Algorithmique Programmation Objet

Python

Andrea G. B. Tettamanzi

Université de Nice Sophia Antipolis

Département Informatique

andrea.tettamanzi@unice.fr

CM - Séance 3

Introduction à la programmation orientée objet

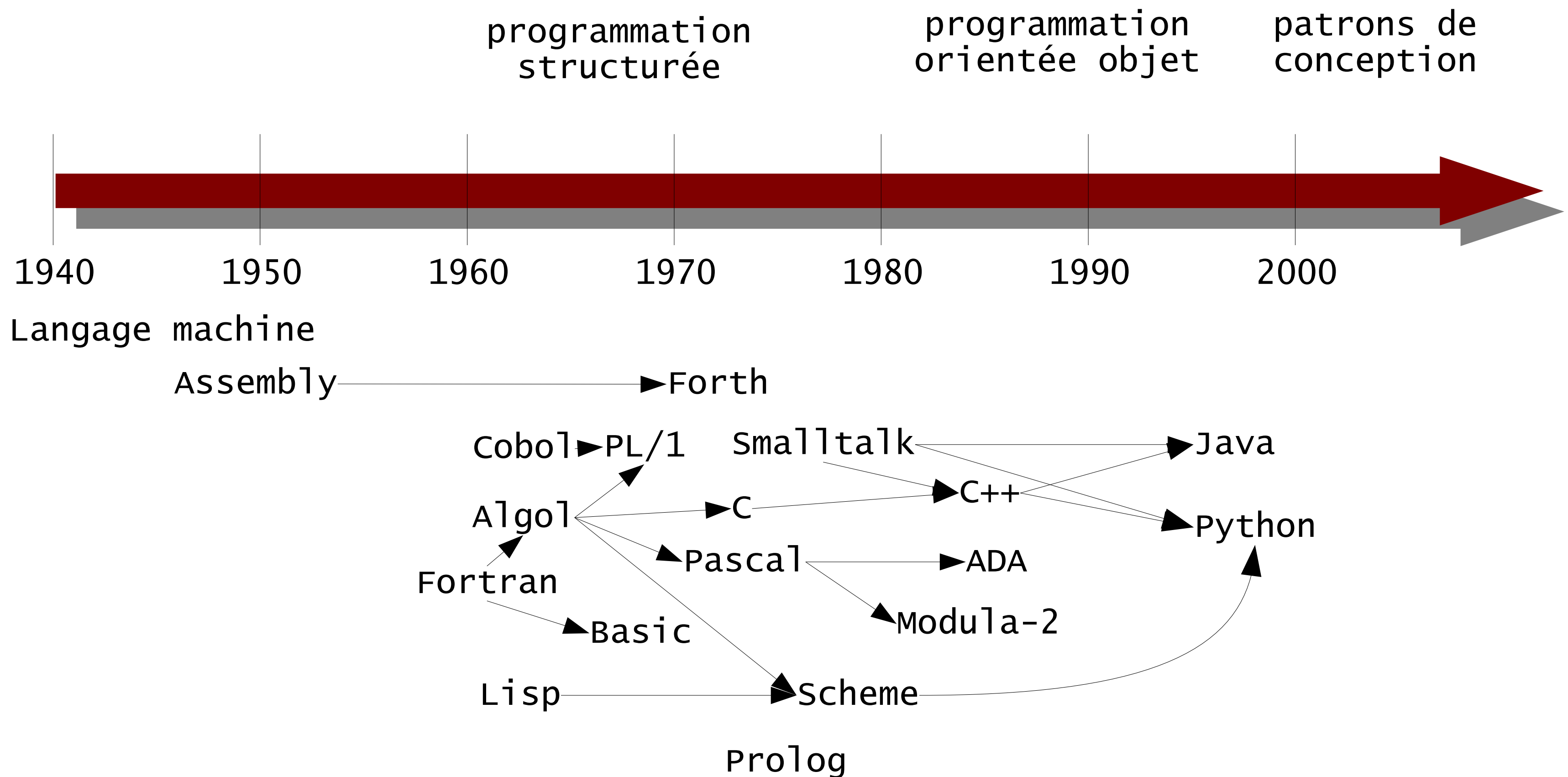
Plan

- Introduction
- Encapsulation
- Types de données abstraits
- Héritage et polymorphisme
- Programmation OO en langage Python
- Éléments d'UML
- Patrons de conception orientés objet

Origines

- La programmation orientée objet surgit dans les années '80
- Logiciels de plus en plus complexes, travail en équipe
- Contrôler la complexité des logiciels
- Code réutilisable pour contenir les coûts du développement
- Contenir les coûts de la maintenance :
  - Ajout de nouvelles fonctionnalités
  - Modification de fonctionnalités existantes
  - *portage* sur d'autres plate-formes ou environnements
- Coordonner et répartir le travail de développement
- Modularité

Evolution de la programmation



Programmation orientée objet

Discipline de programmation dans laquelle le programmeur établit

- non seulement les structures de données,
- mais aussi les opérations qui peuvent leurs être appliquées.

Ainsi,

- la structure de données devient un **objet** qui inclut
  - Données, appelées **attributs**
  - Opérations, appelées **méthodes**
- Le programmeur peut définir des **relations** entre les objets

Encapsulation

- Seules les opérations définies à l'intérieur d'une structure de données peuvent manipuler les données de cette structure :
  - L'utilisation d'opérations certifiées sur les structures de données garantit leur consistence
  - Élimine une parmi les causes d'erreur les plus importantes

- Cacher les détails de réalisation :
  - Avère la modularité du code
  - facilite l'extension et la maintenance
  - facilite l'individuation des erreurs

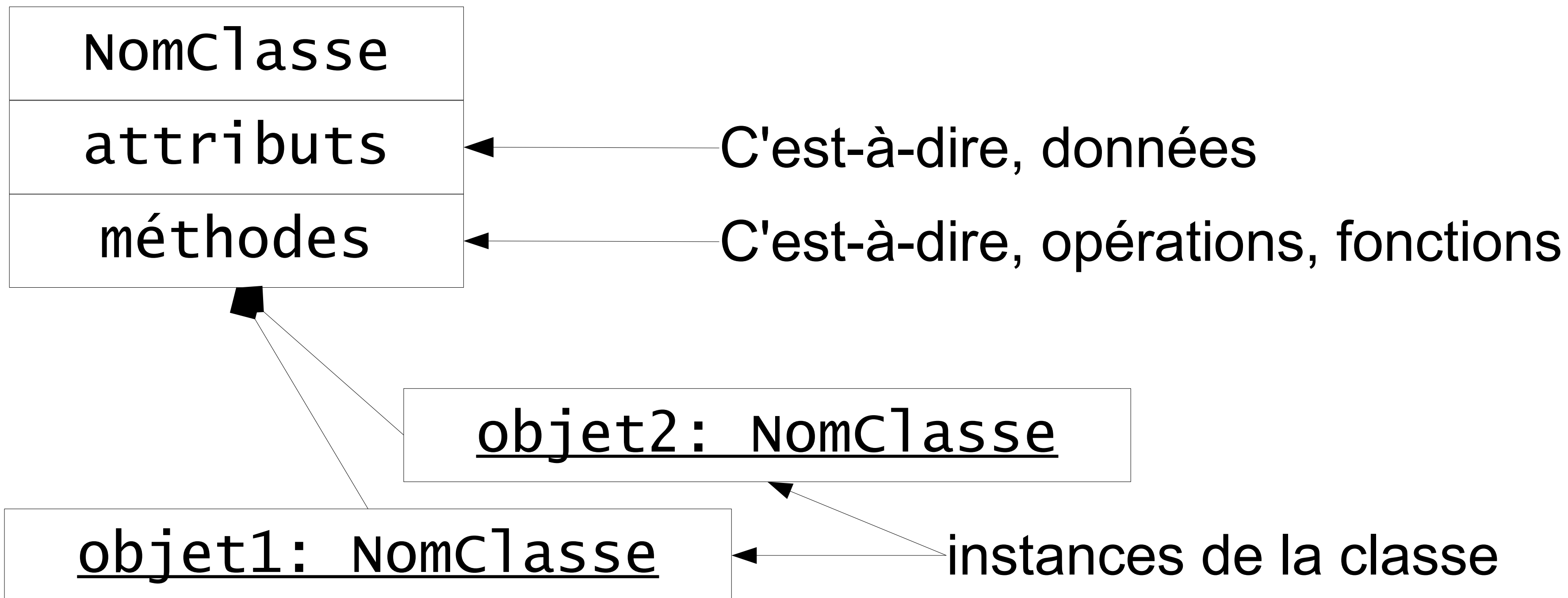
Type abstrait de données

- Possède un ***type***
- Définit un ensemble d'***opérations***, qui constituent son ***interface***
- Les opérations de l'interface sont *la seule manière d'accéder* au type abstrait de données
- Son domaine d'application est défini par des ***axiomes*** et des ***préconditions***

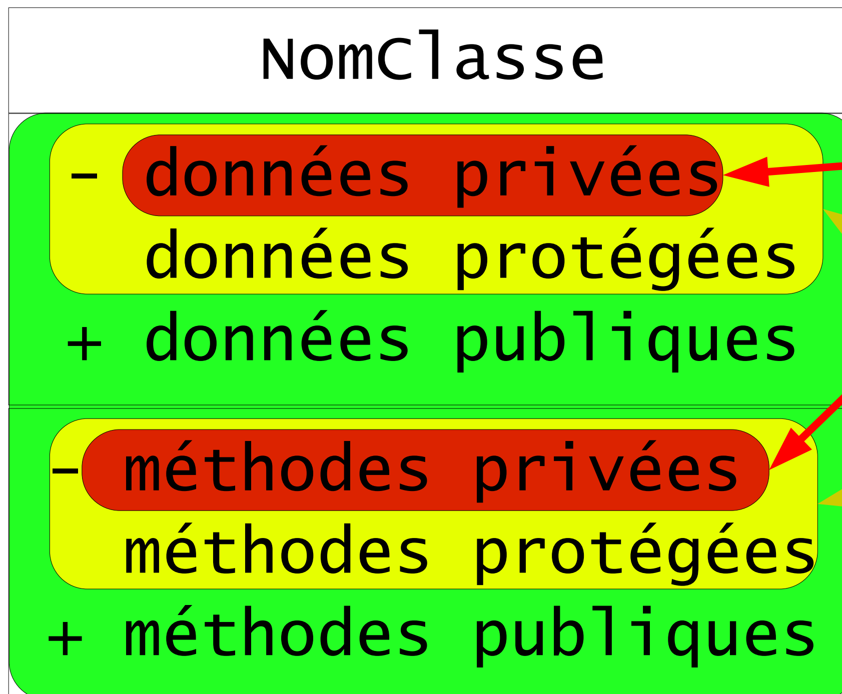
Exemple : Type abstrait de données “Nombre Naturel”

- **Type :** “Nombre Naturel”
- Opérations définies (**Interface**) :
  - $S(n)$, $n + m$, $n \times m$, etc.
  - $n = m$, $n < m$, $n \mid m$, premier(n), etc.
- **Axiomes et préconditions :**
  - $n + 0 = 0 + n = n$
  - $n + S(m) = S(n) + m$
  - $n \times 0 = 0 \times n = 0$
  - $n \times S(m) = (n \times m) + n$
  - *etc.*

Objets et Classes



Visibilité



visibles que au code de la classe

visibles que au code de la classe et des sous-classes

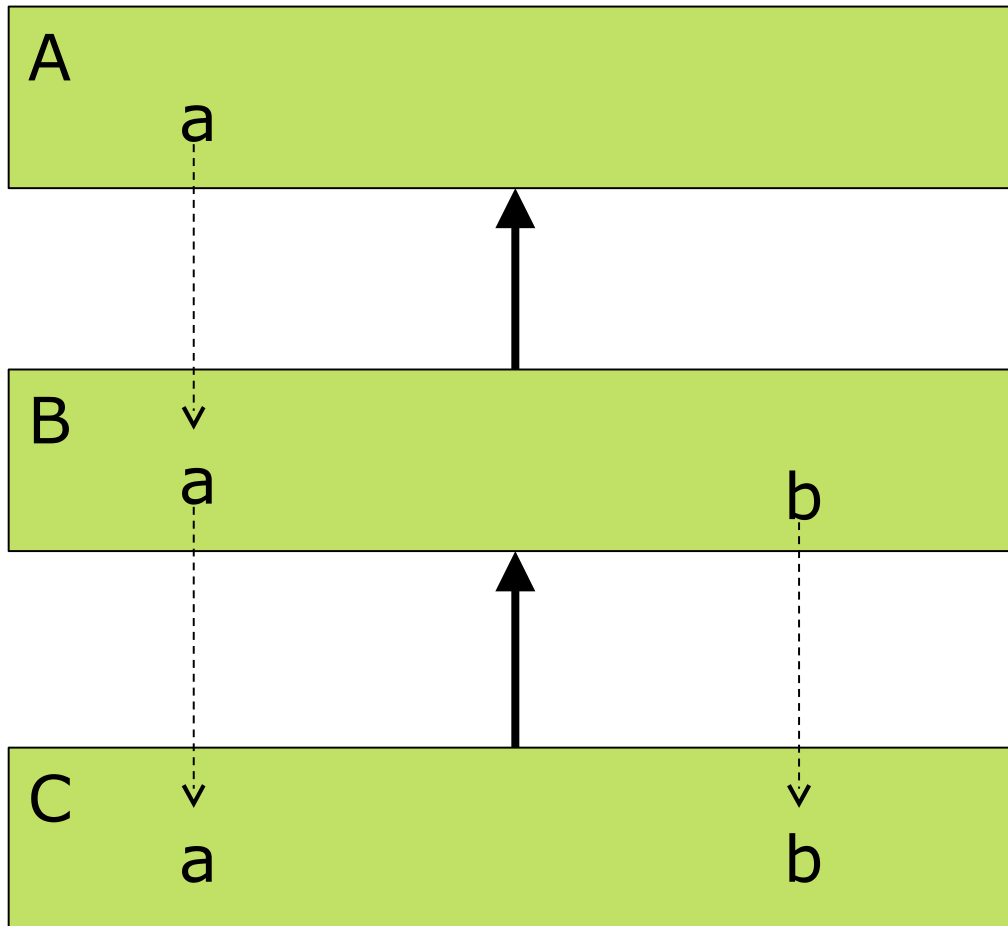
Héritage : Définition

L'**héritage** est la caractéristique d'un langage orienté objet qui fait que les objets d'une classe “héritent” toutes les propriétés définies pour les classes de niveau supérieur :

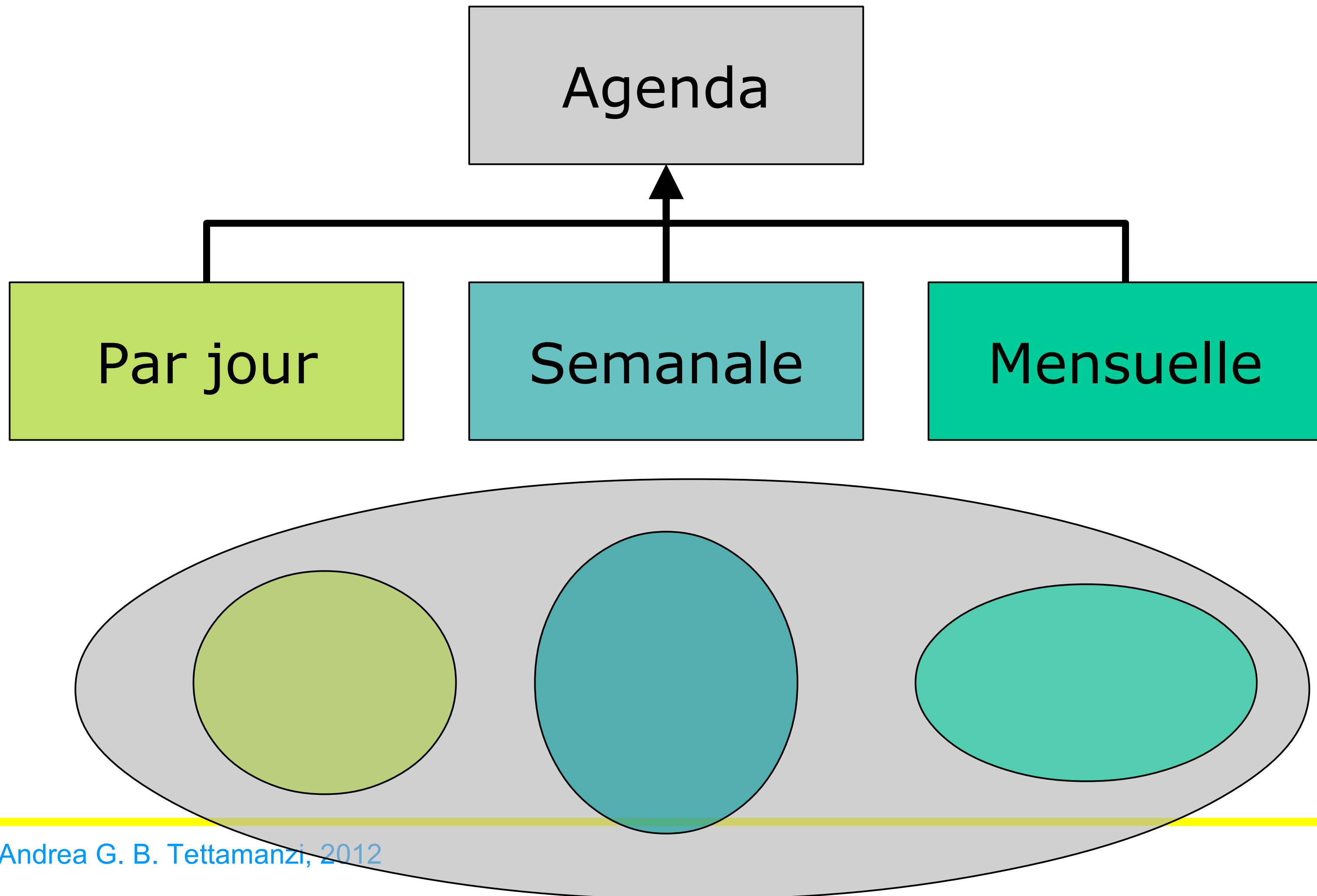
- attributs;
- méthodes;
- etc.

Historiquement, c'est une des caractéristiques les plus controversées.

Héritage



Vision naïve-intuitive



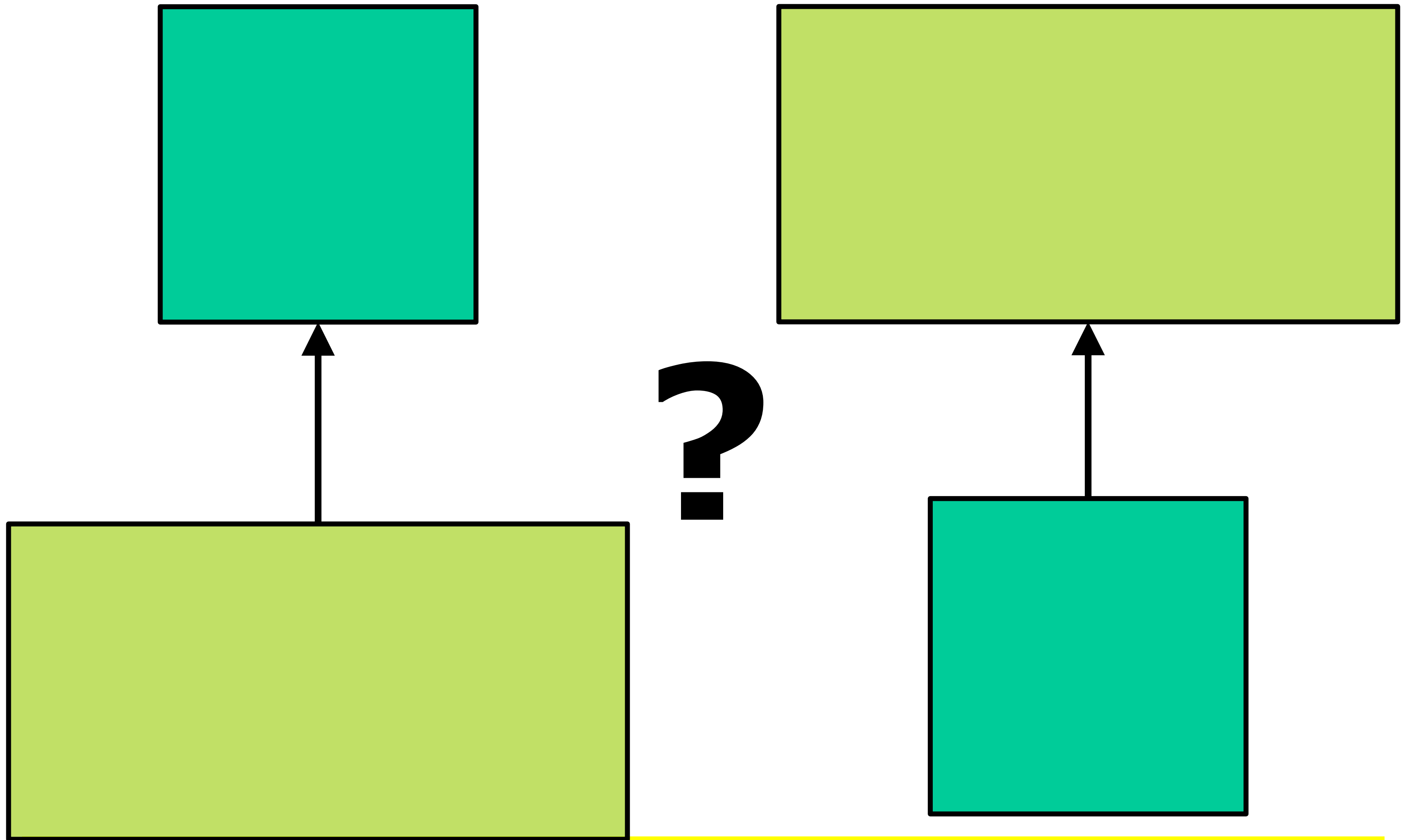
Principe de substitution de Liskov

Sous-classe = sous-type

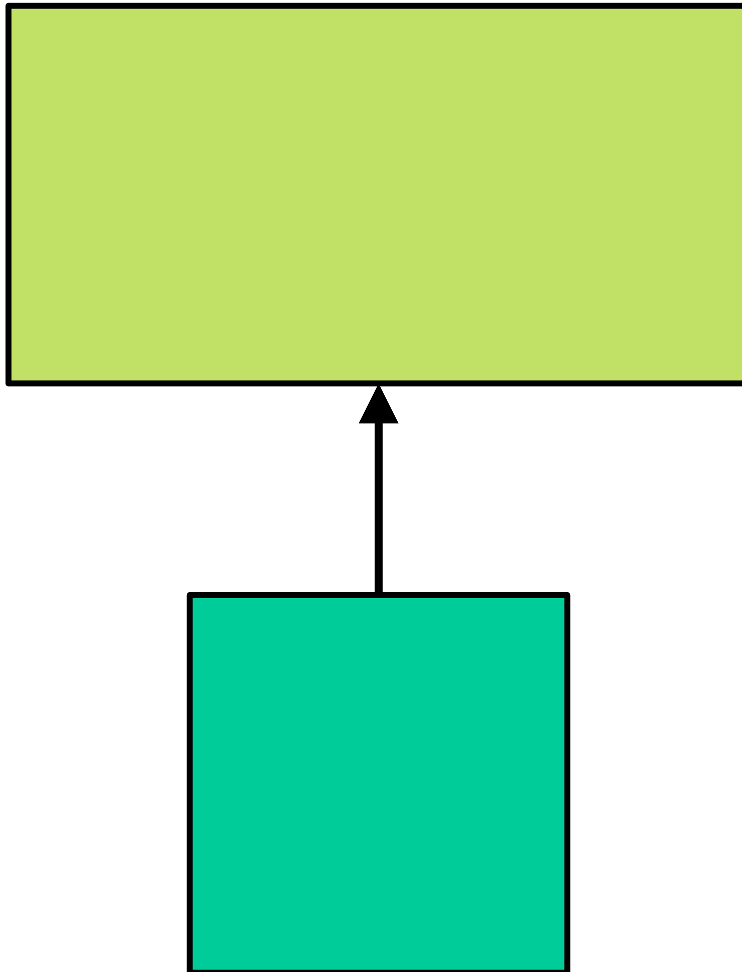
“B sous-type de A ⇔ pour chaque programme qui utilise des objets de classe A, on peut utiliser des objets de classe B à leur place sans que le comportement logique du programme change”.

Une sous-classe ne peut pas contraindre le comportement des super-classes.

Exemple



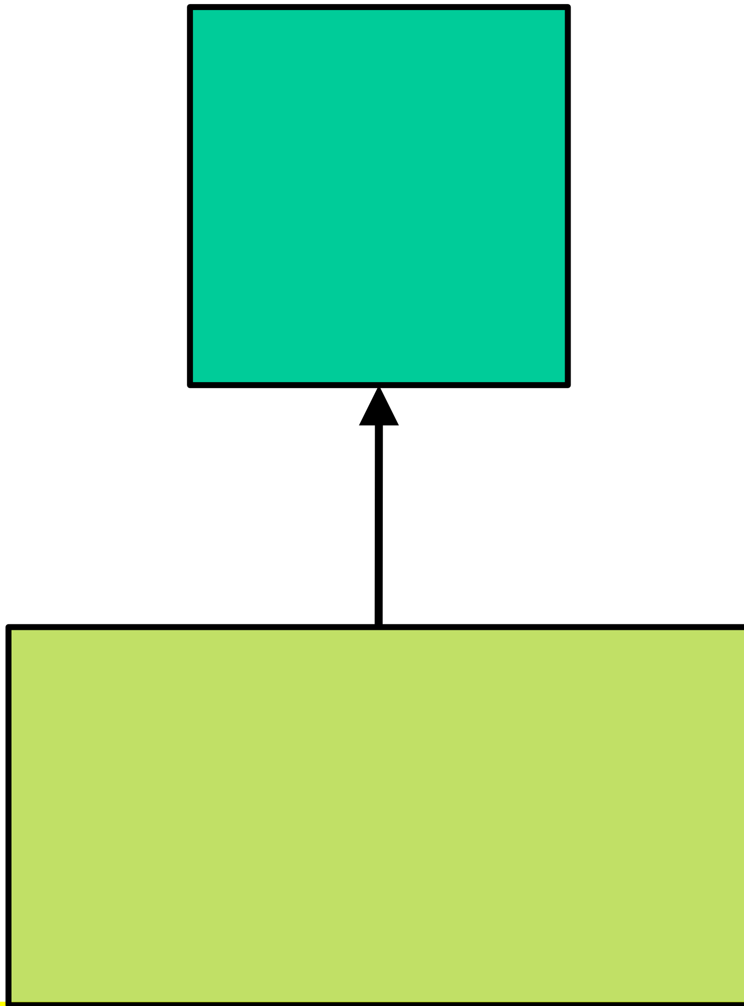
Exemple



protected double b, h;

setBase(double);
setHeight(double);

Exemple



protected double b;

setBase(double);

protected double h;

setHeight(double);

Types d'héritage

Deux notions cohabitent dans l'héritage :

– Héritage d'interface ou *subtyping*;

– Héritage de réalisation ou *subclassing*.

Héritage de réalisation

Il s'agit d'un mécanisme de réutilisation du code.

La sous-classe réutilise les méthodes des super-classes.

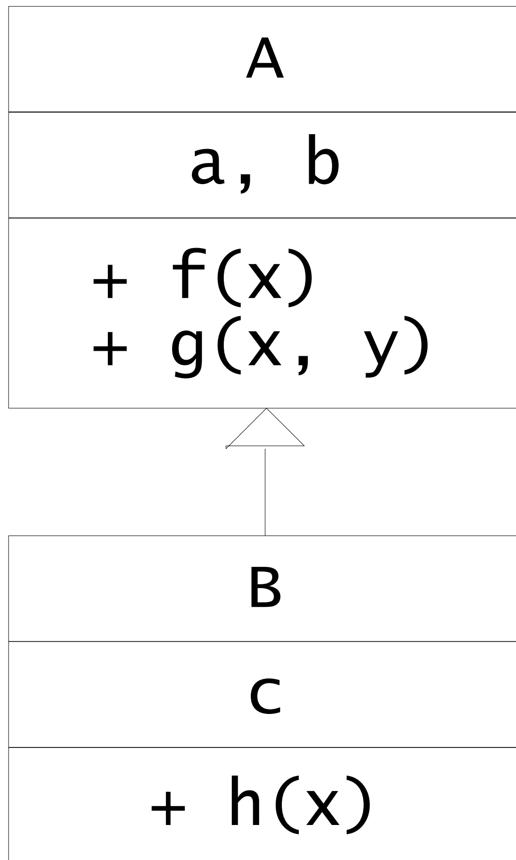
Héritage d'interface

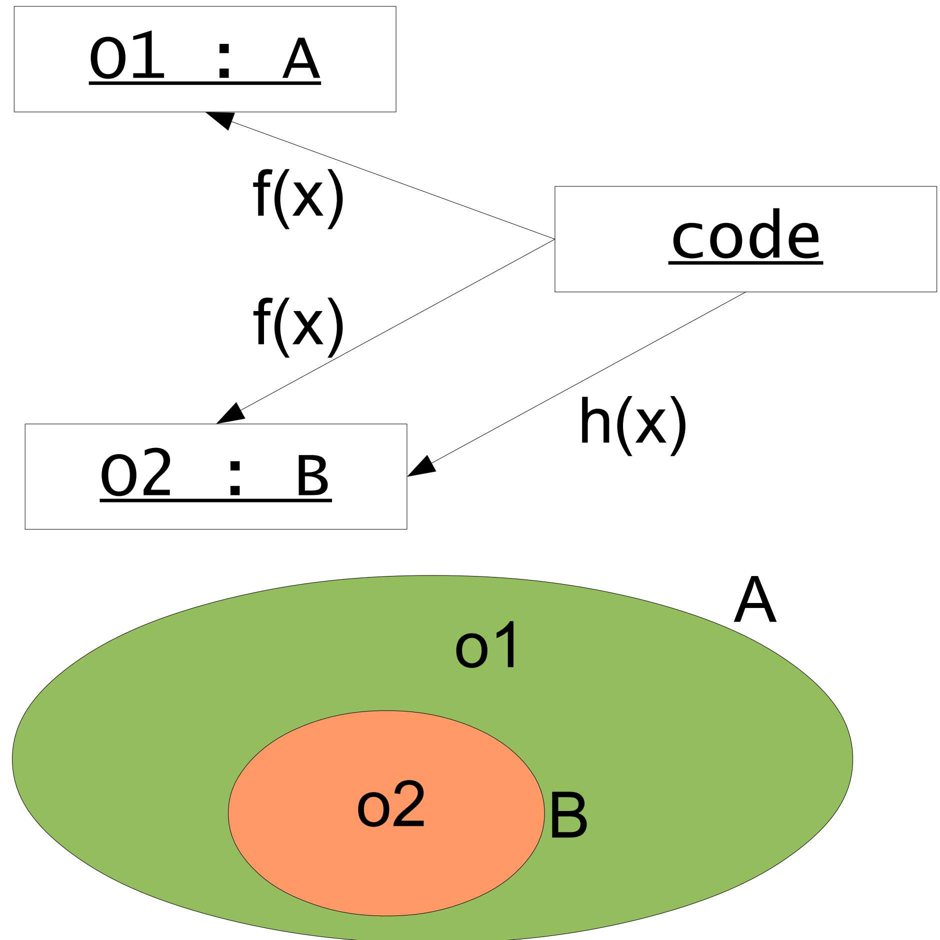
Il s'agit d'un mécanisme de compatibilité entre des types.

Il permet le polymorphisme par sous-typage.

Une sous-classe est un sous-type compatible vers le haut avec tous les types définis tout au long de sa chaîne d'héritage.

Héritage





Héritage multiple

Deux ordres d'avantages :

- fusion d'interfaces provenant de sources différentes ;
- réutilisation de code provenant de sources différentes.

Héritage d'interface :

- La correction peut être vérifiée statiquement (= au moment de la compilation);

Héritage de réalisation :

- Plus problématique : ex., la même méthode réalisée (comment ?) en deux super-classes distinguées.

Polymorphisme

Proprieté d'une même entité qui se manifeste dans des formes différentes dans de contextes différents…

In Informatique :

Idée d'autoriser le même code à être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.

Exemple : monomorphisme

int **max**(int *a,* int *b*) :

    if(*a* > *b*) : return *a*

    else : return *b*

double **maxd**(double *a,* double *b*) :

    if(*a* > *b*) : return *a*

    else : return *b*

Exemple : monomorphisme

int **max**(int *a,* int *b*) :

if(*a* > *b*) : return *a*

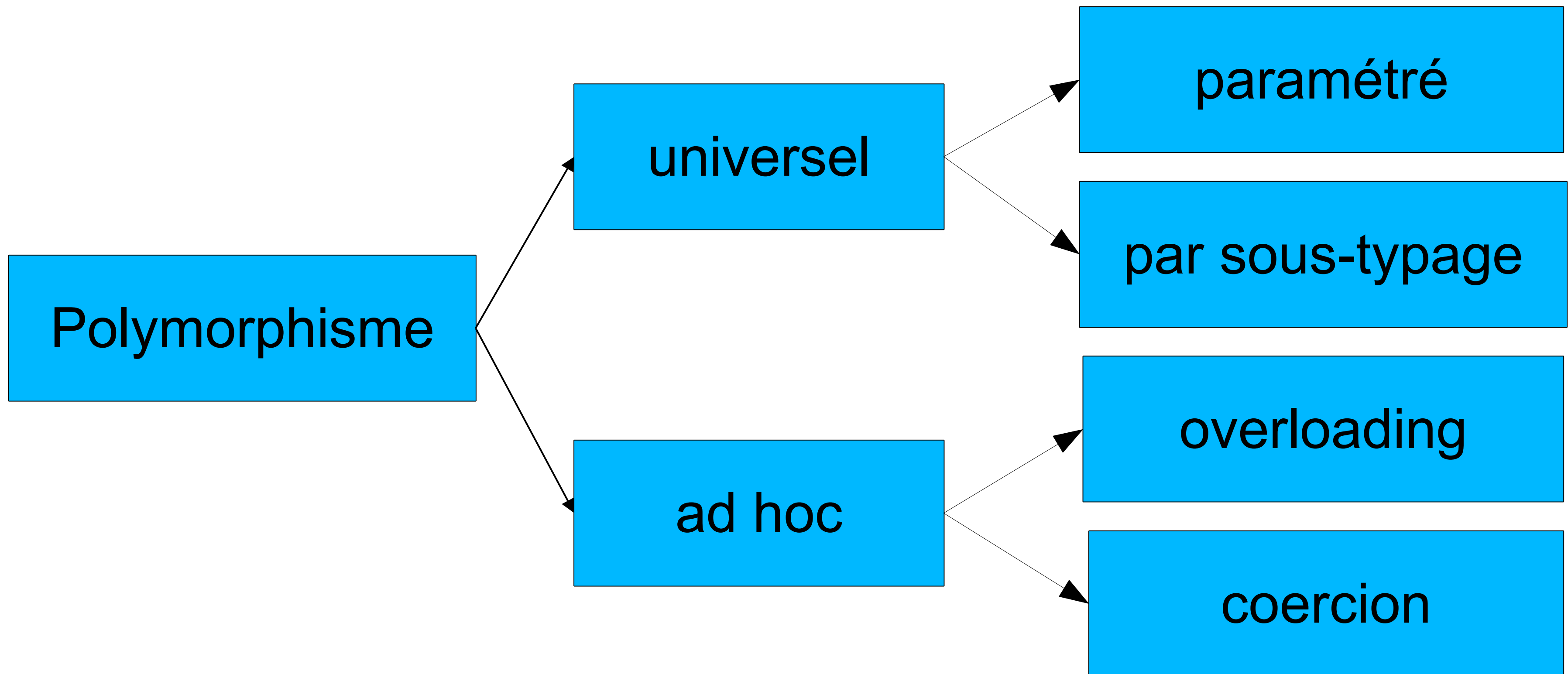
else : return *b*

double **maxd**(double *a,* double *b*) :

if(*a* > *b*) : return *a*

else : return *b*

Classification selon Cardelli-Wegner



Overloading

La même fonction ou les mêmes opérateurs peuvent être appliqués à des types différents.

Exemple (en C):

double *x*, *y*, *z*;
int *i*, *j*, *k*;

z = *x* + *y*;
k = *i* + *j*;

Coercion

Les arguments d'une fonction ou opérateur sont transformés implicitement en le type applicable.

Exemple (en C):

double pow(double, double);

…

double *x*, *y*;

int *n*;

y = pow(*x*, *n*);

Polymorphisme paramétré

Fonctions et opérateurs paramétrés selon le type auquel il peuvent être appliqués.

Exemple (en C++):

```
template <typename T> T max(T a, T b)
{
    if(a > b) return a;
    else return b;
}
```

Typique de la "programmation générique".

Polymorphisme par sous-typage

- Une méthode (c'est-à-dire une opération) peut être appliquée à tous les objets qui appartiennent à la classe qui la prévoit dans son interface.
- Le principe de substitution de Liskov s'applique
- Typique de la programmation orientée objet

Exemple : max

```
interface Comparable
{
    int compareTo(Comparable c);
}



static Comparable max(Comparable a, Comparable b)
{
    if(a.compareTo(b) > 0) return a;
    else return b;
}
```

Programmation OO en Python

- Les objets sont l'abstraction des données en Python.
- Toutes les données dans un programme Python sont représentés par des objets, y compris le code.
- Un objet possède :
  - Une identité (= son adresse en mémoire) : id()
  - Un type (opérations supportées + valeurs possibles)
  - Une valeur (mutable ou immutable, selon le type)
- Chaque classe et instance de classe possède un espace de noms (*namespace*), réalisé par un dictionnaire.
- Python n'adhère pas au fait de protéger le code vis-à-vis du programmeur. Python encapsule les objets comme un *namespace* unique mais c'est une encapsulation transparente

Définition d'une classe

nom de la classe (identificateur)

superclasses desquelles las classe hérite

```
class NomCls(cl1,cl2,cl3):
    """documentation"""
    # bloc instructions, définition des méthodes, etc.
    def __init__(self,arg1,arg2):
        """documentation"""
        # initialisation d'une instance.
        self.arg1 = arg1
        self.arg2 = arg2
```

Pas de définition explicite d'interface, on utilise les classes pour cela

Création et utilisation d'une instance

```
instance = NomCls(arg1,arg2)

instance.méthode(x,y)

instance.attribut

NomCls.attribut_de_classe
```

UML

Le langage universel de modélisation (Unified Modeling Language, UML) est une famille de notations graphiques qui aide à décrire et concevoir des logiciels, notamment ceux construits en utilisant un style orienté objet

Ingrédients de UML

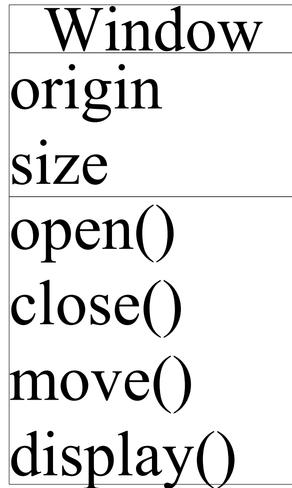
- Le “vocabulaire” de UML inclut trois types d'ingrédients :
  - Entités (choses, *things*): abstractions pertinentes
  - Relations : lient des entités entre elles
  - Diagrammes : regroupent ensembles intéressants d'entités

Types d'entités

- Structurales :
  - Classes, interfaces, collaborations, cas d'utilisation, composantes, nœuds
- Comportementales :
  - Messages, états
- De regroupement :
  - Paquets
- Annotations :
  - Notes

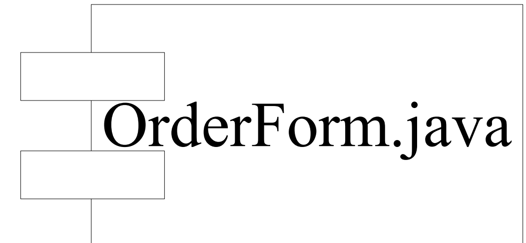
Entités structurales

Classes

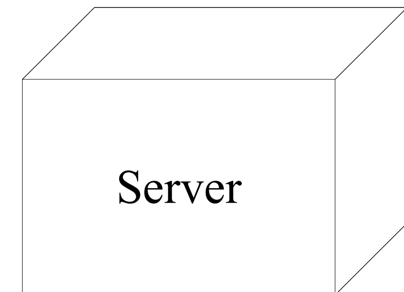


Collaborations

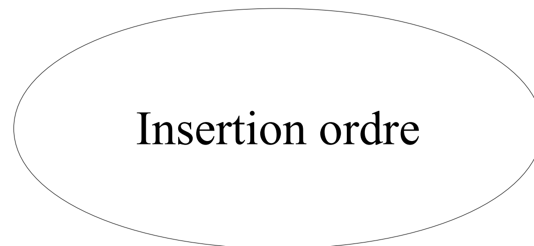
Composantes



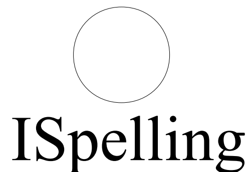
Noeuds



Cas d'utilistion



Interfaces


ISpelling

Entités comportementales

Entités de regroupement

Paquets

Entités d'annotation

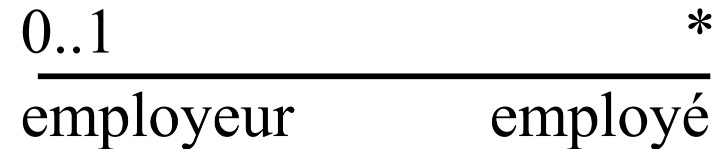
Note

Revient avec une référence au client stub de l'objet distant

Types de Relations

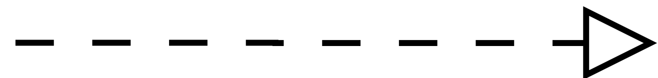
Dépendences

Associations



Généralisations

Réalisations



Diagrammes

- Il existe 13 types distingués de diagrammes UML :
  - Activité
  - Cas d'utilisation
  - Classes
  - Communication
  - Composantes
  - Déploiement
  - Interaction
  - Machines à états
  - Objets
  - Paquets
  - Séquence
  - Structure des composantes
  - Temporisation

Patrons de conception

La programmation orientée objet toute seule ne suffit pas.

Idée de “patron de conception” = schema.

Aggrégations de plusieurs classes logiquement liées.

Conception orientée objet (1)

Concevoir un programme orienté objet réutilisable signifie :

- Identifier des objets pertinents ;
- Les regrouper dans des classes de la juste granularité ;
- Définir leurs interfaces ;
- Définir la hiérarchie des classes et des interfaces ;
- Établir des relations signifiantes entre elles ;
- Répondre aux spécificités du problème ;
- Se maintenir assez généraux pour pouvoir traiter d'autres problèmes du même type.

Conception orientée objet (2)

Difficilement un projet « réussit bien » au premier essai :

- Il va être recyclé dans des contextes différents ;
- Il va être modifié à chaque fois pour repondre à des besoins différents ;
- Il pourra être amélioré ;
- Finalement, la meilleure solution sera trouvée.

Expertise de conception

La différence entre un concepteur débutant et un expert :

- Parcours par essais et erreurs ;
- répertoire de solutions déjà conçues ;
- Aptitude à trouver la solution la plus appropriée.

Autrement dit, le concepteur expert possède un répertoire de schémas de conception prouvés qu'il peut adapter à des nouvelles situations.

Ceci est le sens de patron de conception.

Patron de conception

Un patron de conception décrit :

- Un problème que l'on retrouve souvent en écrivant des programmes ;
- Le noyau de sa solution.

Structure d'un patron

Un patron de conception est constitué par :

- Son nom;
- La description du problème, du contexte d'application ;
- La solution :
  - Éléments (classes et objets) constitutifs ;
  - Les relations entre les éléments ;
- Les conséquences de son application :
  - résultats;
  - Avantages/désavantages.

Exemples de patrons

Des patrons sont la réédition orientée objet de techniques de programmation très anciennes :

ex. interprète.

D'autres patrons peuvent servir comme des briques de base pour construire des architectures logicielles versatiles et performantes :

ex. adaptateur, itérateur, etc.

Pour approfondir

Il existe des ouvrages qui examinent les patrons de conception les plus importants. Par exemple :

Gamma, Helm, Johnson, Vlissides

Design Patterns: Elements of reusable object-oriented software

Addison-Wesley, 1995

Merci de votre attention