Le modèle à surcharge.

Didier Rémy
Octobre 2000

http://cristal.inria.fr/~remy/poly/mot/10/
http://w3.edu.polytechnique.fr/profs/informatique/Didier.Remy/mot/10/

Slide 1

Cours	**Exercises**
1. Principe 2. Exemples 3. Multi-méthodes 4. Difficultés 5. Résolution 6. Cohérence 7. Encapsulation 8. Le langage Nice 9. Summary	1. Nice

Modèle à enregistrement (rappel)

Slide 2

Le modèle à enregistrements est le plus fréquent.

- Les objets sont des enregistrements
- Les variables d'instance et les méthodes sont des champs.
- Les messages sont des étiquettes.
- L'envoi de message est une auto-application : extraction du message de l'objet et application à l'objet
- Les classes sont des enregistrements (prototypes)
- L'héritage est l'extension des enregistrements

Le modèle à enregistrements permet aussi l'encapsulation, *i.e.* de cacher (par abstraction de type, sous-typage ou par limitation de la portée) les variables d'instance.

C'est le modèle du langage Ocaml, mais aussi des langages Java, C++, etc.

Modèle à surcharge (principe)

Slide 3

Dans le modèle à surcharge, les méthodes ne sont pas portées par les objets, mais définies en dehors des objets comme les fonctions dans la programmation traditionnelle.

Un message est une fonction surchargée, parfois appelée une fonction générique. Un message à donc plusieurs implémentations fonctionnant définies pour des objets de classes différentes.

L'envoi d'un message est l'application d'une fonction surchargée. Le mécanisme de résolution de la surcharge sélectionne *dynamiquement* l'implémentation la plus précise en fonction du destinataire, de sa position dans la hiérarchie des classes et de l'ensemble des définitions enregistrées.

Modèle à surcharge (principe)

Slide 4

En résumé :
- Les objets sont des enregistrements mais ne décrivant que leur état interne.
- Les variables d'instance sont des champs.
- Les méthodes sont des fonctions ordinaires.
- Les messages sont des fonctions surchargées.
- L'envoi d'un message est l'application d'une fonction surchargée à le (ou les) objet(s) destinataire(s).

Dans sa version de base, le modèle à surcharge ne permet pas l'encapsulation. (Mais on peut ajouter un mécanisme d'encapsulation au niveau de modules.)

C'est l'approche suivie par les langages CLOS, les fonctions génériques de Scheme, le langage Jazz ou Nice, le langage jouet KOOL. le langage Cecil.

Exemple (pseudo-code)

Slide 5

NB : ce code à la syntaxe "Ocaml" n'est pas du code Ocaml exécutable.

Classes On ne définit que les variables d'instance. Les classes peuvent hériter de classes parentes, dans ce cas, elles héritent de leurs variables d'instances.

```
class point = object val mutable x = 0 end
class cpoint c = object inherit point val couleur = c end
```

Méthodes La classe du receveur est précisé avec la notation @ :

```
method getx  (p @ point) = p.x
method print (p @ point) = print_int (getx p)
method setx  (p @ point) y = p.x <- y
method getc  (p @ cpoint) = p.c
```

Exemples (pseudo-code)

Création des objets

C'est comme la création d'une structure de donnée.

```
let p = new point;;
let cp = new cpoint "noir";;
```

Slide 6

Envoi d'un message

C'est l'application d'une fonction surchargée à un objet.

```
getx p, getc cp, getx cp;;
```

```
getc p;;
```

Mécanisme de résolution

La résolution est dynamique

Les objets portent donc nécessairement en plus de leur variables d'instance une étiquette indiquant leur classe.

Lorsqu'il existe une implémentation pour cette classe, c'est elle qui est choisie. Sinon, on cherche une implémentation de la méthode dans une classe parente la plus proche.

Slide 7

Résolution (super)

On peut modifier le mécanisme de résolution (utiliser la méthode d'une classe parente (super) même s'il y a une définition locale) en utilisant la notation `@`.

```
method print (p @ cpoint) =
  print (p @ point); print_string (getc p)
```

Slide 8

Cela revient à changer le type dynamique de p. (C'est un mécanisme entièrement différent du cast en Java qui ne change jamais la représentation des objets.)

(Une ancienne définition peut toujours être atteinte en le mentionnant explicitement)

```
print p;;
0noir;;
print (p @ point);;
0
```

Slide 9

Multi-méthodes

C'est une généralisation de l'approche précédente permettant d'envoyer un message à plusieurs récepteurs simultanément.

La résolution du message prend en compte tous les destinataires (on parle de *multiple dispatch*) pour choisir l'implémentation la plus précise.

Slide 10 Les langages CLOS, Jazz, Nice, Cecil ont des multi-méthodes.

Exemples (multi-méthodes)

On ajoute une méthode `equal` pour tester l'égalité sur les points. On définit un comportement par défaut pour les paires de points et un comportement spécial pour les paires de points colorés.

```
method equal (p @ point,  q @ point)  = p.x = q.x
method equal (p @ cpoint, q @ cpoint) =
  equal ( p@point, q@point) && p.c = q.c
```

Slide 11

L'égalité d'un point et d'un point coloré n'étant pas définie, elle est traitée comme l'égalité de deux points.

```
equal p cp;;
true
```

Les deux arguments jouent ici un rôle parfaitement symétrique, contrairement au modèle à enregistrements.

Approche mixte

Slide 12

C'est une combinaison du paradigme traditionnel (approche à enregistrements) et des multi-méthodes. Comme dans le modèle à enregistrements, les méthodes sont toujours spécialisées par rapport à un objet, mais ce sont ensuite des fonctions surchargées qui comme les multi-méthodes résolvent le messages sur les arguments supplémentaires, pris simultanément.

Le langage Cecil est le représentant typique de cette catégorie.

Faux-amis

Java a l'apparence d'une approche mixte, car les méthodes sont surchargées. Cependant en Java la surcharge est résolue *statiquement*, donc on ne peut pas parler de multi-méthodes (La surcharge statique est une convenance de notation, mais n'augmente pas l'expressivité).

Codage du modèle à enregistrements

Slide 13

Si l'on oublie le problème de l'encapsulation de l'état interne, il est facile de coder le modèle à enregistrements dans le modèle à surcharge.

Il suffit de
- définir les classes comme précédemment, mais sans les méthodes
- définir les méthodes immédiatement après les définitions de classes.

Le modèle à surcharge est donc plus expressif que le modèle à enregistrements.

En contrepartie, et malgré les apparences, il est aussi beaucoup plus compliqué...

Difficultés du modèle à surcharge

Slide 14

Le modèle implicite précédent cache un certain nombre de difficultés. En particulier,

1. les méthodes peuvent être définies récursivement,
2. le mécanisme de liaison tardive implique une forme de récursion ouverte,
3. le mécanisme de résolution.
4. l'encapsulation entre en conflit avec la définition des méthodes en dehors des classes.

Les deux premières difficultés sont présentes également dans le modèle à enregistrements, où elle sont résolues classe par classe. Ces difficultés sont augmentées ici par la nécessité de traiter le problème globalement. C'est la source des difficultés de typage et de compilation, la compilation séparée étant très difficile.

Le mécanisme de résolution est un problème nouveau.

Restriction sur la formation des messages

Slide 15

Un message est une collection d'implémentations. Il existe deux conditions sur les message pour qu'ils soient bien former.

– Une condition de fermeture pour assurer que la résolution des messages est toujours possible de façon non-ambigü.
– Une condition de cohérence pour assurer que le typage statique est cohérent avec la résolution dynamique.

Mécanisme de résolution

Pour expliquer le mécanisme de résolution, on peut se ramener à un petit calcul avec surcharge (voir le livre de Castagna), mais la présentation reste complexe.

En particulier, il faut faire intervenir le typage dans le mécanisme de résolution qui en dépend.

Slide 16

Nous nous contentons ici d'une présentation informelle des problèmes et des solutions au mécanisme de résolution.

Résolution (cas simple)

En l'absence de multi-méthodes et d'héritage multiple, la résolution est plus simple, car une classe a au plus une classe parente.

Slide 17

Un message m est une liste de méthodes m_i de domaines c_i disjoints (une définition de même domaine qu'une définition existence cache l'ancienne). Lorsqu'il est appliqué à un objet d'une classe c, il suffit de trouver le plus petit élément parmi les c_i dont c est un sous-type (ce plus petit élément est bien défini).

Résolution (cas général)

La résolution dans le cas des multi-méthodes ou de l'héritage multiple est plus difficile, car la relation de sous-typage n'est plus arborescente. Il faut une condition supplémentaire pour garantir l'existence d'une définition la plus précise.

Slide 18

Un message est de la forme $\{m_i : a_i \to b_i\}_{i \in I}$ où a_i est un tuple de types (le type des arguments passés simultanément).

Il faut s'assurer que l'ensemble $\{a_i, i \in I \mid a < a_i\}$ est vide ou admet un plus petit élément pour tout type a. Formellement,

$$\forall a, \forall i, j \in I, a < a_i \wedge a < a_j \implies \exists k \in I, a < a_k \wedge a_k < a_i \wedge a_k < a_j$$

Pour cela, il suffit que pour toute paire a_i, a_j de types incompatibles (de l'ensemble des domaines de m), pour tout type maximal a de l'ensemble de leurs sous-types communs, il existe une implémentation m_k de m de domaine a.

Contre-exemple

On considère les opérations binaires définies sur les entiers ("int") et les flottants ("float") avec la relation `int` $<$ `float`. Le message `plus` est définit par les deux méthodes :

```
method plus (x@float, y@int) = x +. float_of_int  y
method plus (x@int, y@float) = float_of_int  x +. y
```

Slide 19

Ce message est mal formé. En effet, il est défini pour les types `int` $\times$ `float` et `float` $\times$ `int` dont le seul sous-type commun est `int` $\times$ `int` qui est donc maximal, mais pour lequel il n'y a pas d'implémentation.

Il faut donc compléter le message `plus` par une définition pour un argument de type `int` $\times$ "int".

En effet, si le message est appliqué à un argument de ce type, quelle branche faudrait il choisir ?

Cohérence

Slide 20

Pour assurer la cohérence entre le typage statique et l'exécution du programme, il faut imposer une contrainte sur la façon dont les messages peuvent être formés.

Dans l'application d'un message $\{m_i\}_{i \in I}$, l'analyse statique choisit la meilleure branche $m_1 : a_1 \to b_1$ compte tenu de l'information statique connue sur l'argument. Si le type de l'argument à l'exécution est plus précis, une branche plus spécifique $m : a_2 \to b_2$ peut être sélectionnée. Cela est possible si $a_2 < a_1$. Le résultat à l'exécution sera de type b_2, au lieu du type attendu b_1.

La cohérence impose que pour chaque paires de méthodes $m_i : a_i \to b_i$ et $m_i : a_j \to b_j$ composants un message m, si $a_i < b_i$ alors $b_i < b_j$.

Co-variance et contra-variance

Slide 21

Il faut distinguer

- Le recouvrement d'une méthode par une autre (spécialisation) qui est covariant.
- Le remplacement d'une méthode par une autre qui est contra-variant.

Dans le premier cas, la méthode ajoutée ne couvre pas tous les cas de la méthode qu'elle recouvre, l'ancienne méthode reste disponible pour les cas non couverts par la nouvelle, et il faut donc simplement une cohérence entre le résultat des deux méthodes.

Dans le second cas, la première méthode n'est plus disponible, d'où la contra-variance, puisqu'il faut que le domaine de la méthode de remplacement soit plus grand que le domaine de la méthode originale.

Contre-exemple à la cohérence

Imaginons que l'on complète la définition précédente par

```
method plus (x@int, y@int) = num_of_int x +/ num_of_int y
```

où `num` est le type des nombres en précision arbitraire, et supposons que $\texttt{int} < \texttt{num}$.

Slide 22

Cette définition viole l'hypothèse de cohérence. En effet, le message "plus" comporte deux branches de domaines $\texttt{int} \times \texttt{int}$ et $\texttt{int} \times \texttt{float}$ en relation de sous-typage, mais leur codomaine respectif, "num" et `float` ne sont pas en relation de sous-typage (à l'inverse $\texttt{float} < \texttt{num}$).

On pourrait alors appliquer le message à une valeur de type dynamique $\texttt{int} \times \texttt{int}$ mais de type statique $\texttt{int} \times \texttt{float}$. On s'attend statiquement à récupérer une valeur de type `float`, mais on reçoit dynamiquement une valeur de type `num`.

Coercions

La coercion telle que `p@point` n'est plus un simple cast (changement du type statique, qui n'aurait pas d'effet sur le comportement dynamique).

Il doit changer le type dynamique de l'objet pour que la résolution se fasse dans comme si l'objet appartenait à une classe parente.

Slide 23

Bien sûr elle ne peut se faire que si le type statique de l'objet `p` est un sous-type du type vers lequel il est coercé (ce qui implique que la représentation de `p` est compatible (a plus de champs, et les valeurs des champs communs sont compatibles) avec la représentation du type vers lequel il est coercé.

Encapsulation

Slide 24

Pour retrouver une forme d'encapsulation, on peut replacer les méthodes dans les classes. Les méthodes ont alors accès aux variables d'état internes qui peuvent être cachées à l'extérieur qui ne voit l'objet que comme un type abstrait.

Cette approche peut être mixte. Les méthodes qui n'accèdent pas à l'état interne peuvent être placées en dehors de la classe, ce qui donne la possibilité d'une extension transparente (sans avoir à créer de sous-classe).

Mais cette solution reste imparfaite. L'encapsulation peut aussi être ignorée au niveau des objets et laissé à un mécanisme de modules.

Quelques exemples dansle langage Nice

Slide 25

Le langage Nice est un prototype de langage objets avec méthodes comme fonctions surchargées et multi-méthodes. Il se présente comme une extension de Java, avec un système de typage très riche, incorporant du polymorphisme paramétrique.

Ce n'est encore qu'un prototype...

On pourra voir le tout petit tutorial en ligne et quelques exemples.

Sur les machines de la salle 31 (noms d'oiseau : autruche, epervier, mouette, etc.)

C'est installé comme un package Java 1.2 qui se trouve dans ~remy/nice. Le compilateur s'appelle ~remy/nice/bin/nicec et l'interprétre de byte code ~remy/nice/bin/nicer.

Nice : premier programme

On ne peut compiler que des packages

hello/main.nice

```
package hello;
main (args) {
    System.println ("hello,  world!");
}
```

Slide 26

Unix

```
~remy/nice/bin/nicec hello
~remy/nice/bin/nicer hello.jar
```

Nice : les points

Une classe ne déclare que les types des variables d'instances

point/point.nice

```
class Point {
  int x;
}
```

Slide 27

Un constructeur de la classe est défini comme une fonction de construction. On précise d'abord son type, puis son implémentation (en dehors de toute classe)

```
Point point (int);
point (x0) {
  Point p = new Point();
  p.x = x0;
  return p;
}
```

Nice : les méthodes

De même on définit les méthodes en dehors de la classe, en donnant d'abord son type, puis leur implémentation.

```
int getx (Point);
getx (p @ Point) { return p.x; }

void setx (Point, int);
setx (p @ Point, y) { p.x = y; }
```

Slide 28

On peut même surcharger la fonction print de la librairie nice.lang :

```
print (p @ Point) { System.out.print (getx (p)); }
```

On aurait pu aussi choisir un autre nom, bien sûr ; dans ce cas, il aurait fallu déclarer le prototype.

Nice : point coloré

```
class Cpoint extends Point { String c; }
Cpoint cpoint (String);
cpoint (s0) {
  Cpoint p = new Cpoint();
  p.x = 0;
  p.c = s0;
  return p;
}
String getc (Cpoint);
getc (cp @ Cpoint) { return cp.c; }

print (cp @ Cpoint) {
  System.out.print (getx(cp)); System.out.print (getc(cp));
}
```

Slide 29

Nice : l'égalité sur les points

Slide 30

On définit simplement une méthode à deux arguments :

```
boolean equals (Point, Point);
equals (p @ Point, q @ Point) {
  return p.x == q.x;
}
equals (p @ Cpoint, q @ Cpoint) {
  return p.x == q.x && p.c == q.c;
}
```

Nice : le test

Slide 31

points/main.nice

```
package point;

main (args) {
  System.out.print ("p = ");
  Point p = point (1);
  setx (p, getx(p)+1);
  print (p);
  System.out.print ("\ncp = ");
  Cpoint cp = cpoint ("black");
  print (cp);
  System.out.println ("\np = cp ? "
                  + (     equals (p, cp) ? "yes" : "no"));
}
```

Nice : autres exemples

Le système de typage contient des type paramétrés.

Voir par exemple la classe des collections.

Exercice 1 (Nice) *Essayer le langage nice. Mettre en place l'exemple des points colorés.*

Slide 32

En quoi l'implémentation de l'égalité en Nice est-elle préférable à celle en Java ? en Ocaml ?

En Java, il n'y a pas de solution bien typée: on est obligé de perdre de l'information et d'utiliser des coercions pour la récupérer. En Ocaml, la méthode \lst"equal" est bien typée, mais asymmétrique. On doit rendre montrer une partie de la représentation (comme en Nice).

Le modifier pour faire des points à deux dimensions.

Regarder l'exemple des Collections. □

Exercice 2 (Extensibilité bidirectionnelle) *On reprend l'exercice sur en Nice. En particulier, on pourra définir les classes Nil et Cons, puis certaines méthodes sur ces classes, puis définir la classe Append, puis une nouvelle méthode sur l'ensemble des trois classes.* □

Slide 33

Résumé (inconvénients)

L'approche à surcharge est, malgré ses apparences, beaucoup plus difficile, par le caractère global et plus implicite de la récursion et par son mécanisme de résolution.

Elle rend également l'encapsulation plus difficile à moins de replacer les méthodes dans les classes.

Slide 34 La compilation séparée et le typage incrémentale sont plus difficiles et rarement implémentés.

Sa compilation efficace est aussi plus difficile en raison de la résolution dynamique, mais ce problème, qui reste secondaire et souvent résolu par une compilation globale.

Résumé (avantages)

L'approche à surcharge est plus proche de la programmation traditionnelle où les fonctions sont définies séparément des données et s'appliquent à celles-ci.

Elle offre plus de flexibilité dans l'écriture du code source puisque les méthodes peuvent être définies en dehors des classes.

Slide 35 Les multi-méthodes préservent la symétrie des arguments, contrairement aux méthodes binaires dans l'approche à enregistrement.

Conclusions

Slide 36

En conclusion, il faut souligner l'importance du typage et d'une étude formelle des constructions du langage.

Même en suivant cette approche (ce que nous avons fait qu'informellement ici) on rencontre de sérieuses difficultés.

Les objets dans le modèle à enregistrement sont déjà une notion difficile. Les messages comme fonctions surchargés le sont encore bien davantage, malgré, l'apparence trompeuse d'une intuition plus simple.

Enfin, la complexité du modèle trouvé ne doit pas faire oublier la simplicité du modèle recherché, mais celui-ci reste encore à trouver...