Programmation par Objets

Nabil Hameurlain
LIN2

Programmation par Objets (1)

- Caractérisée par la programmation avec des objets, messages, et hiérarchies des objets.
- La PO est une abstraction, généralisation de la Programmation impérative (PI):
  - P.I implique un état et un ensemble d ’opérations transformant l ’état;
  - P.O implique une collection d ’objets, chacun avec un état et un ensemble d ’opérations transformant l ’état.

Programmation par Objets(2)

- S ’intéresse beaucoup plus aux données qu’au contrôle.
- Utilise la métaphore de communication par messages pour capturer les interactions entre objets.

Programmation par Objets(3)

- Programmer en Objets signifie développer des abstractions qui modélisent le problème à résoudre:
  - se focaliser sur les données, les objets, et les opérations nécessaires pour satisfaire une tâche particulière.
- Un objet peut être vu comme un *Serveur*: nécessite des opérations et (parfois) des données.

Programmation VS langages à Objets

- On peut faire de la programmation à Objets dans n ’importe quel langage de programmation.
- Un langage à objets offre des mécanismes qui facilitent la programmation par objets (mécanismes *d’encapsulation*, de *classification*, d *’héritage*, etc).

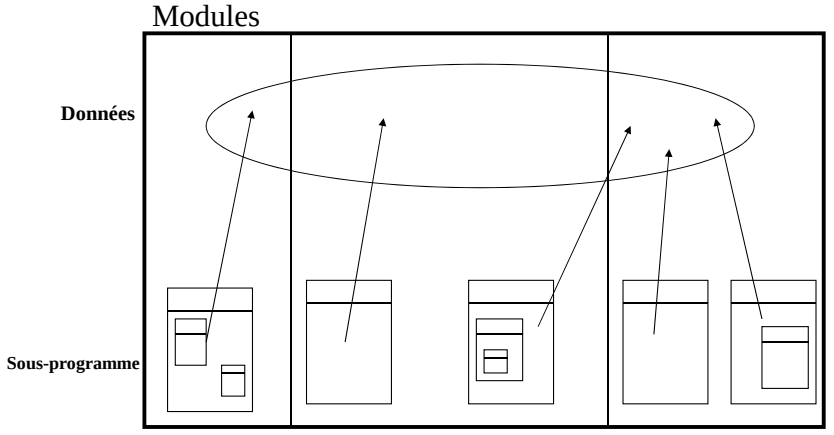
Langage à Objets (1)

- Un Objet renferme son état et ses opérations (**encapsulation**):
  - cache les détails de l ’implémentation d ’un objet.
- Différents objets receveurs d ’un même message peuvent répondre différemment (**polymorhisme**).

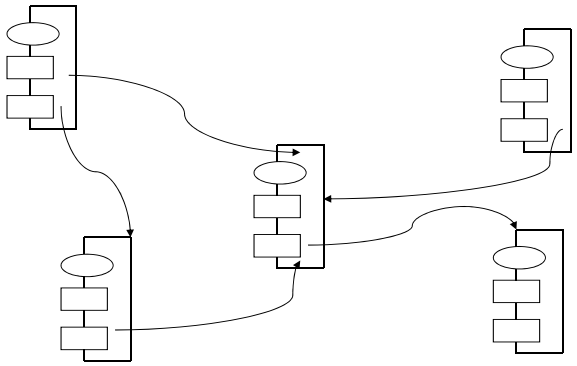
Langage à Objets (2)

- Le code à exécuter en réponse à un message est déterminé dynamiquement (**liaison différée**);
- Les objets sont crées à partir d’un moule (**classe**);
- Une classe peut être construite par **héritage** d’une autre classe déjà existante.
  - La sous-classe peut hériter de la structure et du comportement de sa super-classe.

Historique: approche procédurale



Historique: approche orientée objets



Historique: Objets

- Programmation dirigée par les données et les traitements:
  - la tâche à réaliser est décomposée en un ensemble d'objets.
- Idée : diviser (en sous-programmes autonomes) pour mieux faire évoluer et réutiliser

  Objet = données + traitements

Terminologie (1)

- Objet: collection de données privés et des opérations publics;
- Classe: description d 'un ensemble d 'objets (type encapsulé: partionner en privé et public);
- Instance: une instance d 'une classe est un objet de cette classe;

Terminologie (2)

- *Méthode*: le corps d 'une procédure qui implémente une opération;
- *Message*: appel de procédure. Requête d 'exécuter une méthode;
- *Héritage*: Extension d'une classe précédemment définie;
- *Sous-typage* (principe) : il peut apparaître si un objet d 'un supertype est attendu:
  - c'est un mécanisme d 'héritage qui distingue entre les types et les classes

Les Objets (1)

- Collection d ’opérations partageant un état;
- Les *opérations* déterminent les messages (appels) auxquels l ’objet peut répondre;
- L *’état* partagé est caché depuis le monde extérieur, et est accessible seulement aux opérations de l ’objet;

Les Objets (2)

- Les variables représentant l ’état interne d ’un objet sont appelées des *instances de variables*, et les opérations sont appelées des *méthodes;*
- La collection de méthodes d ’un objet détermine son *interface* et son *comportement*

Types d ’Objets

- Objets *Fonctionnels*: collection de fonctions mais n’ayant pas d ’état;
- Objets *Impératifs*: Objets ayant un état que l ’on peut mettre à jour;
- Objets *Actifs*: peuvent être activés quand un message arrive; contrairement aux Objets fonctionnels et impératifs qui sont passifs, à moins qu’ils sont activés par un message

Objets Fonctionnels

- Ils sont comme les valeurs, ayant une sorte d ’interface objet, mais aucune identité persiste entre les changements d ’état.
- Utilisés dans les langages fonctionnels et logique.
- Syntaxiquement, ils peuvent être représentées par exemple par :

*name**: object***
methods
....

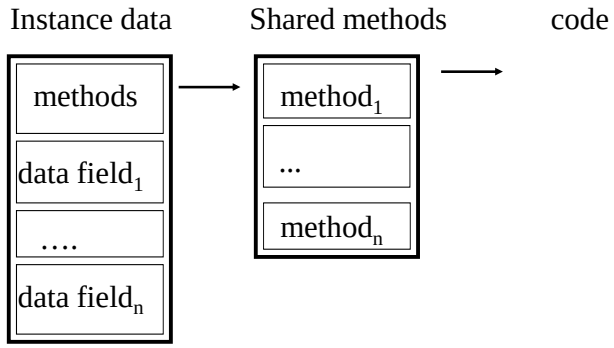
Objets Impératifs

- Sont comme les variables, ayant un état que l ’on peut mettre à jour (Simula, Smalltalk, C++);
- Ont une identité, et une collection de:
  - méthodes qui sont activées par la réception de messages depuis les autres objets
  - Instances de variables qui sont partagées par les méthodes de l ’objet, mais inaccessibles par les autres Objets.

Objets Actifs

- Les Objets Actifs ont trois modes:
  - quand il y a rien à faire, ils dorment;
  - quand un objet (« agent ») est en train de s ’exécuter il est *actif*;
  - et quand un objet est en attente d ’une ressource ou l ’accomplissement des tâches il est en *attente.*
- Les messages envoyés à l ’objet doivent attendre dans une file jusqu’à ce que l ’objet finisse sa tâche:
  - L ’envoie peut être synchrone ou asynchrone.

Implémentation d’un Objet



Les classes(1)

- Les classes servent de moule à partir duquel les Objets peuvent être crées.
- Les classe ont les mêmes instances de variables et d ’opérations que les objets correspondant mais l ’interprétation diffère:
  - Une instance de variable dans l ’objet représente les variables actuelles, alors que les instances de variables d ’une classe sont potentielles, étant instanciées à la création d ’un objet.

Les classes(2)

- On peut penser d ’une classe comme la spécification du comportement commun à tous les objets de la classe.
- les opérations publics d ’une classe déterminent le comportement,
- Les instances de variables spécifient une structure (de données) pour réaliser ce comportement;

Approches de définition de L ’héritage

- *Logique*: exprimé par un prédicat, A(x) est vraie si x $\in$ A. *B hérite de A ssi,*

  $\forall x$, *si A(x) alors B(x).*
- *Ensembliste*: le prédicat est exprimé par une inclusion d ’ensembles. *B hérite de A ssi,*

  $\forall x$, *si* $x \in A$ *alors* $x \in B$.
- *Conceptuelle*: exprimée par une spécialisation.

  B hérite de A ssi B « est une sorte de » A.

L ’héritage (1)

- Permet la réutilisation du comportement d ’une classe dans la définition de la nouvelle classe.
- La *sous-classe* hérite des opérations de la classe parente (*super classe*) et peut rajouter de nouvelles opérations et de nouvelles instances de variables.
- L ’héritage capture une forme d ’abstraction, appelée *super-abstraction*, qui est complémentaire à l ’abstraction de données.

L ’héritage (2)

- Permet la classification des classes de la même façon que les classes permettent la classification des valeurs (premier niveau).
  - Offre un pouvoir de modélisation conceptuelle et de classification (du second ordre).
- Offre la possibilité de gérer, de manipuler, et de partager le comportement (du second niveau), complémentaire à celui des classes vis à vis des objets (premier niveau).

Relation et graphe d ’héritage(1)

- Une relation d ’héritage est préordre (réflexive et transitive), liant une classe à sa super-classe:
  - A hérite (sous-classe) de B est représenté par $A <_h B$;
- La représentation graphique de l ’héritage forme le *graphe d ’héritage* Gh=(Racine, Nœud, Arc):
  - Racine=classe la plus générale,
  - Nœud=l ’ensemble des classes du système.
  - Arc: il existe un arc orienté A---->B ssi A hérite de B.

Relation et Graphe d ’héritage(2)

- *Simple*: une classe ne possède qu’une seule super-classe directe:
  - L ’ensemble des classes est structuré (graphe d ’héritage) sous forme d ’une arborescence.
- *Multiple*: une classe possède plusieurs super-classes directes:
  - le graphe d ’héritage forme un graphe orienté sans circuit.

Les Types et les Classes(1)

- Le concept d ’un type et celui d ’une classe sont liés et peuvent ne pas être distingués:
  - Le typage est utilisé pour la vérification de type;
  - les classes sont utilisés pour gérer et générer des objets ayant des propriétés et des comportements uniformes.
- Toute classe est un type alors qu’un type n ’est pas forcement une classe, puisque les prédicats ne déterminent pas nécessairement des moules d ’un objet.

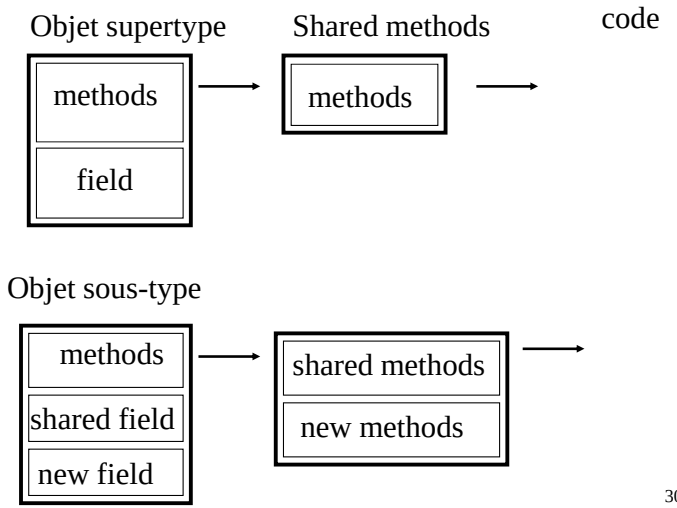
Les Types et les Classes(2)

- La distinction entre les types et les classes peut être détectée quand on compare la relation d ’héritage entre types et sous-types avec la relation d ’héritage entre classes et sous-classes.
- L ’ensemble des valeurs satisfaisant un sous-type est un sous-ensemble de l ’ensemble des valeurs satisfaisant le type.

Sous-types et Sous-classes

- *Exemple1*: les entiers naturels sont un sous-type des réels, mais quand la soustraction est définie pour toutes les paires des réels, elle n ’est pas définies pour toutes les paires des entiers naturels.
- *Exemple 2*: les réels sont une sous-classe des entiers naturels, puisque l ’opération de soustraction sur les entiers naturels peut être étendue à celle des réels.

Implémentation de l ’héritage



Héritage Vs Composition

- L ’héritage multiple permet de définir de nouveaux objets par fusion de la structure et du comportement de plusieurs autres et non par composition.
- La composition d ’objets peut toujours être réalisée par les variables d’instance.
- Exemple:

  Automobile=Moteur+roue+Carosserie, +...

Typage et sous-typage dans les Langages à Objets

N. Hameurlain

Lin2

Objectifs du typage?

- Introduire une plus grande fiabilité dans le langage en détectant les expressions qui ne sont pas bien formées
  - f(x) est bien formée si x appartient au domaine de définition de f.
- Exemples d ’expressions mal formés
  - diviser par zéro, affecter une valeur à un indice hors des bornes d ’un tableau, envoyer un message à un objet qui ne connaît pas, opérateur appliqué sur des objets pour lesquels il n ’est pas défini, etc

Réalisation du Typage

- Regrouper les objets en catégories distinctes: types;
- Définir un ensemble de règles appelées *règles de typage;*
- Simuler l ’exécution du programme en suivant les règles de typage, mais en tenant compte uniquement des types obtenus à chaque étape de simulation;

Catégories de typages

- Typage *dynamique*: réalisé pendant l ’exécution:
  - les tests sont insérés dans le code et compilé, et sont ainsi réalisés pendant l ’exécution.
- Typage *statique*: fait à la compilation
  - la cohérence des types est vérifiée avant la traduction en code machine.

Les problèmes de typage dans les L.O

- Quel est le type d ’un objet, c ’est à dire comment représenter le type d ’un objet (état interne+méthodes);
- Comment typer les utilisations de *self* (par une méthode par exemple d ’un objet)? Ce type change t-il avec l ’héritage?
- Comment typer la liaison tardive?
- Qu’est ce que c ’est le polymorphisme et quel est son lien avec l ’héritage

Langage des types

- Types atomiques (int, bool, float, …);
- Types construits:
  - d'opérateurs prédéfinis : ->, *
    - int --> int, int*int-->int
  - d'enregistrements déclarés (point, cercle, …
- Types récursifs
  - Arbre, etc..
- Notation: on notera t1, t2, t' un type, et *e:t* se lit « l'expression *e* est de type *t* »

Typage d'une expression

- Se fait relativement à un environnement ou contexte (noté E).
- Environnement de typage: Liste de liaisons (idf, type). Exemples:
  - Un idf x est de type t dans un environnement E, si (x, t) est présente dans E;
  - $\frac{e_1: \text{bool dans E} \quad e_2 : \text{t dans E} \quad e_3: \text{t dans E}}{\text{If } e_1 \text{ then } e_2 \text{ else } e_3 : \text{t dans E}}$
  - $\frac{e_1: \text{t->t' dans E} \qquad e_2: \text{t dans E}}{e_1(e_2): \text{t' dans E}}$

Construction de l'environnement de typage

- A partir des déclarations:
  - Toute déclaration introduit une liaison (idf, type)
  - Le système vérifie que l'utilisation de l'idf est conforme au type associé.
- Par synthèse:
  - Le type de chaque idf est calculé à partir des occurrences d'utilisation de l'idf.
    - Pas besoin de déclarer le type.
  - Le calcul est analogue à la résolution d'eqs en mathématiques (notions de constantes, de variables et de contraintes).

Construction de l'environnement de typage (par synthèse)

- Exemples (de règles):
  - Si x est une constante entière, type(x) = int;
  - Si e = (e1, e2), type (e) = type(e1)*type(e2)
  - Si e = (fun x. e1),

    type (e) = type(x) -> type(e1);
  - Si e = e1 (e2), type (e1) = type (e2)->type (e)

I. Le Polymorphisme

- Class point {
  var x;
  method getx{return x;}
  method move (d) {x:= x+d;}
  };
- Class Colorpoint : hérite point{
  Var c;
  method getcolor{return c;}
  method setcolor (c'){c:= c';}
  };
- Soit la fonction Origin (p : point) {
  return(p.getx= =0)}

Le Polymorphisme

- Peut-on appliquer la fonction Origin aux objets de type Colorpoint?
- Oui, s'il existe une notion de polymorphisme dans le langage.
- Définition: une fonction, une méthode ou un objet est polymorphe si on peut l'employer avec plusieurs types.

Problème du polymorphisme

- Soit F(o: t) = … /*opérations sur o*/
- On veut faire F(a) où a : t' et t'≠ t.
  - Comment assurer que cet appel ne provoque pas d'erreurs de typages à l'exécution?
  - Quelles conditions doit remplir tout type t'?
- Il existe trois formes du polymorphisme:
  - Polymorphisme paramétrique;
  - Polymorphisme d'inclusion ou sous-typage
  - Surcharge;

Polymorphisme paramétrique: définition

- Si les opérations effectuées sur o ne sont pas dépendantes de son type (recopie, comparaison, égalité), le type de F est polymorphe paramétrique, et on peut utiliser F avec tout type qui est une instance de ce type polymorphe.

Polymorphisme d'inclusion ou sous-typage: définition

- Si le type t' est tel que: toutes les opérations sur t sont aussi définies sur t', alors on peut utiliser tout objet a: t'où un objet de type t était attendu:
  - t = {réels} et t'= {entiers}
- On dira que t' est un sous type de t ou t' est inclus dans t.

Surcharge

- Le code des fonctions n'est pas partagé, contrairement au poly. par. et d'inclusion.
- Exemple: surcharge d'opérateurs '+' (entiers, réels, etc…)
- Pour chaque utilisation de l'opérateur surchargé, le compilateur doit décider quel code exécuter, et ceci selon le type des arguments donnés.

Polymorphisme (conclusion)

- Le poly. d'inclusion et paramétrique ont en commun une propriété très importante:
  - Une fonction polymorphe peut être appliquée sans changement de code à une infinité de types d'arguments.
- Le poly. d'inclusion et paramétrique est un moyen de partager aussi bien le code source que du code exécutable.

II. Quel polymorphisme pour les objets

- « Peut – on utiliser les opérations définies pour les super-classes sur les objets des sous classes? »
- Exemple: Colorpoint est sous-classe de Point, peut-on utiliser Origin avec des points de couleur.
- L'héritage doit garantir le poly. des méthodes et des opérations des super-classes:
  - N'importe quelle opération définie sur une super-classe doit pouvoir s'utiliser sur une sous-classe.

Polymorphisme paramétrique: variable de types

- Le langage de type est étendu avec des variables, notées a, b, c, …
- Une expression de type peut contenir des var. a-> int, b->b, …
- La valeur de la sous-expression qui est typée par une var. de type n'a pas de contraintes associées et donc n'intervient pas dans le calcul
  - Identite (x) = {x} a pour type a->a
- L'introduction de var. de type permet une abstraction sur le code.

Polymorphisme paramétrique : Instanciation

- Définition: t(a b c) une expression de type qui contient les var. a, b, c. On dit que t' est une instance de t(a b c) s'il existe des types ta, tb, tc tel que t' = t(a b c) [a/ta, b/tb; c/tc], où [x/t] est la substitution de x par t.
- Dans le cas où les types sont déclarés, la substitution est donnée par l'utilisateur.

Polymorphisme paramétrique: définition

- Fonction ayant dans son type des par. ou variable de types qui pourront être *spécialisés* (instanciées) à n'importe quel type
  - L'instanciation d'une variable de type *a* vers un type *t* est notée par [a/t].
- Définition: un type τ' est une instance polymorphe du type τ s'il existe une spécialisation [a/t] des var. de type τ telle que $\tau' = \tau[a/t]$.
- Caractérisé par la relation d'instantiation qui existe entre le type général de la fonction et le type particulier de chaque spécialisation.

Polymorphisme d'inclusion

- Connu sous le nom du sous-typage:
  - pas d'instances de type mais plutôt d'inclusion de type
- Définition 1: un type t' est un **sous-type** d'un (inclus dans un) type t, noté **t' ≤ t**, si n'importe quelle opération définie sur t est aussi définie sur les éléments du type t'.
  - Un objet de type t' peut (être vu) remplacer sans danger un objet de (comme ayant le) type t;
- Définition 2: si *a* a le type t', et si t' est un sous type de t, alors *a* a le type t.
  - Colorpoint est sous-type de Point?

Langage à Objets et polymorphisme

- La plupart des L.O font le choix du Poly. d'inclusion ou sous-typage. Ce mode de poly. pose problème lors de l'héritage:
  - Faire « comme ci » le poly. est « transmis » par héritage: erreur de typage lors de l'exécution.
  - Restreindre l'ensemble des classes que l'on peut écrire: l'expressivité du langage est réduite de manière importante (C++, par exemple).

III. Sous-typage et L.O

  - « La notion d'héritage assure t-elle l'utilisation polymorphe des méthodes héritées? »
- Appel de fonctions avec sous-typage:
  - Sous quelle conditions un appel de fonction est bien typé en présence du sous-typage.
- Sous-typage entre objets:
  - Quand est-ce que deux types d'objets sont en relation de sous-typage?
- Sous-typage entre fonctions:
  - Comment savoir si deux types (fonctionnelles) de fonctions sont en relation de sous-typage

Appel de fonctions avec sous-typage

- Prend en compte le fait que le typage du par. effectif peut être un sous-type du par. formel.
- Règle: $f : \sigma \rightarrow \tau$

  $a : \sigma'$ dans A et $\sigma' \leq \sigma$

  Alors, f(a) est bien typé dans A et a le type τ
  - En effet: si $\sigma' \leq \sigma$ alors toutes les op. sur σ sont aussi définies sur σ'.Ainsi f, qui attend un argument de type σ peut recevoir un type σ' en toute sécurité.

Sous-typage entre objets (1)

- Soit $t = <\mathbf{m_1}: \tau_1; \mathbf{m_2}: \tau_2; \ldots, \mathbf{m_n}: \tau_n >$, quelle est la forme de tout type t' sous-type de t, i.e. qui peut remplacer t de manière sûre?
- On doit pouvoir appliquer aux objets de type t' au moins toutes les opérations valides sur t:
  - t' doit avoir au moins toutes les méthodes de t, mais peut éventuellement en avoir plus.
- Les types de méthodes communes sont en relation de sous-typage:
  - Si m a le type τ' dans t' et τ dans t alors $\tau' \leq \tau$.

Sous-typage entre objets (2)

- Règle: $t = <\mathbf{m_1}: \tau_1; \mathbf{m_2}: \tau_2; \ldots; \mathbf{m_n}: \tau_n>$,

  $t' = <\mathbf{m_1}: \sigma_1; \mathbf{m_2}: \sigma_2; \ldots, \mathbf{m_n}: \sigma_n; \tau'>$, où τ' est une suite de méthodes.

t' est sous-type de t dans A ($t' \leq t$), si $\sigma_i \leq \tau_i$, i =1, …,n;

  - En effet, si o: t et o':t', et si m est une méthode de o avec le type τ , alors o'.m: τ' pourra être vu comme ayant le type τ ssi $\tau' \leq \tau$.
- Remarque: dans la plupart des L.O, l'ordre des méthodes n'a pas d'importance dans la comparaison des types d'objets.
- Exemple: Colorpoint ≤ point

Sous-typage entre fonctions

- Class A = { Class B = {

  …. ….

  method f : t->q} method f : m->n}

  Pour établir B ≤ A il faut vérifier **(**m->n) ≤ (t->q)
- Règle: deux types fonctionnelles, **(**m->n) et (t->q), sont en relation de sous-type, i.e. ***(****m->n) ≤(t->q)*

  ssi *t ≤m et n ≤q.*
- On dit que la règle de sous-typage entre fonctions est co-variante (va dans le même sens) pour les types résultats, alors qu'elle est contre-variante (change de sens) pour les types arguments (en entrée).

Sous-classe ≠ Sous-type (1)

- Le sous-typage entre deux classes impose le sous-typage entre les types de leurs méthodes;
- Le sous-typage fonctionnel doit satisfaire la règle de *variance/covariance*
- Problème: souvent la contre-variance est contraire à la manière dont le programmeur veut définir les classes, les méthodes par héritage, à savoir, par spécialisation des types pour les arguments des méthodes:
- Ce problème connu sous le nom « *Problème de contre-variance* ».

Sous-classe ≠ Sous-type (2)

- Class A= { Class B = *hérite* de A{

  …. …

  method m(a : t); } method m (a : t');}
- Sachant que l'on veut faire une spécialisation des arguments, c à d, t' ≤ t, alors que la règle impose:

  B ≤ A, si t ≤ t' (règle de contre-variance)

Sous-classe ≠ Sous-type: exemple (1)

- Soient les classes: aliments, viande, et légumes telles que:
  - viande ≤ aliments et légumes ≤ aliments
- Class Personne () {

method manger (a : aliments) {}

}

- Class végétarien(): hérite personne {
  method manger (a : légumes){}

}

Sous-classe ≠ Sous-type: exemple (2)

- végétarien est une sous-classe de personne ; Mais est – elle un sous-type de personne?, c'est à dire végétarien ≤ personne (1).
- Si (1) est vrai, par la règle de sous-typage fonctionnelle, on a :
  - aliments ≤ légumes, ce qui est contraire aux hypothèses

Sous-classe ≠ Sous-type: exemple (3)

- Intuitivement: Soit la fonction
  Chez_Macdo (p : personne) = {p.manger(BigMac);}
  où *BigMac*: viande.
- Si (1) est vraie, soit GND: végétarien, qui peut être utilisé à la place du type personne, on obtient donc GND. manger(BigMac)!!!!!!!;
- Cet appel est sémantiquement incorrect, et il est mal typé: la méthode manger d'un *végétarien* n'accepte pas le type viande.
- En réalité (1) n'est pas vérifiée!

Sous-classe ≠ Sous-type: conclusion

- Il est fondamentale de préserver l'égalité,
  « sous-classe = sous-type » dans L.O:
  - Permet d'utiliser les opérations de la super-classe de manière polymorphe.
- Quand cette égalité risque de ne pas être vérifiée
  - Interdire l'écriture de la classe héritière;
  - Imposer à l'utilisateur de lui donner le même type que dans la classe parente, ceci interdit l'utilisation de cette méthode de manière polymorphe (C++).