

FSAB1402: Informatique 2

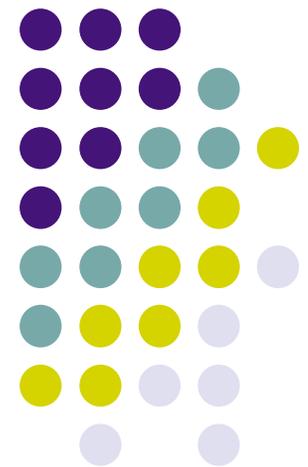
Objets, Classes, Polymorphisme et Héritage



Département d'Ingénierie Informatique, UCL

Peter Van Roy

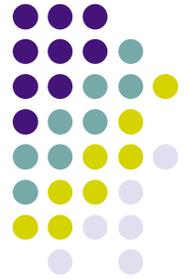
pvr@info.ucl.ac.be



Ce qu'on va voir aujourd'hui



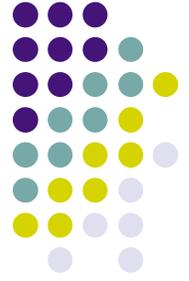
- Résumé du dernier cours
- Les objets et les classes
- Le polymorphisme
 - Le principe de la concentration des responsabilités
- L'héritage
 - Le principe de substitution
 - Lien statique et lien dynamique
 - L'héritage simple et l'héritage multiple



Lecture pour cette séance

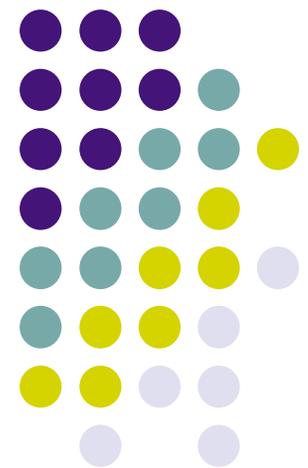
- Chapitre 1 (sections 1.12-1.13):
objets et classes
- Chapitre 5 (section 5.5):
le polymorphisme
- Chapitre 6 (section 6.1):
l'héritage, idées de base et motivation
- Chapitre 6 (sections 6.2):
la classe
- Chapitre 6 (sections 6.3.1-6.3.3):
la classe et l'héritage
- Chapitre 6 (section 6.4.1):
le principe de substitution
- Chapitre 6 (sections 6.4.2-6.4.3):
l'héritage multiple

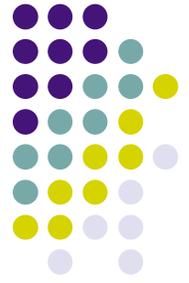
Annonce



- L'échéance pour le mini-projet
 - Le 30 novembre à 18h00
 - Chaque groupe envoie les deux fichiers à l'assistant correspondant (**pas** à staff1402.fsa!) et s'inscrit pour un entretien sur le bureau de l'assistant

Résumé du dernier cours





Les collections indexées

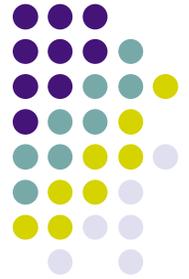
- Les **tuples** et les **enregistrements** sont utiles quand il faut garantir que la valeur ne change pas
 - On peut les utiliser dans le modèle déclaratif mais aussi dans le modèle avec état
- Les **tableaux** et les **dictionnaires** sont utiles pour calculer une collection “incrémentalement”: en petits bouts
 - Le dictionnaire est particulièrement intéressant, à cause de sa souplesse et son efficacité
 - Le dictionnaire est une abstraction qui mérite un regard particulier: il a une interface simple et une implémentation sophistiquée

Comparer le déclaratif avec l'état (pour petits programmes)



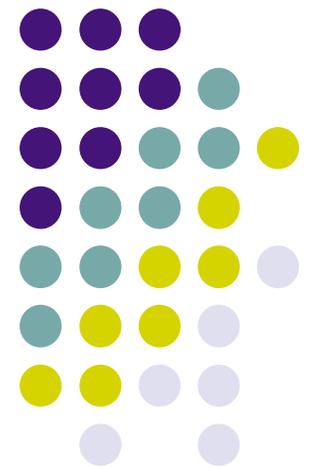
- Nous avons comparé les deux modèles en implémentant des algorithmes sur les matrices
 - Attention: c'est une comparaison “in the small” (pour petits programmes, c'est-à-dire, le codage des algorithmes)
- La complexité des programmes est comparable
 - La complexité dépend surtout de la représentation des données qu'on choisit, pas du modèle (par exemple, “liste de listes” est nettement plus compliquée que “tuple de tuples” ou “tableau de tableaux”)
- Le choix du modèle dépend de ce qu'on veut faire
 - Le modèle avec état est bien quand on construit une collection en petits bouts (incrémentalement)
 - Le modèle déclaratif est bien quand on fait un système multi-agent (à voir plus tard dans le cours)

Comparer le déclaratif avec l'état (pour grands programmes)

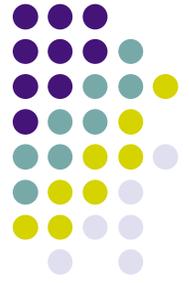


- Nous pouvons aussi comparer les deux modèles “**in the large**” (pour grands programmes)
- Un grand programme est construit comme un ensemble d'abstractions de données, souvent écrit par une équipe
 - Chaque abstraction peut être déclaratif ou avec état
 - Une abstraction déclarative ne change jamais son comportement → bon pour **l'exactitude**
 - Une abstraction avec état peut “apprendre” ou “mémoriser” le passé → bon pour **la modularité**
 - Il est souhaitable de combiner les deux pour avoir le meilleur des deux modèles (comme pour la mémoisation)

Les objets et les classes

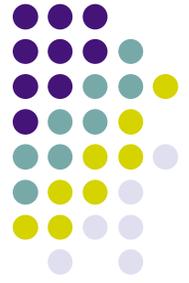


Programmation avec objets



- Le concept **d'objet** est devenu omniprésent dans l'informatique aujourd'hui
 - Une abstraction de données qui contient à la fois la valeur et les opérations
 - Introduit en Simula 67, disséminé partout via Smalltalk et C++
- Avantages
 - L'abstraction de données
 - Le polymorphisme
 - L'héritage
- Les types abstraits sont tout aussi omniprésents!
 - Il est important de bien comprendre les objets et types abstraits purs et de voir comment ils sont mélangés dans chaque langage. Par exemple, un "**objet Java**" est un mélange d'un "**objet pur**" et d'un "**type abstrait pur**" comme nous allons voir la prochaine fois.

Schéma général d'un objet



declare

local

A1={NewCell I1}

...

An={NewCell In}

in

proc {M1 ...} ... **end**

...

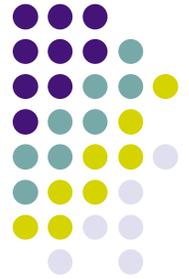
proc {Mm ...} ... **end**

end

Ce fragment crée des nouvelles cellules locales A1, ..., An, qui ne sont visibles que dans les procédures globales M1, ..., Mm.

On appelle souvent A1, ..., An des “attributs” et M1, ..., Mm des “méthodes”

Remarquez que A1, ..., An sont **cachés** de l'extérieur et M1, ..., Mm sont **visibles** de l'extérieur (souvenez-vous de la définition d'une abstraction de données!)



Exemple: un objet “compteur”

declare

local

A1={NewCell 0}

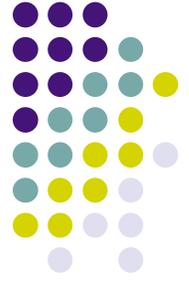
in

proc {Inc} A1:=@A1+1 **end**

proc {Get X} X=@A1 **end**

end

Le compteur avec envoi procédural



```
declare
local
  A1={NewCell 0}
  proc {Inc} A1:=@A1+1 end
  proc {Get X} X=@A1 end
in
  proc {Counter M}
    case M of inc then {Inc}
    [] get(X) then {Get X}
    end
  end
end
```

Toutes les méthodes sont invoquées par l'intermédiaire d'un seul point d'entrée: la procédure Counter:

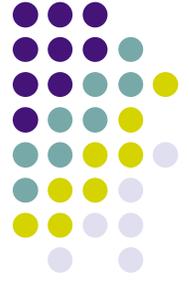
{Counter inc}

{Counter inc}

{Counter get(X)}

L'argument de Counter est appelé un "message"

Une usine pour créer des compteurs



declare

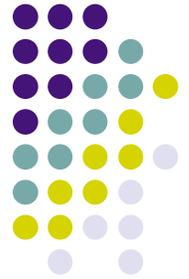
```
fun {NewCounter}
  A1={NewCell 0}
  proc {Inc} A1:=@A1+1 end
  proc {Get X} X=@A1 end
in
  proc {$ M}
    case M of inc then {Inc}
    [] get(X) then {Get X}
    end
  end
end
```

Il est souvent intéressant de créer plusieurs objets avec les mêmes opérations mais avec des états différents

On peut définir une fonction qui, quand on l'appelle, crée un nouvel objet à chaque fois

L'appel `C={NewCounter}` crée l'attribut `A1` et rend un objet avec méthodes `Inc` et `Get`

L'utilisation de la fonction NewCounter



C1={NewCounter}

C2={NewCounter}

{C1 inc}

{C1 inc}

local X in {C1 get(X)} {Browse X} **end**

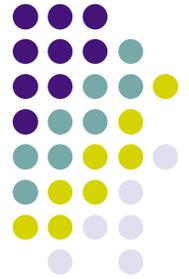
local X in {C2 get(X)} {Browse X} **end**



Syntaxe pour une classe

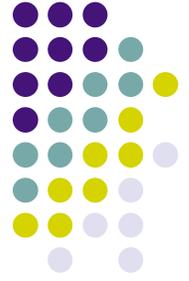
```
class Counter  
  attr a1  
  meth init a1:=0 end  
  meth inc a1:=@a1+1 end  
  meth get(X) X=@a1 end  
end
```

```
C1={New Counter init}  
{C1 inc}  
local X in {C1 get(X)} {Browse X} end
```



C'est quoi une classe? (1)

- La classe **Counter** est passée comme argument à la fonction New:
 - `C={New Counter Init}`
 - La classe **Counter** est **une valeur** (analogue à une procédure)!
 - La définition de la classe et la création de l'objet sont séparées
 - (Pour les futés: les classes forment un type abstrait!)
- La fonction NewCounter fait les deux choses en même temps
 - Le résultat est le même



C'est quoi une classe? (2)

- Comment est-ce qu'on représente une classe comme une valeur?
- Une classe est un **enregistrement** qui regroupe les noms des attributs et les méthodes:

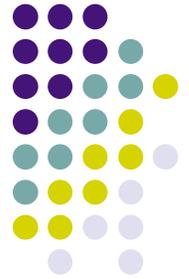
```
Counter=c(attrs:[a1]
           methods:m(init:Init inc:Inc get:Get))
```

- La fonction New prend l'enregistrement, crée les cellules et crée l'objet (une procédure qui référence les cellules et les méthodes):

```
fun {New Class Init}
  ....
end
```

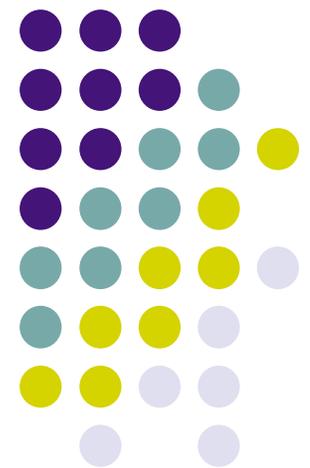
- Exercice: **lisez et comprenez la section 6.2.2** du livre, en particulier les figures 6.1, 6.2, 6.3

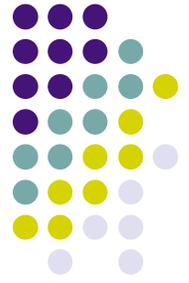
Schéma général d'une classe



```
class C  
  attr a1 ... an  
  meth m1 ... end  
  ...  
  meth mm ... end  
end
```

Le polymorphisme

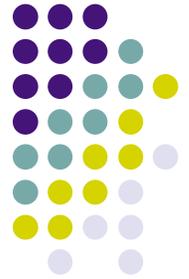




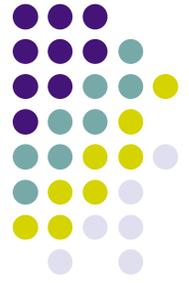
Le polymorphisme

- Dans le langage de tous les jours, une entité est **polymorphe** si elle peut prendre **des formes différentes**
- Dans le contexte de l'informatique, une opération est **polymorphe** si elle peut prendre **des arguments de types différents**
- Cette possibilité est importante pour que les responsabilités soient bien réparties sur les différentes parties d'un programme

Le principe de la concentration des responsabilités

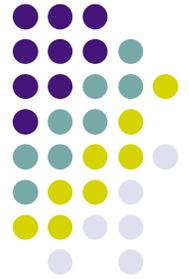


- Le polymorphisme permet d'isoler des responsabilités dans les parties du programme qui les concernent
 - En particulier, une responsabilité doit de préférence être concentrée dans une seule partie du programme
- Exemple: un patient malade va chez un médecin
 - Le patient ne devrait pas être médecin lui-même!
 - Le patient dit au médecin: “guérissez-moi”
 - Le médecin fait ce qu'il faut selon sa spécialité
- Le programme “guérir d'une maladie” est **polymorphe**: il marche avec toutes sortes de médecins
 - Le médecin est un argument du programme
 - Tous les médecins comprennent le message “guérissez-moi”



Réaliser le polymorphisme

- Toutes les abstractions de données que nous avons vues soutiennent le polymorphisme
 - Les objets et les types abstraits
 - C'est particulièrement simple avec les objets
 - Une des raisons du succès des objets
 - Pour ne pas surcharger le cours, on ne parlera que des objets
- L'idée est simple: si un programme marche avec une abstraction de données, il pourrait marcher avec une autre, **si l'autre a la même interface**



Exemple de polymorphisme

```
class Figure
  ...
end
class Circle
  attr x y r
  meth draw ... end
  ...
end
class Line
  attr x1 y1 x2 y2
  meth draw ... end
  ...
end
```

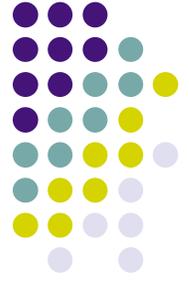
```
class CompoundFigure
  attr figlist
```

```
  meth draw
    for F in @figlist do
      {F draw}
    end
  end
```

```
  ...
end
```

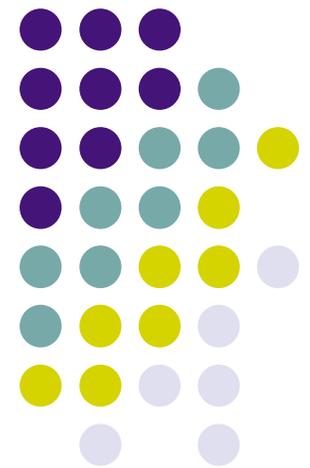
La définition de la méthode `draw` de `CompoundFigure` marche pour toutes les figures possibles: des cercles, des lignes et aussi d'autres `CompoundFigures`!
(Voir aussi l'exemple dans section 6.4.2)

Exécution correcte d'un programme polymorphe



- Quand est-ce qu'un programme polymorphe est correct?
 - Pour une exécution correcte, l'abstraction doit satisfaire à **certaines propriétés**
 - Le programme marchera alors avec **toute abstraction qui a ces propriétés**
 - Pour chaque abstraction, il faut donc **vérifier que sa spécification satisfait à ces propriétés**
- Pour l'exemple des médecins, le programme exige que le médecin guérirait le patient
 - Le polymorphisme marche si chaque médecin guérirait le patient
 - Chaque abstraction ("médecin") satisfait la même propriété ("guérira le patient")

L'héritage



Définition incrémentale des abstractions de données



- Des abstractions de données sont souvent très similaires
- Par exemple, la notion de “collection” d’éléments a beaucoup de variations
 - **Ensemble**: des éléments sans ordre défini
 - **Séquence**: un ensemble d’éléments dans un ordre
 - Séquence = ensemble + ordre
 - **Pile**: une séquence où l’on ajoute et enlève du même côté
 - Pile = séquence + contraintes sur ajout/enlèvement
 - **File**: une séquence où l’on ajoute d’un côté et enlève de l’autre côté
 - File = séquence + contraintes sur ajout/enlèvement



L'héritage et les classes

- Il peut être intéressant de définir des abstractions sans répéter les parties communes
 - Parties communes = code dupliqué
 - Si une partie est changée, toutes les parties doivent être changées
 - Source d'erreurs!
- L'héritage est une manière de définir des abstractions de façon incrémentale
 - Une définition A peut "hériter" d'une autre définition B
 - La définition A prend B comme base, avec éventuellement des modifications et des extensions
- La **définition incrémentale** A est appelée une **classe**
 - Attention: le résultat est une abstraction de données **complète**



L'héritage et la sémantique

- Une classe A est définie comme une transformation d'une autre classe B
- L'héritage peut être vu comme transformation **syntaxique**
 - On prend le code source de B et on le modifie
- L'héritage peut aussi être vu comme transformation **sémantique**
 - La définition de A est une fonction f_A avec $c_A = f_A(c_B)$
 - f_A prend une classe B comme entrée (la valeur c_B) et donne comme résultat une autre classe (la valeur c_A)



Dangers de l'héritage

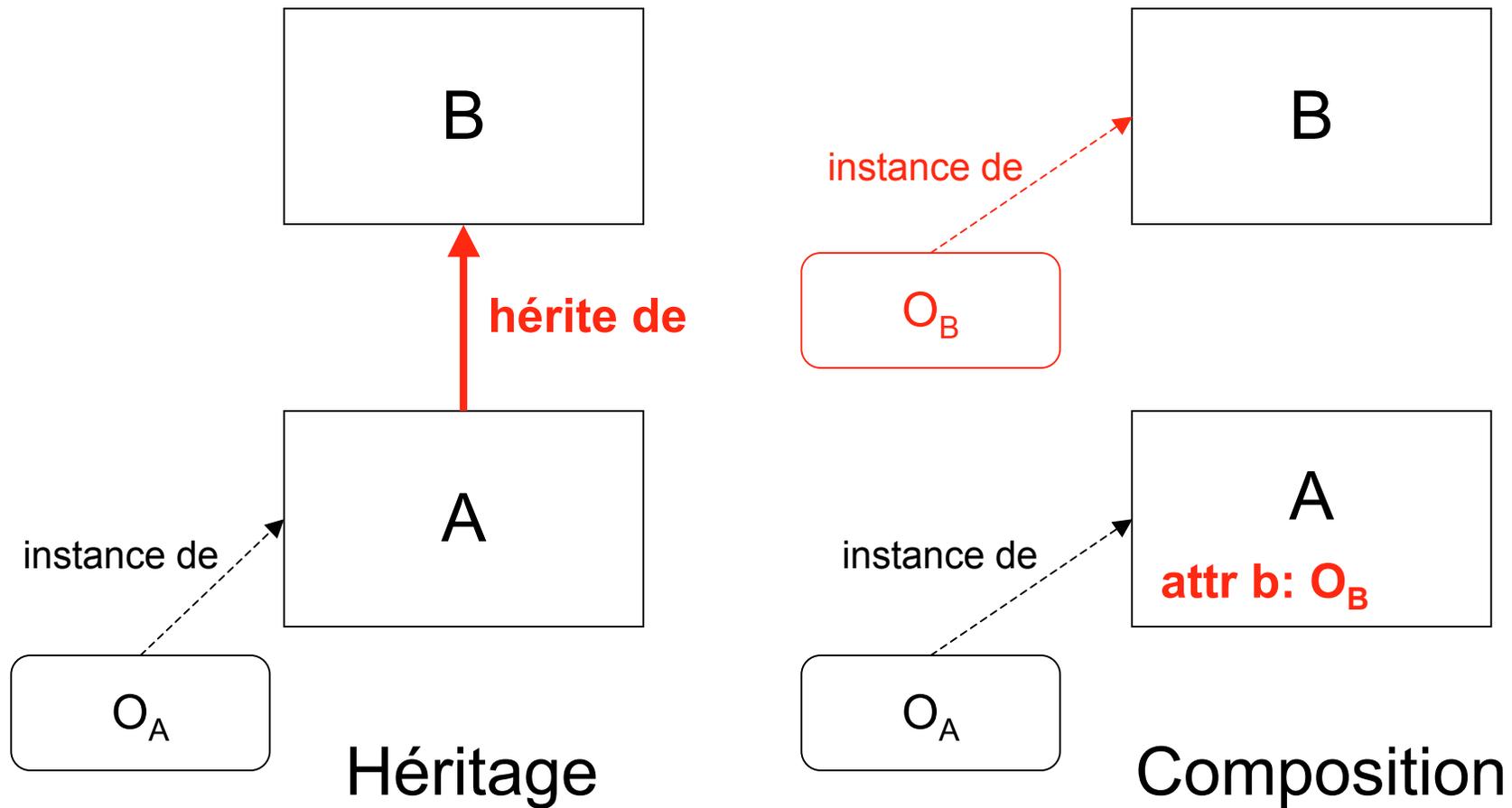
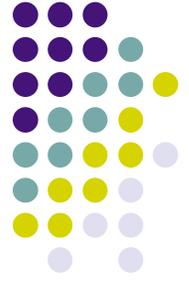
- L'héritage est parfois très utile, mais il faut l'utiliser avec beaucoup de précautions
- La possibilité d'étendre A avec l'héritage peut être vue comme **une autre interface à A**
 - Une autre manière d'interagir avec A
- Cette interface doit être maintenue pendant toute la vie de A
 - Une source supplémentaire d'erreurs!



Notre recommandation

- Nous recommandons d'utiliser l'héritage le moins possible
 - C'est dommage que l'héritage soit considéré comme tellement important par les mandarins de la programmation orientée objet
- Quand on définit une classe, nous recommandons de la prendre comme “**final**” (non-extensible par l'héritage) par défaut
- Nous recommandons d'utiliser **la composition** de préférence sur l'héritage
 - La composition = une classe peut dans son implémentation utiliser des objets d'autres classes (comme la liste de figures dans CompoundFigure)

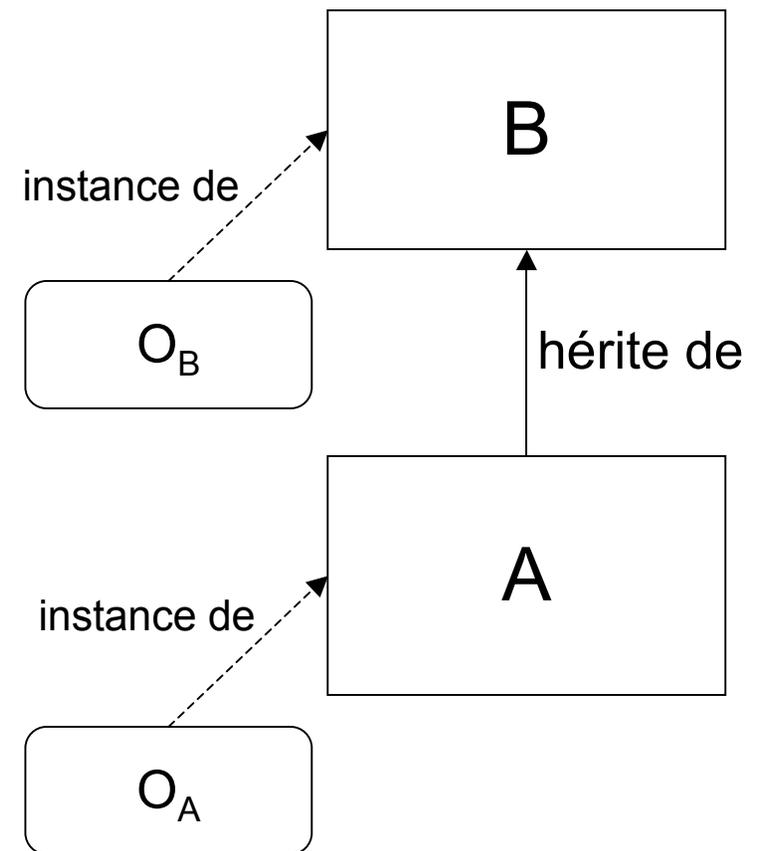
L'héritage versus la composition

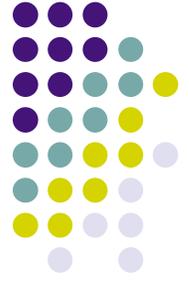




Le principe de substitution

- La bonne manière d'utiliser l'héritage
 - Supposons que la classe A hérite de B et qu'on a deux objets, O_A et O_B
- Toute procédure qui marche avec O_B doit marcher avec O_A
 - L'héritage ne doit rien casser!
 - A est une **extension conservatrice** de B

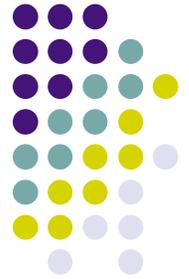




Exemple: classe Account

```
class Account
  attr balance:0
  meth transfer(Amount)
    balance := @balance+Amount
  end
  meth getBal(B)
    B=@balance
  end
end
A={New Account transfer(100)}
```

Extension conservatrice (respecte le princ. de subst.)



VerboseAccount:
Un compte qui affiche
toutes les transactions

```
class VerboseAccount
  from Account
  meth verboseTransfer(Amount)
  ...
end
end
```

La classe
VerboseAccount
a les méthodes
transfer, getBal et
verboseTransfer.

Extension non-conservatrice (ne respecte pas le pr. de sub.)

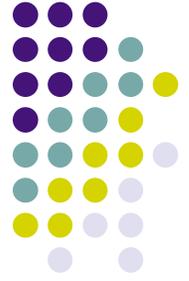


AccountWithFee:
Un compte avec des frais

```
class AccountWithFee
  from VerboseAccount
  attr fee:5
  meth transfer(Amount)
  ...
end
end
```

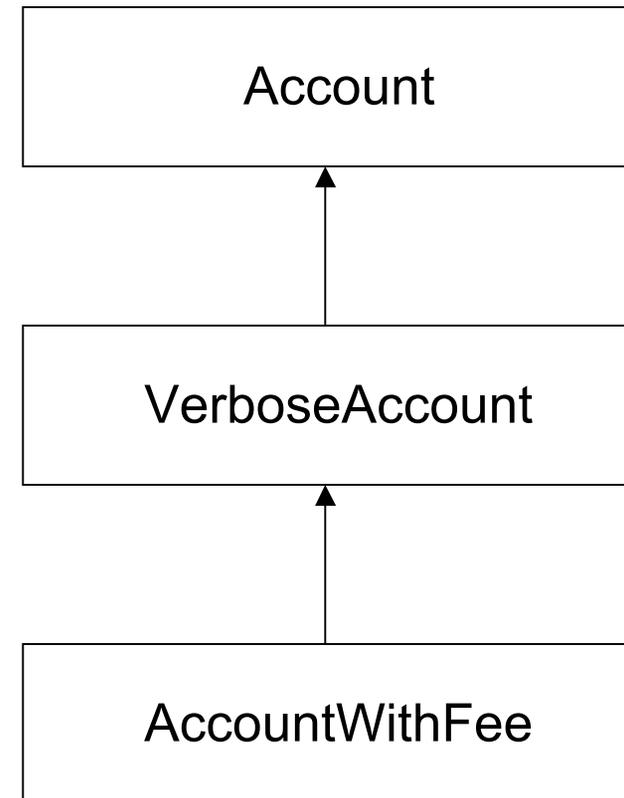
La classe AccountWithFee a les méthodes transfer, getBal et verboseTransfer. La méthode transfer a été **redéfinie**.

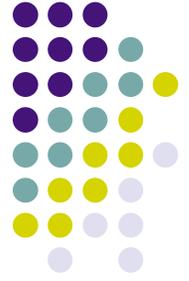
Hiérarchie de classe de l'exemple



```
class VerboseAccount
  from Account
  meth verboseTransfer(Amount)
  ...
end
end

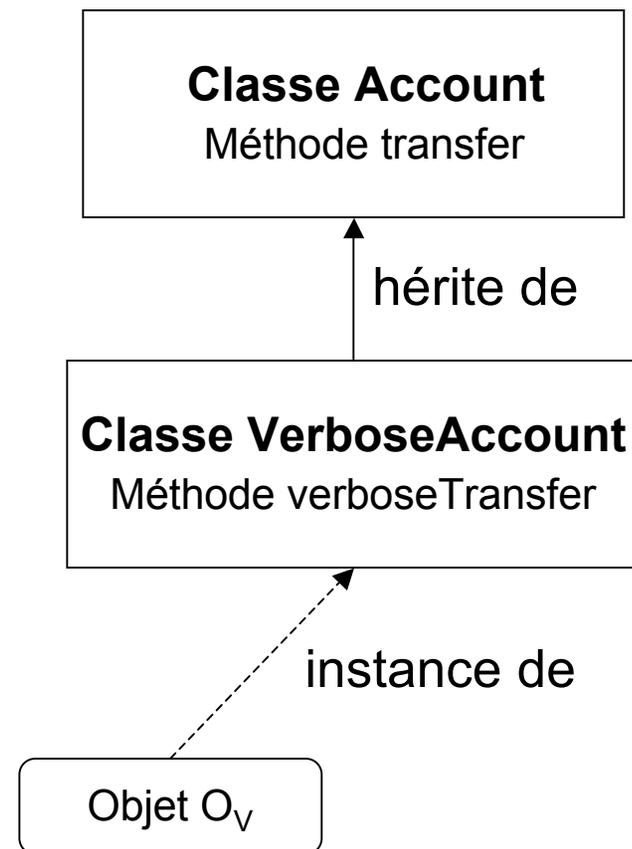
class AccountWithFee
  from VerboseAccount
  attr fee:5
  meth transfer(Amount)
  ...
end
end
```



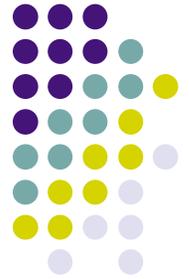


Lien dynamique

- Nous allons maintenant définir la nouvelle méthode verboseTransfer
- Dans la définition de verboseTransfer, nous devons appeler transfer
- On écrit {**self** transfer(A)}
 - La méthode transfer est choisie dans la classe de l'objet lui-même O_V
 - **self** = l'objet lui-même, une instance de VerboseAccount

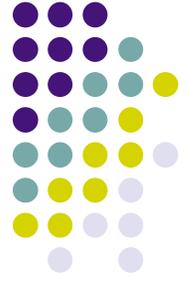


Définition de VerboseAccount



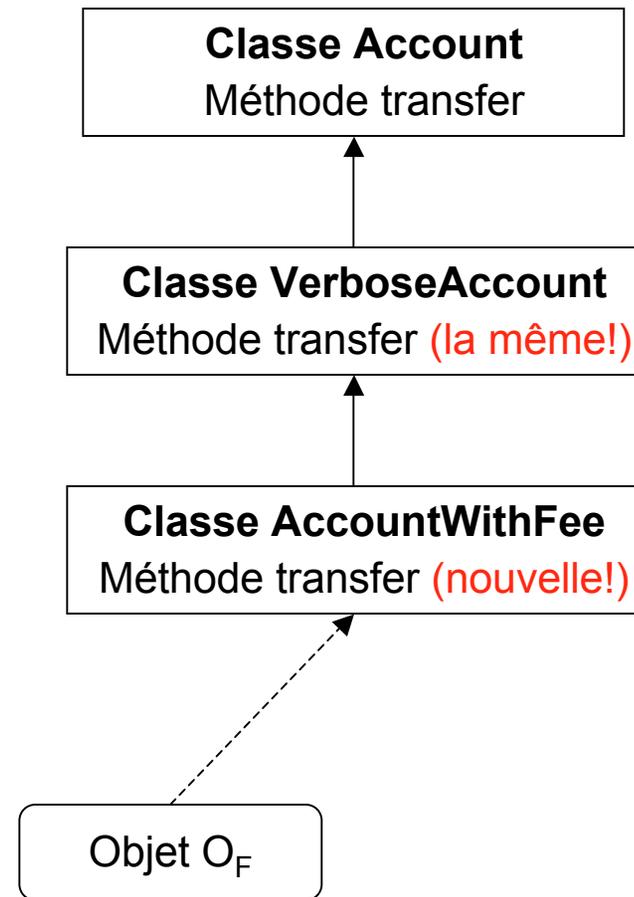
La classe
VerboseAccount
a les méthodes
transfer, getBal et
verboseTransfer

```
class VerboseAccount
  from Account
  meth verboseTransfer(Amount)
    {self transfer(Amount)}
    {Browse @balance}
  end
end
```

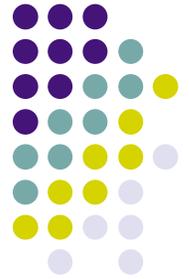


Lien statique

- Nous allons maintenant redéfinir l'ancienne méthode transfer dans AccountWithFee
- Dans la nouvelle définition de transfer, nous devons appeler l'ancienne définition!
- On écrit:
VerboseAccount,transfer(A)
 - Il faut choisir la classe dans laquelle se trouve l'ancienne!
 - La méthode transfer est choisie dans la classe VerboseAccount



Définition de AccountWithFee



```
class AccountWithFee
  from VerboseAccount
  attr fee:5
  meth transfer(Amt)
    VerboseAccount,transfer(Amt-@fee)
  end
end
```

La classe AccountWithFee a les méthodes transfer, getBal et verboseTransfer. La méthode transfer a été redéfinie.

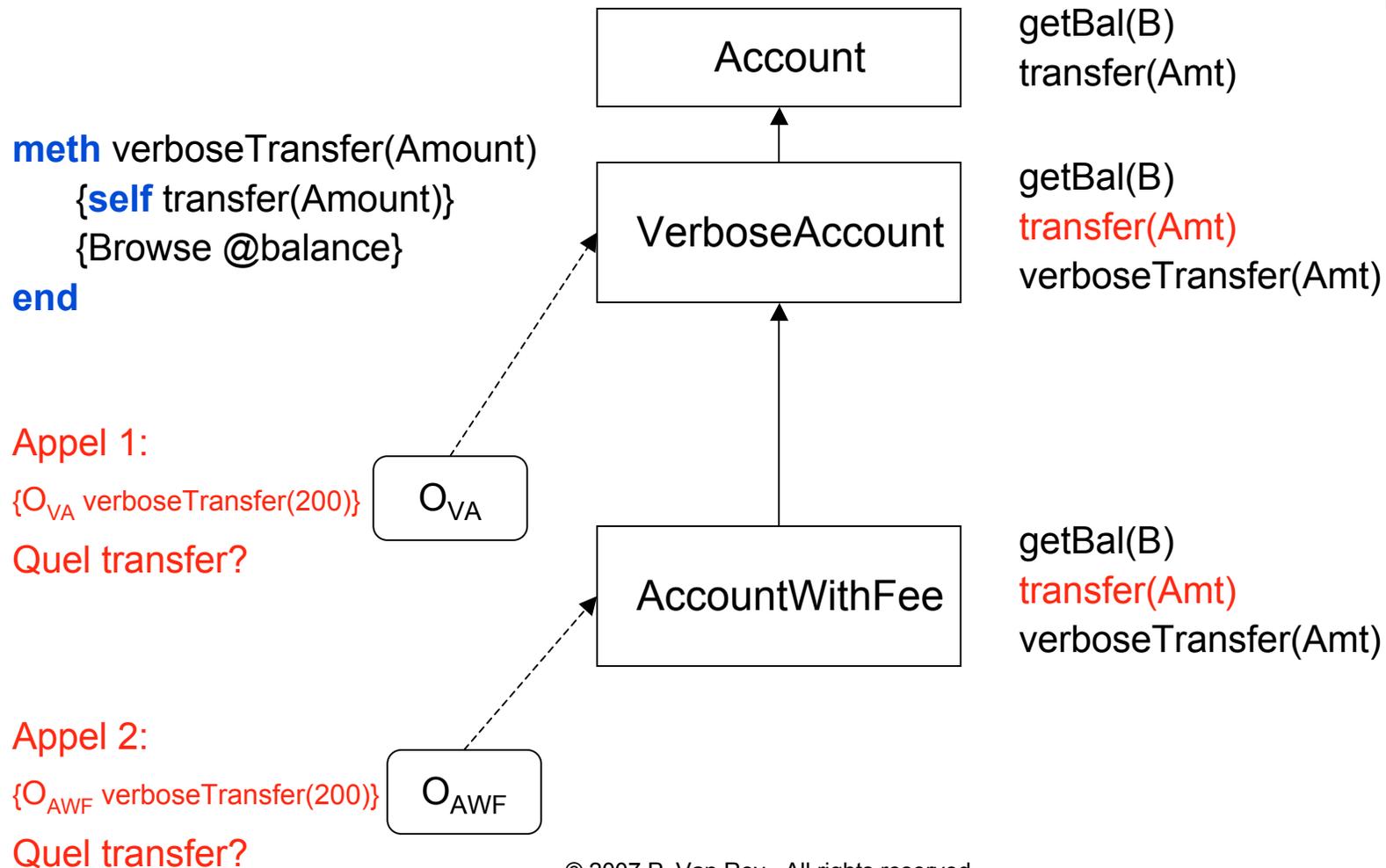


La magie du lien dynamique

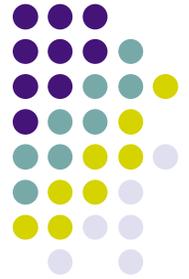
- Regardez le fragment suivant:
A={New AccountWithFee transfer(100)}
{A verboseTransfer(200)}
- Question: qu'est-ce qui se passe?
 - Quelle méthode **transfer** est appelée par **verboseTransfer**?
 - L'ancienne ou la nouvelle?
 - Attention: au moment où on a défini VerboseAccount, on ne connaissait pas encore l'existence de AccountWithFee
- Réponse: **!!**



Exemple du lien dynamique



Extension non-conservatrice: danger, danger, danger!

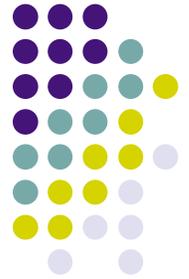


Danger!
Les invariants
deviennent faux.

```
class AccountWithFee
  from VerboseAccount
  attr fee:5
  meth transfer(Amt)
    VerboseAccount,transfer(Amt-@fee)
  end
end
```

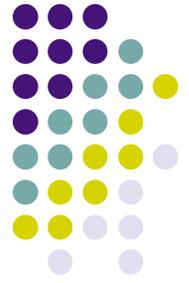
Invariant:
{A getBal(B)}
{A transfer(S)}
{A getBal(B1)}
% B1=B+S ?
% **Faux!**

Le principe de substitution: une leçon coûteuse

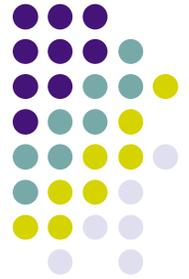


- Dans les années 1980, une grande entreprise a initié un projet ambitieux basé sur la programmation orienté-objet
 - Non, ce n'est pas Microsoft!
- Malgré un budget de quelques milliards de dollars, le projet a échoué lamentablement
- Une des raisons principales était une **utilisation fautive de l'héritage**. Deux erreurs principales avaient été commises:
 - **Violation du principe de substitution**. Une procédure qui marchait avec les objets d'une classe ne marchait plus avec les objets d'une sous-classe! Résultat: prolifération de procédures!
 - **Création de sous-classes pour masquer des problèmes**, au lieu de corriger ces problèmes à leur origine. Le résultat était une hiérarchie d'une grande profondeur, complexe, lente et remplie d'erreurs.

Liens statiques et dynamiques: recapitulatif



- Le but du lien dynamique et du lien statique est de sélectionner la méthode qu'on va exécuter
- **Lien dynamique**: {**self** M}
 - On sélectionne la méthode dans la classe de l'objet lui-même
 - Cette classe n'est connue qu'à l'exécution, c'est pourquoi on l'appelle un lien *dynamique*
 - C'est ce qu'on utilise **par défaut**
- **Lien statique**: SuperClass, M
 - On sélectionne la méthode dans la classe SuperClass
 - Cette classe est connue à la compilation (c'est SuperClass), c'est pourquoi on l'appelle un lien *statique*
 - C'est utilisé uniquement pour la **redéfinition** ("overriding")
 - Quand une méthode est redéfinie, il faut souvent que la nouvelle méthode puisse accéder à l'ancienne



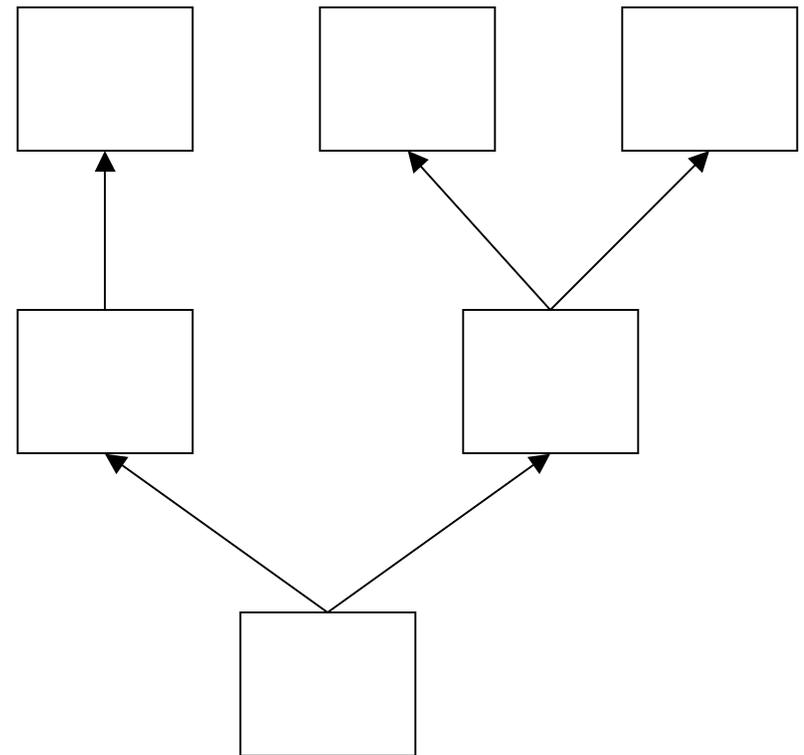
La relation de super-classe

- Une classe peut hériter d'une ou plusieurs autres classes, qui apparaissent après le mot-clé **from**
- Une classe B est appelée **super-classe** de A si:
 - B apparaît dans la déclaration from de A, ou
 - B est une super-classe d'une classe qui apparaît dans la déclaration from de A

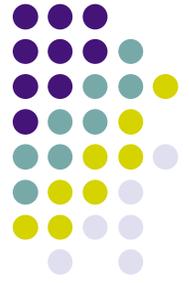


La relation de super-classe

- La relation de super-classe est orienté et acyclique
- Une classe peut avoir plusieurs super-classes
 - Héritage multiple

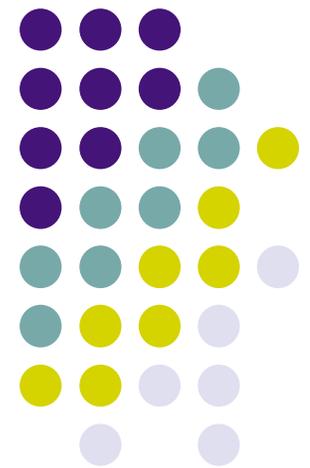


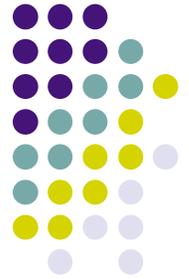
L'héritage simple et l'héritage multiple



- L'héritage simple = une seule classe dans la clause **from**
 - Beaucoup plus simple à implémenter et à utiliser
 - Java ne permet que l'héritage simple des classes
- L'héritage multiple = plusieurs classes dans la clause **from**
 - Un outil puissant, mais à double tranchant!
 - Voir sections 6.4.2 et 6.4.3 pour un bon exemple
 - (Voir aussi le livre “**Object-Oriented Software Construction**” de Bertrand Meyer)

Résumé





Résumé

- Les objets et les classes
 - Attributs (cachés) et méthodes (visibles)
 - L'envoi procédural
 - Une classe comme une fonction pour créer des objets
 - Soutien syntaxique pour les classes
- Le polymorphisme
 - Le principe de la concentration des responsabilités
- L'héritage
 - Le principe de substitution
 - Lien dynamique et lien statique
 - L'héritage simple et l'héritage multiple