de processus activables.

Le gestionnaire (`Processor`) donne toujours le contrôle au processus en attente qui a le plus haut niveau de priorité. Lorsque plusieurs processus de même niveau de priorité sont en attente, pour que les autres puissent s'exécuter, le processus actif doit redonner la main en exécutant `Processor yield`, ce qui provoque le renvoi du processus actif à la fin de la file des processus en attente qui ont le même niveau de priorité.

Plusieurs méthodes d'instance sont définies sur les processus pour gérer les changements d'état et de priorité :

- `priority` et `priority:` permettent d'obtenir ou d'affecter la priorité du processus receveur ;
- `suspend` arrête temporairement le processus receveur, en préservant son contexte ;
- `resume` permet de rendre à nouveau `activable` un processus qui avait été suspendu ;
- `terminate` arrête définitivement le processus receveur.

D'autres états (notamment l'état d'attente) sont dus à l'utilisation d'un sémaphore, notion que nous allons approfondir à présent.

Les sémaphores

Les sémaphores sont des mécanismes de synchronisation entre processus qui permettent de protéger des données ou des ressources utilisées par plusieurs processus. Ils permettent, en particulier, de mettre en attente un processus qui demande une certaine ressource, tant que cette dernière n'est pas disponible, et ce, sans que le processus doive vérifier périodiquement la disponibilité de la ressource demandée.

Un sémaphore peut être créé en envoyant le message `new` à la classe `Semaphore` (d'autres méthodes de création sont possibles, par exemple `forMutualExclusion`, qui crée un processus avec un signal d'avance). Lorsqu'un processus souhaite utiliser la ressource protégée par le sémaphore, il lui envoie un message ; les méthodes de la ressource doivent donc être conçues de telle façon que cela provoque l'envoi du message `wait` au sémaphore pertinent (un exemple de ce mécanisme est fourni avec les `SharedQueue`, abordées au chapitre 6 consacré aux collections). Si le nombre de processus connectés à ce moment au sémaphore est inférieur au nombre de processus qui sont autorisés à se connecter, ce message n'est pas bloquant et le processus peut continuer ses opérations en utilisant la ressource comme il le souhaitait. Dans le cas contraire, le processus est

stocké en fin de la file d'attente du sémaphore, et ne sera relancé que lorsque les processus qui le précèdent auront libéré la place.

Un processus signifie à un sémaphore qu'il libère la ressource en lui envoyant le message `signal`. Le sémaphore envoie alors un message `resume` au premier processus en attente de sa file d'attente, ce qui rend ce dernier activable, c'est-à-dire qu'il est ajouté dans la file d'attente du `ProcessorScheduler` correspondant à sa priorité (voir figure 9-2 ci-après).

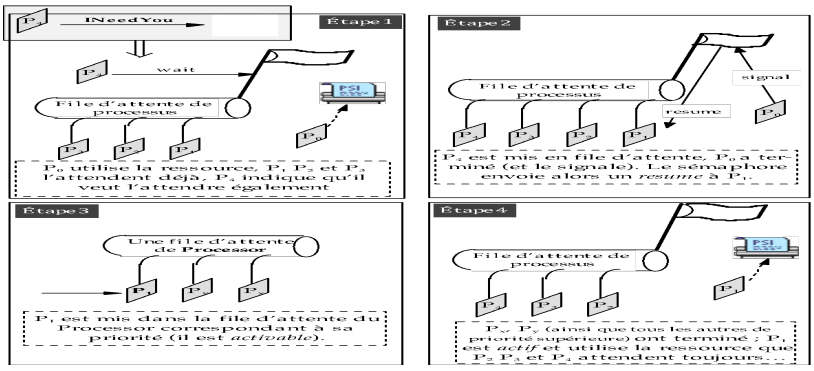

Figure 9-2. Fonctionnement d'un sémaphore gérant l'accès à une ressource

Différents états d'un processus

Par souci de synthèse, nous allons préciser les différents états dans lesquels un processus peut se trouver à un instant donné : suspendu, activable, actif ou en attente (voir figure 9-3 ci-après).

- *actif* : il est en cours d'exécution ;
- *activable* : il est dans une des files d'attente de `Processor` ;
- *en attente* : il est suspendu sur un sémaphore (il se trouve dans la file d'attente du sémaphore ; il n'est donc pas encore activable) ;
- *suspendu* : si c'est le processus actif, il est interrompu et pourra être réactivé par la suite ; sinon, il est ôté de la file d'attente des processus activables où il se trouve.

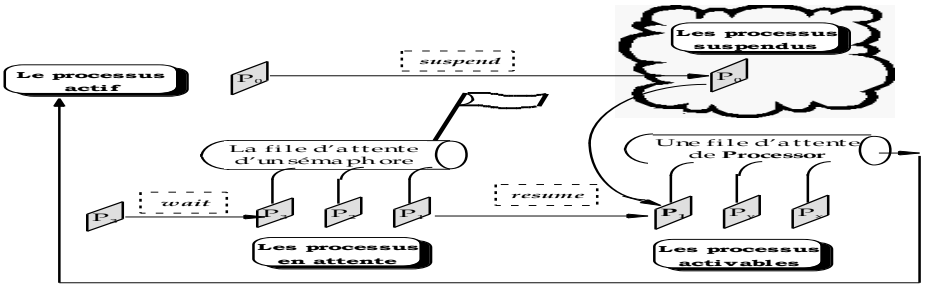

Figure 9-3. Différents états des processus et les messages qui font passer de l'un à l'autre

Les mécanismes de temporisation : la classe `Delay`

Les instances de la classe `Delay` sont utilisées pour mettre en attente un processus durant un certain temps.

Un `Delay` (sous-classe d'`Object`) peut être créé en spécifiant une durée (`forMilliseconds:` ou `forSeconds:`). Il est principalement constitué d'un sémaphore (stocké dans la variable d'instance `delaySemaphore`) sur lequel le processus actif est mis en attente lorsque l'instance de `Delay` ainsi créée reçoit le message `wait`. Ce message consiste pour l'essentiel en l'envoi de `wait` au sémaphore `delaySemaphore`, et en l'envoi du message `resume` au même sémaphore lorsque le délai spécifié est venu à expiration.

Dans l'exemple suivant, nous utilisons les processus et les délais pour afficher une horloge qui signale l'heure dans le `Transcript` toutes les trois secondes.

Nous créons pour cela une classe `Horloge`. Chaque horloge a un processus (`process`), qui a pour seule fonction de notifier un changement toutes les trois secondes, et une variable booléenne (`stop`) utilisée pour arrêter le processus.

```
Object subclass: #Horloge
      instanceVariableNames: ' process stop '
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Exemples'
```

Nous définissons les trois méthodes d'instance suivantes :

```
start
      process resume.
```

```
stop
      stop := true.
      Transcript cr; show: 'l''horloge est arrêtée'
```

```
initialize
      process := [| delay |
            delay := Delay forSeconds: 3.
            [stop]
               whileFalse: [
                  delay wait.
                  Transcript cr; show: Time now printString]] newProcess.
      process priority: Processor timingPriority.
      stop := false
```

La boucle sans fin du processus de l'horloge (ligne 5 de la méthode `initialize`) est temporisée par l'envoi du message `wait` (ligne 6) selon le délai qui a été créé en ligne 3, avec une durée d'attente de trois secondes.

Exemples d'utilisation des processus

Les quelques exemples suivants illustrent différents aspects des processus.

Le message `fork` envoyé aux blocs présentés ci-après crée deux processus P1 et P2 qui effectuent chacun dix affichages dans le `Transcript` (respectivement, les nombres de 1 à 10 pour le premier, de 11 à 20 pour le second). Ces deux processus sont créés avec la même priorité.

```
[1   to: 10 do: [:i| Transcript show:' '; show:  i printString. ]] fork. "(P1)"
[11 to: 20 do: [:i| Transcript show:' '; show:  i printString. ]] fork.  "(P2)"
```

L'affichage dans le `Transcript` (`1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20`) montre alors qu'aucun des deux ne redonne le contrôle au processeur durant sa boucle :le premier s'exécute jusqu'à sa fin, puis le second fait de même. Cela prouve, en outre, qu'aucun quantum de temps n'est attribué par défaut aux processus.

Redonner la main : le message `yield`

Pour que les exécutions des processus puissent alterner, nous devons introduire `Processor yield` dans chacune des boucles, ce qui entraîne l'interruption du processus actif et permet à d'autres processus de même priorité de s'exécuter (à noter l'utilisation de `newProcess` puis de `resume`, autre manière de lancer des processus) :

```
      p1 :=  [1   to: 10 do: [:i| Transcript show:' '; show:  i printString. Processor
yield]. ] newProcess.
      p2 :=  [11 to: 20 do: [:j| Transcript show:' '; show:  j printString.  Processor
yield]. ] newProcess .
      p2 resume. p1 resume
```

Nous obtenons alors l'affichage `11 1 12 2 13 3 14 4 15 5 16 6 17 7 18 8 19 9 20 10`, qui montre que les processus se sont bien déroulés de façon « quasi parallèle ».

La méthode `yield` est très élégamment implémentée de la façon suivante :

```
yield
  | semaphore |
  semaphore := Semaphore new.
  [semaphore signal] fork.
  semaphore wait
```

Dans le cas où seraient présents des processus activables de même priorité que le processus actif, un sémaphore nouveau est créé (ligne 3), ainsi qu'un nouveau processus (ligne 4 – nous l'appellerons Px_2, voir les étapes de la figure 9-4) dont le rôle consiste à envoyer le message `signal` de libération de la ressource au nouveau sémaphore ; cela signifie que le processus est créé mais non pas qu'il envoie ce message : ligne 5, P1 est mis en attente et redeviendra activable quand Px_1 sera rendu actif, et donc quand tous les processus avant Px_1 auront été terminés ou (re)mis en attente. Le fonctionnement de l'ensemble est synthétisé dans les différentes étapes de la figure 9-4.

Figure 9-4. Les diverses séquences produites par l'instruction `yield` lors d'un affichage de nombres par deux processus

Gestion des priorités

Remarque importante

Les processus qui ont une priorité supérieure à 5 ont priorité sur les interactions de l'utilisateur. Ces processus dominent donc les actions de ce dernier, ce qui peut rendre impossible l'utilisation des menus ou du clavier. Il faut noter également que les processus de priorité supérieure ou égale à 7 prennent le pas sur les processus de gestion des fenêtres, et doivent donc être de durée limitée, sauf à geler l'interface le temps de leur exécution.

Dans l'exemple suivant, nous créons un processus de priorité basse (`3`), qui affiche les nombres de `1` à `50` dans le `Transcript`. Pendant le déroulement de ce processus, l'utilisateur exécute de temps en temps l'expression `Transcript show: 'coucou'`, qui est affichée immédiatement. Cela montre que l'utilisateur peut continuer à interagir avec le système pendant le déroulement du premier processus :

```
 [1 to: 50 do: [:i| Transcript show:' '; show:  i printString. ]] forkAt: (Processor
userBackgroundPriority).
```

Nous obtenons alors l'affichage suivant :

```
 1 2 3 4 5 6 7 8 9 10 11 12 13  coucou 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 coucou 32 33 34 35 36 37 38 39 40 41 42 43 44 45 coucou 46 47 48 49 50
```

Avertissement

Attention, des problèmes peuvent survenir lors de l'utilisation simultanée du `Transcript` par plusieurs processus qui effectuent des affichages. En effet, les méthodes d'affichage ne sont pas protégées contre les accès concurrents, et peuvent être interrompues à des moments inopportuns. Il peut alors être nécessaire de définir un `Transcript` particulier (sous-classe de `Transcript`), protégé contre les accès concurrents. Ce problème se pose d'ailleurs pour toute ressource partageable, et nécessite l'utilisation de sémaphores d'exclusion mutuelle et de sections critiques.

Ainsi que nous l'avons souligné plus haut, un processus activable d'un niveau de priorité donné est toujours activé avant un autre processus activable de niveau de priorité inférieur, comme le montre l'exemple suivant :

```
[10 timesRepeat: [Transcript show:' '; show: '3'. Processor yield]] forkAt: 2.
[10 timesRepeat: [ Transcript show:' '; show: '2'. Processor yield]] forkAt: 3.
[10 timesRepeat: [ Transcript show:' '; show: '1'. Processor yield]] forkAt: 4.
```

Le troisième processus, de priorité 4, est exécuté avant le deuxième (priorité 3), lui-même exécuté avant le premier (priorité 2), comme le montre l'affichage suivant :

```
 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
```

Dans bien des situations, l'usage de simples priorités ne suffit pas : les processus doivent être explicitement coordonnés. Dans l'exemple suivant, nous utilisons un sémaphore pour synchroniser l'exécution de deux processus : nos contraintes sont que `p1` doit être le premier à commencer son exécution, mais qu'il doit ensuite attendre que `p2` ait effectué une action donnée pour pouvoir poursuivre.

Il faut veiller à ce que la priorité de `p1` soit supérieure à celle de `p2` pour que le premier se mette en attente (`wait`) sur le sémaphore que le second débloquera au moment voulu (`signal`).

```
|s p1 p2|
s := Semaphore new.
p1 := [
```

```
    Transcript cr; show: 'P1 : je suis le premier'; cr.
    s wait.
    Transcript cr; show: 'P1 : je suis débloqué'; cr.] newProcess priority: 4.
p2 := [
    Transcript show: 'P2 : j''envoie signal au sémaphore S'; cr.
     s signal.
    Transcript  show: 'P2 : j''ai envoyé signal à S'.] newProcess priority: 3.
p1 resume. p2 resume.
```

Nous obtenons alors les affichages suivants, qui montrent que `p2` s'interrompt bien pour laisser `p1` s'exécuter : `p1`, de priorité supérieure à `p2`, a été activé dès que ce dernier a envoyé le message `signal` au sémaphore, `p2` recommençant son exécution dès que `p1` a fini la sienne.

```
P1 : je suis le premier
P2 : j'envoie signal au sémaphore S

P1 : je suis débloqué
P2 : j'ai envoyé signal à S
```

L'importance des priorités transparaît dans le code suivant (où les priorités ont été échangées), et qui ne fonctionne pas de la façon voulue, car `p2` conserve le contrôle jusqu'à la fin de son exécution, `p1` n'étant activé qu'à ce moment-là, comme le montrent les affichages.

```
|s p1 p2|
s := Semaphore new.
p1 :=
      [s wait.
     Transcript cr; show: 'P1 : je suis débloqué'; cr.] newProcess priority: 42.
p2 :=
     [Transcript show: 'P2 : j''envoie signal au sémaphore S'; cr.
      s signal.
     Transcript  show: 'P2 : j''ai envoyé signal au sémaphore S'.] newProcess priority:
43.
p1 resume.
p2 resume.
```

Voici l'affichage qui en résulte :

```
P2 : j'envoie signal au sémaphore S
P2 : j'ai envoyé signal au sémaphore S
P1 : je suis débloqué
```

Un grand classique de la synchronisation de processus : le dîner des philosophes

Le dîner des philosophes est un excellent modèle pour un grand nombre de problèmes réels.

Il s'agit d'un problème de synchronisation inter-processus, qui permet de mettre en évidence deux problèmes fondamentaux : la coordination des processus entre eux, et les problèmes d'inter-blocages (*dead-lock* en anglais), lorsque deux processus se bloquent mutuellement, chacun attendant que l'autre libère une ressource pour pouvoir redémarrer.

Présentation du problème

Le problème du dîner est le suivant : cinq philosophes sont réunis autour d'une table ronde pour dîner et se détendre tout en philosophant. Le repas est constitué de spaghettis. Pour qu'un philosophe puisse manger, il doit utiliser deux couverts (afin d'enrouler ses spaghettis selon les

règles de bienséance en vigueur dans les cercles de philosophie). La difficulté est qu'il n'y a que cinq couverts sur la table (voir figure 9-5 ci-après) :

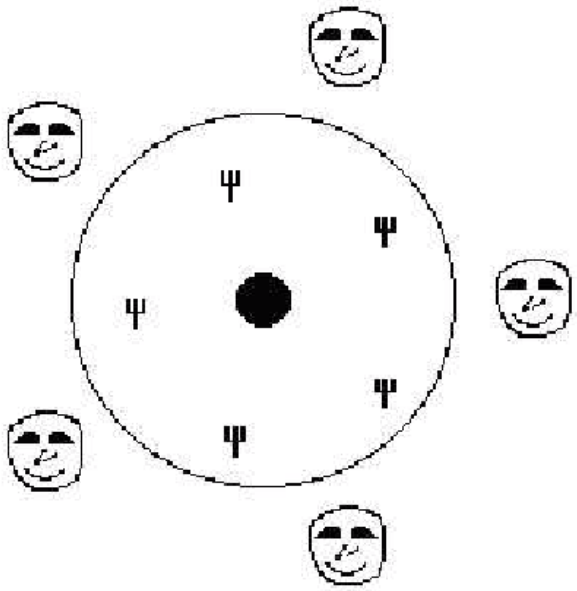

Figure 9-5. La situation à la table des philosophes

Bien entendu, dans un cercle de philosophie, on ne mange pas avec les mains, on n'arrache pas les couverts des mains de son voisin et on n'utilise que les couverts à proximité (celui de gauche et celui de droite). Toute la difficulté du problème (et son intérêt) réside donc dans le choix d'une méthode qui permette de synchroniser les actions des philosophes.

Un philosophe peut être dans l'un des trois états suivants :

- il pense, et dans ce cas il a la politesse de lâcher ses couverts ;
- il est affamé, ce qui l'empêche de penser et le pousse à essayer de saisir les deux couverts qui sont à sa proximité pour pouvoir manger ;
- il mange, et utilise dans ce cas deux couverts.

Un philosophe reste un homme ; il ne peut donc penser pendant un temps indéfini et doit finir par manger. Toutefois, même affamé, un philosophe est un homme parfaitement éduqué et ne mange pas tout son plat d'un seul coup. Nous modéliserons ces deux aspects en considérant d'une part qu'un philosophe reste dans un des deux états *pensant* ou *mangeant* pendant une durée aléatoire et, d'autre part, en imposant un nombre fixe de fois où il doit manger.

Modélisation

Il nous faudra donc recourir à trois classes pour modéliser cette situation : une classe `Philosophe` dont les instances seront les philosophes que nous avons évoqués, une classe `Fourchette` dont les instances seront les fourchettes qu'ils utilisent (en nombre égal au nombre de philosophes) et une classe `TableDePhilosophes`, qui contiendra l'ensemble des éléments (fourchettes et philosophes) impliqués dans le traitement.

Nous définirons `TableDePhilosophes` comme une sous-classe de `Object` (les trois variables d'instance – `nombreDeDineurs`, `fourchettes`, `philosophes` – parlent d'elles-mêmes). La définition de la classe `Philosophe` comprend les variables `nom` (l'identifiant du philosophe), `etat` indique s'il pense, mange ou est affamé, tandis que `fourchetteGauche` et `fourchetteDroite` indiquent les numéros des fourchettes qui se trouvent respectivement à la gauche et à la droite de chaque philosophe.

```
Object subclass: #TableDePhilosophes
     instanceVariableNames: 'nombreDeDineurs fourchettes philosophes '
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Philosophes'
```

```
Object subclass: #Philosophe
     instanceVariableNames: 'fourchetteGauche fourchetteDroite nom etat '
     classVariableNames: ''
     poolDictionaries: ''
     category: Philosophes
```

Les couverts (tous identiques et représentés ici par des fourchettes) sont des ressources rares sur lesquelles les philosophes font des accès concurrents. Ils peuvent donc être modélisés par des sémaphores d'exclusion mutuelle, garantissant ainsi l'usage d'une ressource (une fourchette) par un seul processus à la fois (un philosophe). La classe `Fourchette` est donc définie essentiellement avec la variable d'instance `semaphore`.

```
Object subclass: #Fourchette
     instanceVariableNames: 'semaphore '
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Philosophes'
```

Gestion des accès concurrents

L'initialisation d'une fourchette doit créer le sémaphore qui la protège contre les accès concurrents :

```
initialize
     semaphore := Semaphore forMutualExclusion
```

On accédera donc à une fourchette en envoyant le message `wait` à son sémaphore, et la fourchette sera libérée par l'envoi du message `signal` à son sémaphore. Ainsi les méthodes `wait` et `signal` définies sur `Fourchette` se contentent-elles de retransmettre ces messages au sémaphore :

```
signal
     semaphore signal.
wait
     semaphore wait.
```

La méthode d'initialisation d'une `TableDePhilosophes` crée les fourchettes et les philosophes (lignes 5 à 7) – le « nom » d'un philosophe est son numéro – et répartit les fourchettes aux philosophes (lignes 8 à 11, les fourchettes sont associées aux philosophes dans l'ordre de placement autour de la table) :

```
initialize: n
 self nombreDeDineurs: n.
 self fourchettes: (Array new: n).
 self philosophes: (Array new: n).
 1 to: n do: [:p |
   self fourchettes at: p put: (Fourchette new).
   self philosophes at: p put: (Philosophe new: p)].
 1 to: n do: [:p |
   (self philosophes at: p)
    fourchetteGauche: (self fourchettes at: p)
 fourchetteDroite: (self fourchettes at: ((p\\n) + 1))].
 "(p\\n)+1 est egal à p + 1, sauf pour p=n où il vaut alors 1"
```

La méthode de lancement du dîner (méthode d'instance de `TablesDePhilosophes`) est alors définie de cette façon :

```
dine: rep
   1 to: self nombreDeDineurs do: [:p |
     (self philosophes at: p) philosopheCycle: rep].
```

Dans cette dernière méthode, `rep` représente le nombre de cycles (penser, récupérer des couverts, manger, relâcher ses couverts...) que doit effectuer un philosophe.

Cycle de fonctionnement d'un philosophe

Puisque les comportements des philosophes sont autonomes, nous pouvons les modéliser par des processus concurrents, chaque processus étant une suite de changements d'états. Le code présenté ci-après définit un processus par l'envoi du message `fork` à un bloc qui fait passer un philosophe par les actions successives *penser*, *récupérer deux fourchettes*, *manger*, *relâcher ses fourchettes* :

```
philosopheCycle: rep
     [rep timesRepeat: [
        self pense.
        self recupereFourchettes.
        self mange.
        self relacheFourchettes].
     self etat: #philosopheEndormi.
     ] fork
```

Pour introduire un peu de variabilité dans les comportements des philosophes, nous introduisons une durée aléatoire (classe `Random`) dans les méthodes `mange` et `pense` :

```
mange
     self etat: #philosopheMangeant.
     (Random new  next * 20) rounded  timesRepeat: [Processor yield]
```

```
pense
     self etat: #philosophePensant.
     (Random new  next * 20) rounded timesRepeat: [Processor yield]
```

L'utilisation de `Processor yield` permet de laisser la main aux autres processus lorsqu'un processus (un philosophe) est en train de manger ou de penser. Pour passer de l'état « pensant » à l'état « mangeant », un philosophe doit saisir deux couverts, ce qui correspond techniquement à la réservation sur le sémaphore de ces ressources rares (`fourchetteGauche wait ; fourchetteDroite wait`), et les relâcher (`fourchetteGauche signal ; fourchetteDroite signal`) dès qu'il s'arrête de manger.

Gestion des problèmes d'inter-blocage

On résout bien ainsi le problème de la *synchronisation*, mais pas celui de l'*inter-blocage*. Supposons en effet que tous les philosophes soient affamés en même temps, et saisissent tous, par exemple, la fourchette qui est à leur droite. Aucun ne pourra accéder à la fourchette du côté opposé, puisque c'est son voisin qui la tient. Il attendra donc que son voisin s'arrête de manger et qu'il repose son couvert. Le problème est que ce voisin est lui-même bloqué en attente de son autre voisin, lui-même bloqué… À moins qu'un des philosophes ne se rende compte du problème et ne se sacrifie en reposant sa fourchette (ce qui ne peut pas se produire car chacun sait qu'un philosophe affamé n'est pas capable de penser), la situation est irrémédiablement bloquée.

Parmi toutes les solutions qui ont été proposées pour résoudre ces problèmes d'inter-blocage, il en est une qui s'applique facilement au cas des philosophes : il suffit de créer un comportement asymétrique chez les philosophes. Un philosophe assis à une place paire saisira toujours en premier la fourchette qui est à sa droite, tandis qu'un philosophe assis à une place impaire saisira celle de gauche (ligne 3). Nous pouvons ainsi écrire les méthodes `recupereFourchettes` et `relacheFourchettes` de la façon suivante :

```
recupereFourchettes
    self etat: #philosopheAffame.
    nom \\ 2 == 0
      ifTrue:
         [self fourchetteGauche wait.
         self fourchetteDroite wait.]
      ifFalse:
         [self fourchetteDroite wait.
         self fourchetteGauche wait.]
```

```
relacheFourchettes
     self etat: #philosophePosant.
     self fourchetteDroite signal.
     self fourchetteGauche signal.
```

Les méthodes que nous venons de définir permettent (si on y ajoute les méthodes d'accès et d'affectation habituelles, qui n'ont pas été reproduites ici) d'effectuer une simulation d'un dîner de philosophes.

Exemples de fonctionnement

Si nous souhaitons avoir une idée de ce qui se passe effectivement, nous pouvons modifier la méthode `philosopheCycle:` pour demander des affichages dans le `Transcript` :

```
philosopheCycle: rep
     [rep timesRepeat: [
        self pense.
```

```
        Transcript cr; show: 'Philosophe ', self nom printString,' est dans l''état
#philosophePensant '.
        self recupereFourchettes.
        self mange.
        Transcript cr; show: 'Philosophe ', self nom printString,' est dans l''état
#philosopheMangeant '.
        self relacheFourchettes
        Transcript cr; show: 'Philosophe ', self nom printString,' est dans l''état #
philosophePosant '.].
      self etat: # philosophePensant.
      Transcript cr; show: 'Philosophe ', self nom printString,' est dans l''état #
philosophePensant '.
] fork
```

On peut ainsi obtenir un affichage qui permet de retracer l'évolution des événements :

```
Philosophe 1 est dans l'état #philosophePensant
Philosophe 2 est dans l'état #philosophePensant
Philosophe 3 est dans l'état #philosophePensant
Philosophe 4 est dans l'état #philosophePensant
Philosophe 5 est dans l'état #philosophePensant
Philosophe 3 est dans l'état #philosopheAffame
Philosophe 3 est dans l'état #philosopheMangeant
Philosophe 1 est dans l'état #philosopheAffame
Philosophe 1 est dans l'état #philosopheMangeant
Philosophe 3 est dans l'état #philosophePosant
Philosophe 3 est dans l'état #philosophePensant
Philosophe 3 est dans l'état #philosopheAffame
Philosophe 3 est dans l'état #philosopheMangeant
Philosophe 4 est dans l'état #philosopheAffame
Philosophe 2 est dans l'état #philosopheAffame
Philosophe 5 est dans l'état #philosopheAffame
Philosophe 1 est dans l'état #philosophePosant
Philosophe 1 est dans l'état #philosophePensant
Philosophe 5 est dans l'état #philosopheMangeant
Etc.
```

En résumé

Nous avons analysé en détail le fonctionnement des processus, mécanisme fondamental dans toute application un peu complexe. Gérés par un ordonnanceur à priorités, les processus de Squeak sont exécutés par un processeur virtuel opéré par la machine virtuelle, qui le rend le code indépendant de la plate-forme physique et donc totalement portable. Les processus peuvent être synchronisés par le temps (délais) ou par les données (sémaphores). Ces mécanismes ont été illustrés sur l'exemple classique du dîner des philosophes, situation ou les problèmes de synchronisation et les risques d'interblocages sont résolus par l'usage de sémaphores d'exclusion mutuelle et d'un algorithme de choix.