Introspection en Squeak

Dr. 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-meme et *l'intercession*, i.e., permettre de modifier depuis le langage lui-meme sa sémantique et son comportement. Notons que bien que Java définisse une interface réflexive, il n'est pas réflexif et seulement permettant une introspection limitée. Dans cet article, nous allons montrer quelques exemples des aspects introspectifs de Squeak et nous aborderons le mois prochain l'intercession. Nous avons choisi des exemples pratiques qui peuvent vous aider dans votre programmation quotidienne. Commençons par passer en mode Morphic (menu **open...Morphic project** puis **enter**).

1. Ouvrons le couvercle: instances.

Lors du précédent article montrant comment définir une classe, nous avons montré comment utiliser un inspecteur: Un inspecteur est un petit outil de développement qui permet de voir et modifier les valeurs des variables d'instances d'un objet ou de lui envoyer des messages. Créez un workspace avec le script suivant.

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

Vous obtenez un inspecteur qui dans la partie gauche présente les variables d'instances (ici `dependents, contents et bindings`) et la valeur de la variable sélectionnée dans la partie droite. Si vous sélectionnez la variable `contents` qui représente le texte contenu dans le workspace, vous obtenez une chaîne vide. Maintenant tapez `'1+2'` à la place de la chaîne vide puis faites **accept**. La valeur de la variable `contents` a changé. Pour voir son effet dans le workspace évaluer (**do it**) `self contentsChanged` dans la partie basse de l'inspecteur car l'accesseur `contents:` ne propage pas ses changements automatiquement.

Comment l'inspecteur fonctionne-t-il ? En Smalltalk, les variables d'instances sont protégées donc si une classe ne définit pas d'accesseurs il est théoriquement impossible d'y accéder. Or l'inspecteur permet d'accéder à n'importe quelle variable d'instances. L'inspecteur utilise les possibilités réflexives de Smalltalk. L'inspecteur est basé sur le fait que l'on peut accéder à la valeur d'une variable d'instance grâce aux méthodes `instVarAt:, instVarAt:put:, instVarNamed: instVaraNamed:put:` définies sur la classe `Object`. `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: uneValuer` permet de changer la valeur de la variable d'instance représentée par uneChaine.Notons que ces méthodes sont utiles pour cons-

truire des outils de développement ou débogage mais leur utilisation lors de développement d'applications est à proscrire.

Le script suivant montre comment on peut afficher dans le transcript (menu **open...Transcript**) les variables d'instances et leurs valeurs d'une instance de la classe `SystemWindow` sélectionnée au hasard.La méthode `allInstVarNames` permet d'obtenir toutes les 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 meme 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: [ :c | (c instVarNamed: 'systemOrganizer') notNil. ]
```

Regardons l'implantation de la méthode `instanceVariableValues` définie sur la classe `Object` qui rend un tableau dont les éléments sont les valeurs des variables d'instances définies exclusivement sur la classe (et non les variables d'instances héritées). Cet exemple montre l'utilité de pouvoir accéder aux variables via leur position. La méthode `intSize` rend le nombre de variable d'instance d'une classe.

```
 (1@2)  instanceVariableValues
rend anOrderedCollection(1 2)

Object>>instanceVariableValues
   "Answer a collection whose elements are the values of those instance variables
   of the receiver which were added by the receiver's class"
     | c |
     c := OrderedCollection new.
     self class superclass instSize + 1 to: self class instSize do:
            [:i | c add: (self instVarAt: i)].
     ^ c
```

Voici d'autres méthodes pouvant servir lors d'outils de développement. `isKindOf: aClass` rend vrai si le receveur est instance de la classe ou superclasse spécifiée. `1.5 isKindOf: Number` rend `true` mais `1.5 isKindOf: Integer` rend `false`. `respondsTo: aSymbol` rend vrai si le receveur sait exécuter la méthode spécifiée par aSymbol. Par exemple,`1.56 respondsTo: #floor` rend `true` car la classe `Number` implante la méthode `#floor` qui arroundit un nombre. `Exception respondsTo: #,` rend `true` ce qui signifie que l'on peut créer un ensemble d'exception en envoyant le message #, à la classe Exception.

Encore une fois ces fonctionnalités sont intéressantes lors de la définition d'outils de programmation et leur utilisation est suspicieuse pour le développement d'applications. Smalltalk offre d'autres possibilités d'introspection sur des objets tels que les méthodes, les piles d'exécution qui sont représentées par des objets seulement à la demande, les processus, le gestion-

naire de mémoire....Mais nous ne pouvons pas tout aborder ici, essayez de lire le code des classes et expérimentez.

2. Classes

Comme montré par 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
```

Regardons maintenant comment les classes en Smalltalk offrent de nombreuses possibilités de références croisées et navigation entre méthodes et classes. Voici quelques exemples qui montrent à quel point les classes permettent de construire d'outils de navigation.

- `Point whichSelectorsAccess: 'x'` rend la liste des méthodes qui accède la variable d'instance x de la classe Point.
- `Point whichSelectorsStoreInto: 'x'` rend la listes de méthodes qui accède en écriture à la variable d'instance x.
- `Rectangle whichSelectorsReferTo: #+` rend la liste de méthodes définies sur la classe Rectangle qui invoque la méthode +.
- `Point crossReference` rend un tableau dont chacune des lignes représente tous les messages invoqués par la méthode étant premier élément.
- `Rectangle whichClassIncludesSelector: #+` rend les classes définissant.
- `Rectangle unreferencedInstanceVariables` rend la liste de variables d'instances qui ne sont pas référencées dans la classe ou ses souclasses.

La classe `SystemDictionary` qui représente les espaces de nom 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 représentant l'espace de nom de base (Squeak utilise un seul espace de nom). Essayez les expressions suivantes:

```
Smalltalk allClassesImplementing: #+.

Smalltalk allSentMessages

Smalltalk allUnSentMessages.

Smalltalk allUnimplementedCalls
```

`Smalltalk allCallsOn: (Smalltalk associationAt: Point name)` cette dernière expression rend toutes les références sur la classe Point.

3. Métriques

Il est possible d'utiliser les capacités introspectives de Smalltalk pour rapidement définir des métriques. Lorsque l'on aborde une application inconnue, calculer des métrique permet d'obtenir une première idée de l'application. Les métriques simples que l'on peut calculer sont: 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 nombres de souclasses, et le nombre total de souclasses. Les expressions suivantes illustrent comment ces métriques peuvent être calculés.

```
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 intéressants dans le domaine des langages à objet est de connaitre le nombre de méthodes qui étendent des méthodes héritées de leur superclasse. Cela permet d'avoir une meilleure compréhension de la relation qui existe entre une classe et ses superclasses.

Voici un script montrant comment identifier les méthodes de la classe `Browser` qui font un appel à une méthode cachée, i.e., une invocation via super de la forme suivante:

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

 Browser selectors
   select: [:eachSelector |
            | method |
            method := Browser compiledMethodAt: eachSelector.
            (method sendsToSuper)]
```

rend les méthodes: #`systemOrganizer`: et #`veryDeepInner`:

Le nombre d'invocation à des méthodes cachées est donc simplement la taille de la collection rendue.

4. Détection de possibles bugs.

Invoquer une méthode masquée via la variable `super` en utilisant un nom de méthode différent que celui de la méthode contenant super peut créer des bugs lorsque la classe est souclassée. Tout bon programmeur évite ce genre de code. Identifier de telles pratiques peut être très simplement réalisé: il suffit de trouver les méthodes effectuant un appel via super qui ne contiennent pas le sélecteur de la méthode analysée.

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

Notons que ce script ne détecte pas toutes les méthodes. Ainsi les méthodes contenant des boucles ou des récursions montrées ci-après ne sont pas détectées.

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

En exécutant ce script vous devez obtenir la méthode `systemOrganizer` qui effectivement pourrait invoquer la méthode `initialize` via `self` et non via `super`.

5. Conclusion et Références

Le mois prochain nous vous montrerons comment les aspects réflexifs servent pour prototyper très rapidement des applications. Nous montrerons aussi comment le controle de l'envoi de message permet d'implanter certains design patterns comme le Proxy et de créer des outils d'espionnage des objets.

Si vous voulez en savoir plus voici quelques informations utiles.

- http://www.squeak.org/ est le site officiel.
- http://www.squeaklang.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 email gratuite: http://www.esug.org/
- Mon cours sur Smalltalk: http://www.iam.unibe.ch/~ducasse/