La syntaxe par l'exemple

Dr. Ducasse

ducasse@iam.unibe.ch
http://www.iam.unibe.ch/~ducasse/

Le mois dernier nous avons présenté la syntaxe minimaliste de Squeak.Ce mois ci nous allons définir quelques petits scripts afin de vous familiariser avec cette syntaxe qui peut surprendre. Nous discutons aussi du choix de conception qui anime les aspects syntaxiques des Smalltalk. Durant les prochains mois nous montrerons comment définir de nouvelles classes et méthodes, comment configurer un Wiki (un serveur de pages collaboratives), programmer une première Morph et les aspects réflexifs de Smalltalk (c-à-dire comment le langage peut utiliser la représentation qu'il a de lui-même pour construire des browseurs de code ou s'auto-modifier).

1. Quelques Scripts

Sons. Squeak permet la manipulation de sons. Voici un petit script très simple qui crée un son (déjà disponible dans Squeak) et dérive plusieurs notes puis les joue.

```
|note1 note2 note3|                          "des variables locales"
inst := AbstractSound soundNamed: 'oboe1'. "un message à mots-clès envoyé à une
classe"
  "des messages à mots-clès avec plusieurs arguments"
note1 := instr soundForPitch: #c4 dur: 0.5 loudness: 0.4.
               note2 := instr soundForPitch: #ef4 dur: 0.5 loudness: 0.4.
note3 := instr soundForPitch: #g4 dur: 0.5 loudness: 0.4.
(note1, note2, note3) play.          "des messages binaires et un message unaire"
```

Constatez qu'il n'y a pas de différence entre envoyer un message à une classe comme `soundNamed:` et à un objet comme `soundForPicth:dur:loudness:`. Les deux utilisent *strictement* les mêmes règles syntaxiques. En Smalltalk, il n'y a pas de notion de constructeurs. Ceci vient du fait que les classes sont aussi des objets, instances de classes spéciales dites métaclasses (un nom compliqué pour signifier que leurs instances sont des classes). En fait, une méthode de classes (envoyée à une classe) possède *exactement* la même sémantique qu'une méthode d'instance. Au niveau de l'implantation, il n'y a qu'un seul mécanisme. Une méthode de classe n'a pas à avoir le même nom que sa classe. Même au niveau de l'héritage les règles d'invocations des méthodes masquées (overriden) sont les mêmes. Il suffit d'utiliser la pseudo-variable super. Nous aborderons ce point lors de la définition des classes.

Capture d'écran retardée.Le mois dernier nous avons montré quelques manipulations graphiques. Nous vous proposons un petit script qui permet de capturer l'écran. Un des problèmes lorsque l'on fait une capture d'écran est que l'on ne veut pas que la capture s'effectue immédiatement mais après un certain délai. La solution à ce problème est de créé une expression dont l'exécution est retardée.Pour cela, nous créons une fermeture lexicale (*block closure*) que nous exécutons dans un processus indépendant. Ce processus est automatiquement ajouté à la liste des processus et pris en compte par l'ordonnanceur de taches de Squeak.

Voici donc un script qui crée un processus qui attend 5 secondes, émet un signal sonore pour avertir l'utilisateur, capture l'écran puis réduit l'écran ainsi capturé et montre le résultat sous forme d'une image. Attention ce script ne fonctionne que dans l'environnement Morphic de Squeak, il faut donc l'exécuter après avoir ouvert un projet morphique (**Menu principal Open... Morphic project**).

```
[| imageForm sketchMorph |  "debut de fermeture et déclaration de variables"
(Delay forSeconds: 5) wait. "message à mots clès puis unaire"
Smalltalk beep; beep.       "deux messages unaires envoyés au meme objet"
(Delay forSeconds: 1) wait.
imageForm := World imageForm.
sketchMorph := SketchMorph withForm: imageForm.
sketchMorph scalePoint: ((1/2) @ (1/2)).
sketchMorph openInWorld] fork.  "fin de fermeture et création d'un processus"
```

Examinons cela de plus près.

- La fermeture est créée à l'aide des [et] puis le message `fork` lui ait envoyé ce qui crée un nouveau processus de priorité standard.
- Deux variables locales sont déclarées. La variable `imageForm` représente l'instance de la classe `Form` qui sera créée. Une `Form` est un tableau de pixels utilisé pour représenter les images. Une `Form` possède une profondeur qui indique combien de bits sont utilisés pour coder une couleur. Les bits sont contenus dans une instance de `Bitmap` dont la structure interne dépend de la profondeur.
- Ensuite nous créons un délai de 5 secondes et le mettons en attente. Ainsi les autres processus vont être gérés sans que ce délai n'occupe du temps. Il s'agit d'un message à mots clès dont le résultat reçoit un message unaire.
- `Smalltalk` est une variable globale représentant une sorte d'espace de noms. Certaines fonctionnalités y sont aussi définies comme beep et d'autres qui ne devraient pas y être ! On utilise une *cascade*: deux messages, unaires ici, sont envoyés au meme objet.
- Ensuite une image de l'écran est capturée. Cette image est transformée ensuite en SketchMorph est réduite de taille puis montrée à l'écran. `sketchMorph scalePoint: ((1/2) @ (1/2))` mérite d'être expliqué. Il s'agit un envoi de message à mots-clès dont l'argument est un point. Ici @ rend un point dont les coordonnées sont 0.5, 0.5. Les parenthèses sont nécessaires afin de résoudre les priorités entre / et @ qui sont des messages binaires donc de meme priorité. Dans l'expression 1/2 @ 1/2, les messages sont envoyés de gauche à droite donc on aurait obtenu un point avec 0.5@1 le tout divisé par 2!. La simplicité du modèle n'est pas gratuite.

A propos de concurrence. Squeak possède les briques de bases de la programmation concurrente: `Thread`, `Delay`, `Semaphore`,... Squeak comme la plupart des Smalltalk définit un modèle de processus légers et inclut son propre ordonnanceur de taches. Chaque processus est ainsi géré dans des queues de différentes priorités. (Certains Smalltalks permettent de créer des processus natifs, Squeak n'offre pas cette possibilité). Cet ordonnanceur est complètement écrit en Smalltalk et donc peut être complètement changé. C'est ainsi qu'un Smalltalk Real-Time avait vu le jour. SqueakNOS (http://sourceforge.net/projects/squeaknos/) qui part du principe que Squeak peut être un aussi un OS (Operating System) se propose donc d'éliminer simplement l'OS et d'utiliser Squeak en remplacement.

Vous pouvez inspectez `Processor` qui une variable globale pointant sur une instance de la classe `ProcessorScheduler`. Cette instance contient des listes chaînées qui représentent les différentes queues d'attentes dans lesquelles sont rangées les processus en attente.

2. Tableaux dynamiques ou statiques

Le mois dernier nous avons vu la syntaxe complète de Squeak. Afin de vous permettre de comprendre du code. Nous vous montrons aussi la différence entre des tableaux statiques (appelés tableau de littéraux, literal array)et les tableaux dynamiques. De plus, nous montrons une particularité de Squeak qui inclut aussi `{}` qui est une sorte de macro-expansion permettant de définir des tableaux dynamiques contrairement à l'expression `#()` qui crée des tableaux statiques dont les éléments doivent être connus à la compilation de la méthode et non à l' exécution. Par exemple essayez:

```
| sta |
sta := #((1 + 2) 4).
sta at: 1
-> $(
```

L'expression suivante rend le caractère `$(` et non le nombre `3` comme on aurait pu si attendre. En effet, les éléments d'un tableau définit à l'aide de `#()` ne sont pas évalués et donc doivent être connus lors de la compilation. Par contre, l'expression suivante elle rend bien 3 car l'expression est évaluée.

```
| dyn |
dyn := Array with: ( 1 + 2) with:4.
dyn at: 1
-> 3
```

Notez que cette expression est équivalente à l'expression suivante:

```
| dyn |
dyn := Array new: 2.
dyn at: 1 put: (1 + 2).
dyn at: 2 put: 4.
dyn at: 1.
```

D'autre part, {} permet de créer des tableaux dynamiques. Ainsi, `{(1 + 2) . 4}` est équivalent à `Array with: (1 + 2) with: 4.` Je suis personnellement contre l'ajout de cette impureté dans le langage qui complique le compilateur. Si l'on veut avoir des tableaux dynamiques il suffit de créer un tableau dynamique en utilisant `Array`!

3. Mais où sont passées les ...

Comme vous l'avez sûrement remarqué la syntaxe de Smalltalk n'inclut ni de définition de classes ou méthodes, ni les boucles, ni les conditions. Et oui, tout cela est fait par envoi de message (invocation de méthodes) définies sur des classes pertinentes. La création de classe se fait par envoi du message `subclass:instanceVariablessubclass: instanceVariableNames:classVariableNames:poolDictionaries: category:` à une classe, avec la description des variables d'instances et autres informations purement liées à l'organisation des classes. Nous reviendrons sur ce point lors d'un autre article.

Par exemple, les conditions `ifTrue:`, `ifTrue:ifFalse:`, `ifFalse:`, `ifFalse:ifTrue:` sont des méthodes définies sur les classes `Boolean`, `True`, `False`. Si l'on part de l'hypothèse que les booléens sont des objets, la seule manière élégante de définir les méthodes sur les classes `True` et `False` de manière polymorphique. Par exemple, la méthode `not`

sur la classe `False` rend `true` et sur la classe `True` rend `false`. Ici `True` est la classe de l'instance `true`. Nous reviendrons sur la conception objet intéressante de ces classes dans un future article.

```
(2 odd)
   ifTrue: [ 'odd']
   ifFalse: ['even']
```

Ainsi `ifTrue:ifFalse:` est simplement une invocation de méthode envoyée à un booléen (ou une expression rendant un booléen). Les codes à exécuter sont des arguments de la méthode. Ici les arguments ne devant être exécuter que de manière conditionnelle, on a besoin d'utiliser un bloc ou fermeture lexicale afin de différer leur exécution. `whileFalse:` et `whileTrue:` fonctionne de la même façon mais l'objet exécutant la méthode doit être aussi un bloc comme le montre un des scripts de la section précédente. Smalltalk propose aussi `timesRepeat`: qui est défini sur la classe `Integer`.

Parmi les nombreuses façons de boucler sur une collection Smalltalk propose la méthode (et non mots-clés) `do:`. Cette méthode est définie sur la classe Collection. Elle est bien-sur optimisée par la machine virtuelle (voir la discussion à la section 4). Ouvrez un `Transcript` qui correspond à une sorte de sortie standard de l'environnement Smalltalk (**Open -> Transcript**) et compilé l'expression suivante dans votre workspace (choix **accept**).

```
#(1 2 3 4 5) do: [:each| Transcript show: (100 + each) printString ;cr]
```

L'argument passé à la méthode `do:` est une fermeture lexicale aussi appelé un block ou block fermeture (une lambda-expression pour les lispiens). De loin, on peut voir cela comme une classe inclue (inner class). En fait, les inner classes ont été introduites en Java car il n'y avait pas d'équivalent de fermeture et cela manquait cruellement. Donc on pourrait dire que les inner classes sont les fermetures lexicales du pauvre car les closures peuvent être manipuler comme n'importe quel objet, c'est-à-dire affecter à une variable, passer en argument lors d'une invocation, retourner par une méthode.

Ici la fermeture, que l'on aurait pu associer à une variable puisque c'est un objet de première classe, nécessite un argument que nous avons choisi d'appeler `each`. La syntaxe pour déclarer un argument de bloc est: `[` pour déclarer un bloc, `:nomdelavariable`, `|` pour la fin de la déclaration. `:each` est dont la déclaration d'une variable nommée `each` qui est utilisée en tapant simplement each dans le corps du bloc.

La librairie de Collection de Smalltalk est riche et a été souvent copiée en C++ (Rogue Wave C++ Collection Librairies) et Java. Essayez donc un petit printIt sur les expressions suivantes:

```
#(1 2 3 4 5) select: [:each| each odd]
#(1 2 3 4 5) collect: [:each| each odd]
#(1 2 3 4 5) includes: 3
```

Squeak est en retard au niveau des fermetures lexicales par rapport aux autres Smalltalk. En Squeak les fermetures sont simulées, les arguments de fermetures sont définies dans la méthode qui la contient. Certaines implantations sont en cours.

4. Prenons de la distance

Nous aimerions maintenant vous faire réfléchir à la différence entre un modèle et son implantation. Ce n'est pas parce que les types comme les booléens ou les entiers sont des objets à part entière d'un langage que ce langage est lent. En effet, en Smalltalk il n'y a pas de distinction entre 1 qui est un SmallInteger et un browseur de code qui est une instance de la classe Browser. En fait, il n'y a pas de différence *conceptuelle mais il y a une différence d'implantation.* En effet, c'est le rôle de l'implanteur d'un langage de proposer un modèle conceptuel propre et uniforme mais ayant une implantation efficace.

Ainsi en Smalltalk, les objets et les méthodes associées comme les booléens avec les conditions ne font pas partie du langage mais le compilateur les optimise de manière drastique. En fait, la méthode`ifTrue:` n'est *jamais* exécutée (sauf si on est fou et demande au compilateur de ne plus l'optimiser). De la même manière, les petits entiers sont extrêmement optimisés. Par exemple, il ne font pas référence comme on pourrait le croire à un objet normal mais la référence est elle-même cet objet. En conclusion, il faut éviter de mélanger (comme le font même certains concepteurs de langages récents) le modèle conceptuel et son implémentation. Autant l'un peut être naïf et simple, autant l'autre peut être extrêmement complexe. C'est le cas de Smalltalk.

Pour la petite histoire, `bitShift:` n'est pas un élément du langage mais juste une méthode définie sur `Integer`. Elle est bien-sur optimisée car elle est centrale pour la machine virtuelle. Notez d'autre part, que Smalltalk offre une coercion automatique est des grands nombres de manière automatique sans redéfinition d'opérateurs. Il n'y a pas de notion d'opérateur en Smalltalk seulement des méthodes.

```
1 bitShift: 200
  1606938044258990275541962092341162602522202993782792835301376

(1 + SmallInteger maxVal) class
   LargePositiveInteger
```

Cet exemple illustre la coercion automatique entre les nombres. Si on ajoute à un petit entier le plus grand petit entier on obtient un grand entier.

5. Squeak et ses grands frères

Il n'y a jamais eu autant de Smalltalk disponibles sur le marché qu'aujourd'hui. Squeak n'est qu'un de ces Smalltalks. Squeak qui est le dernier né, est un des plus ludiques et portables mais aussi un des plus lents et certainement le plus bouillonnant ce qui ne va pas sans poser quelques problèmes de qualité du code. Comme nous l'avons déjà mentionné une fondation et un effort de refactorisation du code sont en cours d'élaboration et un JIT existe pour la version mac depuis plusieurs années déjà mais il reste du chemin à parcourir.

Peut-être de nombreux programmeurs séduits par Java vont apprécier à leur juste valeur les qualités des Smalltalks: uniformité, simplicité, proximité avec le code et les objets, conforts de développement et stabilité des systèmes produits. C'est dans l'objectif de montrer que Squeak n'est qu'un Smalltalk open-source parmi de nombreux produits professionnels de qualité que nous esquissons les autres Smalltalk existants. Nous ne présentons que les Smalltalks encore développés.

- Dolphin Smalltalk est un Smalltalk professionnel intégré à Windows développé par d'anciens programmeurs C++ séduit par Smalltalk qui est vraiment très bon marché et propose une alternative intéressante à Visual-Basic. www.object-arts.com
- Gnu Smalltalk est un autre Smalltak open source. Il tourne sur la plupart des Unix et partout ou une librairie POSIX est disponible. Il propose la possibilité de scripter les applications en mode non interactif. www.gnu.org/software/smalltalk/
- Gemstone est une base de données active orientée objet qui inclut son propre Smalltalk tournant sur le serveur ou client. www.gemstone.com/products/s/index.html
- Object Studio est un Smalltalk qui a évolué à partir d'un 4GL. Il est très utilisé dans les "coorporate environments". www.cincom.com/smalltalk/
- Pocket Smalltalk est un Smalltalk open source pour Palm-Pilot basé sur de la cross-compilation. www.pocketsmalltalk.com/
- Quasar Smalltalk et SmallScript. Quasar Smalltalk a été un des Smalltalks développé en interne pour Apple jusque dans les années noires d'Apple. Il offre la possibilité d'avoir des threads natives www.qks.com/ . SmallScript est un nouveau Smalltalk développé par la même équipe qui intègre la plupart des facilités de scripts des langages comme Perl, Python ou Ruby. SmallScript dans sa version Windows est intégré dans .NET et permet la génération de dll très efficace. SmallScript sera disponible cet été à www.smallts-cript.com/ et cet automne sur MacOS.
- Smalltalk/X est un Smalltalk qui tourne sur Unix et Windows NT. En autre, il permet d'exécuter du code Java. Ce Smalltalk a été utilisé pour des applications critiques pour Alstom.www.exept.de/
- Smalltalk MT est un Smalltalk qui n'est pas basé sur une technologie de machine virtuelle mais produit directement de l'assembleur pour PC. Il permet une efficacité proche de 100% du C, il permet une programmation très bas niveau des threads, dll. Smalltalk MT est utilisé pour la programmation de jeux sur la XBox. www.objectconnect.com/
- VisualWorks est le brillant descendant du Smalltalk développé à Xerox Parc. Il tourne sur Linux, Mac, PC, Unix, HP, Irix et est un des plus rapides des Smalltalk à base de JIT. C'est le Smalltalk qui sert www.ez-board.com et le systèmes de traçages des paquets de UPS. www.cincom.com/smalltalk/
- VisualAge Smalltalk and IBM Smalltalk est le Smalltalk développé par IBM. Il est très implanté dans l'industrie, possède une très grande bibliothèque de composants. VisualAge Java ainsi que les VMs comme celles de VisualAge Macro-Edition sont écrites en Visual Age Smalltalk. www.ibm.com/software/ad/smalltalk/

6. Références

Si vous voulez en savoir plus voici quelques informations utiles.

- http://www.squeak.org/ est le site officiel.
- http://minnow.cc.gatech.edu/ est le wiki de la communauté (wiki = un serveur de pages web que tout le monde peut éditer), il regorge d'informations.
- Le guide rapide de la syntaxe Squeak: http://www.mucow.com/squeak-qref.html
- European Smalltalk User Group est gratuit: http://www.esug.org/
- Mon cours sur Smalltalk: http://www.iam.unibe.ch/~ducasse/