M2 Informatique - HMIN305 - "Métaprogrammation et Réflexivité"

Année 2020-2021. Session 1, janvier 2021.

Christophe Dony, Blazo Nastov

Durée : 2h00. **Documents non autorisés.**

Notes : On parlera de *Pharo* pour désigner Pharo-Smalltalk où *Smalltalk* est le langage historique proposant entre autres un système de métaclasses implicite universel. *Pharo* est un environnement de développement logiciel fondé sur *Smalltalk*. Similairement, *Objvlisp* est un langage historique proposant un système de métaclasses explicites universel utilisé entre autres dans le système objet de *Common-Lisp* (également nommé CLOS).

1 Utilisation de Système réflexifs

Exercices d'utilisation du méta-niveau de certains languages dont Pharo ou Common-Lisp ou Java.

Si vous ne vous souvenez pas exactement du nom de certaines méthodes, ce n'est pas dramatique, vous pouvez utiliser des nom approchants à condition qu'ils correspondent à des choses existantes. Bien sûr si vous vous souvenez c'est mieux.

1. Sachant qu'en *Smalltalk*, tout est objet, que tout objet est instance d'une classe, et que la méthode `class` rend la classe de son receveur, dites pour chacune des expressions ci-dessous quelle sa valeur et quel le type de cette valeur, par exemple : la valeur de `3 + 4` est le *SmallInteger* 7.
   a) 1 class
   b) 1 class class
   c) 1 class class class
   d) (Pile new) class
   e) (Pile new) class class class
   f) (1 class class class) == ((Pile new) class class class)
   g) ((1 class class class) == ((Pile new) class class class)) class
2. En *Pharo* et *Java* ou autres - Introspection : écrire une méthode booléenne disant si un objet o comprends un message m. Si oui l'envoi du message m à l'objet o exécutera une méthode, si non il provoquera une erreur de type *"message non compris" (doesNotUnderstand* an Pharo).
   Rappel : en syntaxe Java, `o.f(x)` est un envoi de message dont le receveur est o et le sélecteur `f` ; `x` est un argument, `o` est d'ailleurs aussi un argument.
   Une telle méthode est utile :
   a) en typage dynamique de façon générale si l'on souhaite vérifier avant d'envoyer un message
   b) en typage statique en cas d'utilisation d'une couche réflexive et si l'on souhaite vérifier avant de réaliser des envois de message dont le sélecteur est calculé dynamiquement. A noter qu'un compilateur Java par exemple utilise une telle méthode pour compiler un envoi de message.
   Lisez les 3 sous-questions avant de répondre.
   (a) A écrire en *en Pharo*, la méthode s'appelera *understands :*.
   Exemple d'utilisation :

```
1  p := Pile new.
2  p understands: #pop "-> true" "car pop définie sur Pile"
3  p understands: #machin "-> false" "car machin non définie pour une Pile"
4  p understands: #clone "-> true" "car clone définie sur Object héritée par Pile"
5  p understands: #class "-> true"
6  Class understands: #class "-> true"
```

   (b) *Java* : A écrire en Java, en utilisant le package *Reflect*. La méthode sera une méthode statique à deux paramètres `understands(Object o, String selector)`.
   Pourquoi doit-on faire une méthode statique et pas une méthode normale comme en *Pharo* ? Que manque-t-il à la couche réflexive de Java pour cela ?
   (c) Si vous savez écrire une telle méthode dans un autre langage doté d'une couche réflexive, à la place de l'un des 2 précédents, faites le ici.
3. En *Pharo* : Transformation de modèle (transformation d'une classe (un modèle) à runtime.
   (a) Ecrivez une méthode qui s'applique à une classe et qui enlève de son dictionnaire de méthodes toutes les méthodes abstraites (ceci pourrait être une étape dans une transformation de code ou on essaierait de mieux

factoriser les méthodes abstraites). On suppose que les méta-objets représentant les méthodes comprennent le message booléen `isAbstract` (si vous connaissez le vrai nom en *Pharo*, utilisez le).

(b) Pouvez vous faire la même chose avec Java Reflect ? Utilisez les termes "introspection" et "intercession" pour argumenter.

(c) Connaissez vous un système pour Java ou un autre langage avec lequel on puisse réaliser cet exercice ?

2 Utilisation de systèmes à méta-classes - Objvlisp-Clos versus Pharo

Rappel, on a vu en cours un système de méta-classes nommé *Objvlisp*, basé sur la classe *Object* et la méta-classe *Class* dont on rappelle les définitions sous forme d'envois de messages ci dessous dans une syntaxe que nous nommons "syntaxe *Objvlisp*". Cette définition est bien sûr circulaire et nécessite un bootstrap pour être mise en oeuvre, définition et bootstrap que nous avons présenté dans le cours.

Ce système de méta-classes est aussi celui de `CLOS` mais avec des noms différents : (*standard-object* et *standard-class*).

```
1 Class new
2     name: #Class
3     superclass: Object
4     attributs: 'name superclass
          attributs methods'
5     methods: '(new ... primitive code to
          instantiate a class ...)
6               (name: return name)
7               (superclass: return
                    superclass)
8             ...'
```

Listing 1 – Creation de `Class` en syntaxe Objvlisp

```
1 Class new
2     name: #Object
3     superclass: nil
4     attributs: ''
5     methods: 'many methods ...'
```

Listing 2 – Creation de `Object` en syntaxe Objvlisp

On va dans les sous-sections suivantes utiliser ce système puis le comparer au système de méta-classes de `Pharo`.

2.1 Réalisations en *Objvlisp/Common-Lisp/CLOS*

1. Soit une méta-classe *MemoClass* définissant des classes capables de mémoriser la liste de leur instances.
   a) Donnez, dans le système *Objvlisp*, en syntaxe *Objvlisp* ou en syntaxe *CLOS*, une définition de la méta-classe `MemoClass` sans le code des méthodes.
2. Comment décririez-vous la variable utilisée pour stocker pour chaque classe la liste de ses instances.
3. On souhaite créer la classe `Point` des points du plan (les points possédant une abscisse x et une ordonnée y) comme une *MemoClass*. Donnez en *Objvlisp* ou *CLOS* la définition de la classe `Point`.
4. Dites quelle est la méthode *new* exécutée lors de l'exécution de cette définition, expliquez.
5. On crée une instance I de la classe *Point* par I := *Point new*, Dites quelle est la méthode *new* exécutée lors de l'exécution de cette instruction, expliquez.
6. Faites un schéma représentant les objets `I`, `Point` et `MemoClass`
7. Considérons une méta-classe *AbstractClass* dont les instances devront être des classes abstraites, i.e. non instanciables, i.e. des classes pour lesquelles l'envoi du message new provoquera une erreur. Ajoutons à cette description la contrainte suivante : *AbstractClass* doit mémoriser la liste de ses instances.
   Donnez en *Objvlisp* ou *Clos* la définition de la méta-classe *AbstractClass*. Commentez.
8. *meta commentaire : vous pouvez traiter les sections suivantes avant de revenir à cette question.*
   Considérons une méta-classe *ConcreteClass* dont les instances seront des classes instanciables (les classes standard d'une application usuelle). On pose comme contrainte que toutes les classes instanciables devront être capables de fournir la collection de leurs instances. Par contre on demande que *ConcreteClass* ne mémorise pas la liste de ses instances.
   Donnez en *Objvlisp* ou *Clos* la définition de `ConcreteClass`.

2.2 Réalisations en *Pharo*

On reprend en partie l'énoncé de l'exercice précédent (section 2.1) et on le réalise en *Pharo* avec son système de méta-classes. On rappelle que dans ce système, il y a une méta-classe (dite implicite) automatiquement créée et associée à

chaque classe. On accède à l'édition de la méta-classe via le bouton "meta" du browser, ou par programme en envoyant le message `class` à la classe.

1. Créez en *Pharo* la classe `Singleton1` et faites en sorte que ce soit une classe singleton au sens du schéma de conception du même nom ; donc qu'elle ne puisse avoir qu'une seule instance.
   Donnez le code de la méthode *new* que vous avez à définir.
2. Si on souhaitait créer une classe `Singleton1Bis` soit aussi une classe singleton, voyez vous un moyen de réutiliser (sans copier/coller) le code de la méthode `new` de `Singleton1 class`.
3. Au vu des questions précédentes. Donnez une liste de différences (limitations/avantages) entre le système de méta-classes de *Pharo* et celui *d'Objvlisp* ou *CLOS*.

3 Projet phase 1

Vous avez dans votre projet imlanté en *Pharo* un système à méta-classes explicites de type *Objvlisp*.

- Pour cela vous avez créé la classe *Obj* ci-dessous.

```
1 Array variableSubclass: #Obj
2     instanceVariableNames: ''
3     classVariableNames: ''
4     package: 'ObjVSkeleton'
```

   1. Que représente la classe `Obj` dans le système implanté ? Que représentent ses instances ?
   2. Pourquoi est-elle une sous-classe de `Array` ?
   3. Qu'est-ce que `ObjObject` par rapport à `Obj` ?

- Vous avez suivi la réalisation du *bootstrap* du système et à ce propos vous avez lu et exécuté la méthode suivante :

```
1     manualObjClassStructure
2     | class |
3     class := Obj new: 6.
4     class objClassId: #ObjClass.
5     class objName: #ObjClass.
6     class objIVs: self classInstanceVariables.
7     class objKeywords: #(#name: #superclass: #iv: #keywords: #methodDict:).
8     class objSuperclassId: #ObjObject.
9     class objMethodDict: (IdentityDictionary new: 3).
10    ^ class
```

   1. Que fait cette méthode ? Expliquez pourquoi le "manually".
   2. Que fait l'expression Obj new : 6 ? pourquoi 6 ?
   3. Que représentent les noms dans le tableau `#(#name: #superclass: #iv: #keywords: #methodDict:)`

4 Projet phase 2 - Lecture

Vous avez dans votre projet étudié un langage réflexif ou lu un article lié au sujet. Résumez succintement un point particulièrement intéressant que vous avez appris.