Réflexion en Squeak

Dr. S. Ducasse

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

Squeak comme tous les Smalltalks est un langage *réflexif*. Etre réflexif pour un langage signifie permettre *l'introspection, i.e.*, permettre d'analyser les structures de données qui définissent le langage lui-même et *l'intercession,* i.e., permettre de modifier depuis le langage lui-même sa sémantique et son comportement....

Déjà, par expérience, je vous entends dire mais comment on peut développer de vrais programmes si l'on peut changer le langage? Exact! L'idée est que tout le monde ne change pas le langage mais que seul un méta-programmeur, un programmeur travaillant qu niveua du langage, introduit à la demande des opérations nécessaires que les autres développeurs vont utiliser. Si vous voulez apprendre plus sur les implantations ouvertes et les protocoles de méta-objets lisez [Kicz91] [Kicz96]. Sachez aussi que la programmation réflexive est à la base de la programmation par aspects. D'autre part, de nombreuses applications industrielles sont développées en CLOS qui un des premiers langages Lisp à objets à être réflexif et à proposer un protocole à base de méta-objets (voir www.franz.com, www.nichimem.com) et en Smalltalk comme le système de suivi de paquet de UPS ou EasyBoard (www.ezboard.com) qui a un cluster de 100 PCs en parallèle utilisant VisualWorks comme serveur et que cela ne pose pas de problèmes spéciaux.

Le mois dernier nous vous avons montré les aspects introspectifs de Squeak.Dans cet article, nous allons montrer quelques exemples des aspects *intercessifs* de Squeak. Les Smalltalks sont écrits en Smalltalk, aussi de nombreux aspects du système peuvent être modifiés comme l'envoi de message, la façon dont le ramasse-miettes fonctionne, les processus, l'ordonnanceur de tâches, la pile d'exécution..... Dans cet article, nous nous limitons à des exemples liés au contrôle du comportement des objets et qui peuvent nous faciliter la vie en phase de prototypage. Dans un prochain article, nous souleverons le couvercle un peu plus...Si vous êtes intéressé(e)s, sachez que la réflexivité fait de Smalltalk un excellent langage d'expérimentation. Lisez les articles suivants pour en savoir plus [Gold89], [Riva96], [Bran98], [Duca99]. De nombreux travaux de recherches comme la conception de nouveaux langages, la programmation d'objet concurrents ou distribués ont utilisé Smalltalk principalement pour ces aspects réflexifs et sa malléabilité.

Nous montrons comment les aspects réflexifs sont utilisés pour construire des fonctionnalités qui aident le prototypage d'applications et la création d'outils d'espionnage. Notez que les auteurs de Design Patterns Smalltalk Companion utilisent les techniques présentées ici pour implanter des Proxy [ABW98].

Commençons par passer en mode Morphic (menu **open...Morphic project** puis **enter**). Lorsque vous faites des expériences, il est possible que vous endommagiez des parties vitales du Squeak donc prenez comme habitude de bien comprendre ce que vous faites, de travailler sur un système vierge et de sauver avant de tester!.

1. Quelques Bases

Nous vous montrons les deux points de Smalltalk que nous allons ensuite utiliser: le changement de référence et la récupération d'erreur. Ces deux points seront utilisés dans la construction de l'espion. En Smalltalk, il est possible de substituer toutes les références pointant sur un objet par les références pointant sur un autre. On peut ainsi substituer un objet par un autre. On dit aussi qu'un objet devient un autre car du point de vue du prorgramme une variable pointant sur un objet, pt1, dans l'exemple, pointe sur pt2, après exécution de `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. “a 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.
```

Différentes façons existent pour contrôler les messages envoyés à un objet, et ainsi de créer des messages asynchrones, distribués ou outils d'espionnage [Duca99]. Nous expliquons la méthode la plus simple qui est basé sur la récupération d'erreur. Lorsqu'un message `m` est envoyé à un objet, si la classe ou superclasses de cet objet n'implante pas de méthode `m`, la méthode `doesNotUnderstand:` est invoquée sur l'objet receveur du premier message. Par défaut, la méthode de `doesNotunderstand:` lève une exception qui aboutit à l'ouverte d'un débuggeur. En spécialisant cette méthode on peut donc contrôler les messages non compris par un objet. Pour contrôler les méthodes existantes, une des techniques est alors de créer un objet minimal i.e., qui n'implante que les méthodes vitales et donc invoque la méthode `doesNotUnderstand:` pour toutes les autres.En Squeak, la classe `ProtoObject` est une classe minimale. L'implantation de l'espion est basée sur cette technique. Maintenant nous allons montrer une utilisation de la récupération d'erreur et finalement créer un espion.

2. Des accesseurs invisibles

Au cours de leur longue expérience, certains programmeurs Smalltalk ont développé des patterns qu'ils utilisent lors des phases de prototypage des applications. Smalltalk Scaffolding Patterns [DA00], littéralement patterns Smalltalk pour la construction d'échafaudage, propose une liste complète dont nous présentons ici seulement le premier.

Imaginons que l'on prototype une application dont on ne soit pas sûr de sa structure. C'est bien pour cela que l'on prototype!!! Et que l'on ne veuille pas éditer en permanence la définition des classes, changer les accesseurs à chaque fois que l'on découvre que l'on a besoin de nouvelles variables. Ceci peut être simplement due au fait que l'on peut être plusieurs à développer le prototype en parallèle et que l'on ne veut pas constamment demander aux autres développeurs de modifier leur classes.

Une solution est d'utiliser un dictionnaire afin de stocker les variables d'instances et leur variables. Définissons la classe `Stuff` comme suit avec la variable `variables` que l'on initialise 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 suit:

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

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

On peut alors définir un accesseur par exemple `widget` comme suit:

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

Cependant, cette façon de faire implique la définition des accesseurs ou l'utilisation explicite du dictionnaire, ce qui est pour le moins ennuyeux. En fait, on aimerait utiliser un accesseur sans avoir à le créer et que l'accesseur accède le dictionnaire automatiquement.Ainsi on veut que `widget` soit automatiquement transformé en`variables at: #widget.`

La solution est de spécialiser la méthode `doesNotUnderstand:` afin de récupérer les messages incompris et de les convertir automatiquement en accès à notre dictionnaire. 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 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 représentation n'est effectuée qu'en cas d'erreur afin de ne pas pénaliser l'exécution normale d'un programme.

Cette méthode tout d'abord regarde s'il s'agit d'une méthode ayant 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. Notez 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`. Nous vous laissons cela comme exercice.Pour bien comprendre, mettez un `self halt` à l'intérieur de la méthode et inspectez son argument.

3. Génération d'accesseurs

Lorsque nous avons terminé la phase de prototypage, nous aimerions 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 vous proposons de générer automatiquement le code des accesseurs. Nous vous laissons comme exercice d'imaginer des solutions pour changer la définition de la classe afin de remplacer le dictionnaire par les variables effectivement utilisées en utilisant les techniques que nous avons montrées dans notre précédent article.

Pour générer automatiquement les accesseurs, nous remplaçons la méthode `doesNotUnderstand:` afin qu'elle compile des accesseurs de 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 sure que la précédente: en effet, nous vérifions que le message non compris est unaire ou a mot cles avec un seul argument. Ainsi si l'on invoque la méthode xxx:yyy: non aurons une erreur et ne compilons pas de méthodes. 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 nous envoyons le message `widget` à une instance de la classe `Stuff`, l'accesseur

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

sera compilé.

4. Espionnage

Un des exemples typiques d'utilisation du contrôle de l'envoi de messages est d'espionner des objets à des fins d'optimisations ou de compréhension des applications. Nous allons construire un espion très rudimentaire. Sauvez votre image avant toute exécution d'objets nouveaux!

L'idée est de définir un espion est de le substituer à la place de l'objet que l'on veut espionner. Pour cela, nous allons définir une sous-classe `Spy` de `ProtoObject` ayant comme variable d'instance `spyedObject` qui représente l'objet espionné. `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 autres les messages.

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

Définissez la méthode `on:` 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
```

Ensuite nous définissons la méthode de classe `on:` qui installe l'espion.

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

Une fois l'expression `spy become: anObject` executée notez que la variable `spy` pointe sur l'objet et la variable `anObject` pointe l'instance de `Spy` nouvellement créée.Maintenant ouvrez un `Transcript` (open...Transcript) et évaluez l'expression suivante dans un workspace (open...Workspace)

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

Vous devez obtenir une trace des messages envoyés.

Remarque. Cette technique a les limitations suivantes: on ne peut pas espionner les classes car elles ne supportent pas de `become:`, ensuite, si un objet envoye un message à l'espion qui lui le renvoie à l'objet espionné et que celui-ci retourne `self`, on peut alors envoyer des messages à ce résultat et donc courcircuiter l'espion. De la même façon, si un objet espionné est stocké dans une collection comme un dictionnaire, la différence entre les nombres de hachage entre l'espion et l'objet espionné peut provoquer des comportements très étranges. J'ai essayé d'appliquer des espions sur des morphs et cela a laissé mon système dans un état étrange. D'autres techniques plus lourdes existent pour contrôler les objets [Duca99].

Une version plus sophistiquée de la méthode `doesNotUnderstand:` peut aussi afficher quel objet a envoyé le message espionné garce 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
```

A propos du Proxy. L'implantation du design pattern Proxy proposée dans [AWB98] utilise les deux techniques présentées (changement de référence et récupération d'erreur). Nous décrivons succintement l'idée et vous laissons l'implémenter. Un Proxy est un objet représentant un objet complexe auquel il délègue les messages le cas écheant. Pour son implantation en Smalltalk, l'idée est que le proxy est un objet minimal qui génère des erreurs qui sont capturées. Lors de la capture, l'objet que le proxy représente va être créé puis substitué via `become:` au proxy. Ainsi toutes les références au proxy sont mises à jour seulement lorsqu'une des fonctionalités du Proxy est devenue nécessaire et l'on évite de continuellement déléguer les messages. Imaginons que l'on ait la classe `Image` et une classe `ImageProxy` incluant des informations afin de créer une image comme un chemin de fichier, 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
```

5. Squeak News et Pointeurs

La version 3.1 est maintenant en bêta version et sera bientôt en final. Nous espérons pouvoir la joindre au CD du mois de Décembre. Un effort de matérialisation va ensuite débuter. De plus, nous pensons savoir de sources non-officielles qu'un livre en français va être disponible pour Noël ou au début de l'année mais ce projet est encore secret. Sauf pour les lecteurs de Programmez!

Si vous voulez en savoir plus voici quelques liens utiles.

- http://www.squeak.org/ est le site officiel.
- http://www.squeakland.org/ est le site officiel des projets créés par les enfants utilisant Squeak.

- 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 propose une liste de émail gratuite: http://www.esug.org/
- Un nouveau Wiki pour les Smalltalkiens Francophones est né: aidez le! http://www.iutc3.unicaen.fr:8000/fsug/
- Mon cours sur Smalltalk: http://www.iam.unibe.ch/~ducasse/
- Il y a maintenant un fanzine complètement écrit en Squeak: http://www.squeaknews.com/

6. Références

[AD00] J. Doble et K. Auer, Smalltalk Scaffolding Pattern, Addison-Wesley, 2000.

[ABW 98] S. Alpert, K. Brown et B. Woolf, The Design Patterns Smalltalk Companion, Addison-Wesley, 1998.

[BFJR 98] J. Brant, B. Foote, R. Johnson et D. Roberts, "Wrappers to the Rescue," Proceedings ECOOP'98, LNCS 1445, 1998, pp. 396—417.

[Duca99] S. Ducasse, Evaluating Message Passing Control Techniques in Smalltalk, Journal of Object-Oriented Programming (JOOP), vol. 12, no. 6, June 1999, pp. 39-44.

[Gold89] A. Goldberg et D. Robson, Smalltalk-80: The Language, Addison-Wesley, 1989,

[Kicz91] G. Kiczales, J. des Rivières and D. G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

[Kicz 96] G. Kiczales Beyond the Black Box: Open Implementation, IEEE Software, January 1996, http://www.parc.xerox.com/csl/groups/sda/projects/oi/

[Riva96] F. Rivard, Smalltalk: a Reflective Language, Proceedings of REFLECTION'96, 1996, pp. 21-38.http://www.emn.fr/recherche/recherche02.html