HLIN603 : Programmation Objet Avancée

Héritage Multiple

Sommaire

Héritage multiple

Une classe peut avoir plusieurs super-classes directes
Deux conséquences immédiates :

- le graphe d'héritage a une structure de graphe sans circuit. Il peut avoir une racine, mais ce n'est pas obligatoire (ce n'est pas le cas en C++)
- lorsqu'on ferme transitivement le graphe, on obtient un ordre partiel entre les classes
- l'ensemble des super-classes d'une classe n'est plus totalement ordonné par la relation d'héritage

Sommaire

Une meilleure classification

Multi-classification avec plusieurs critères

Classification des polygones sur plusieurs critères : longueur des côtés, propriétés des côtés opposés, propriétés des côtés consécutifs.

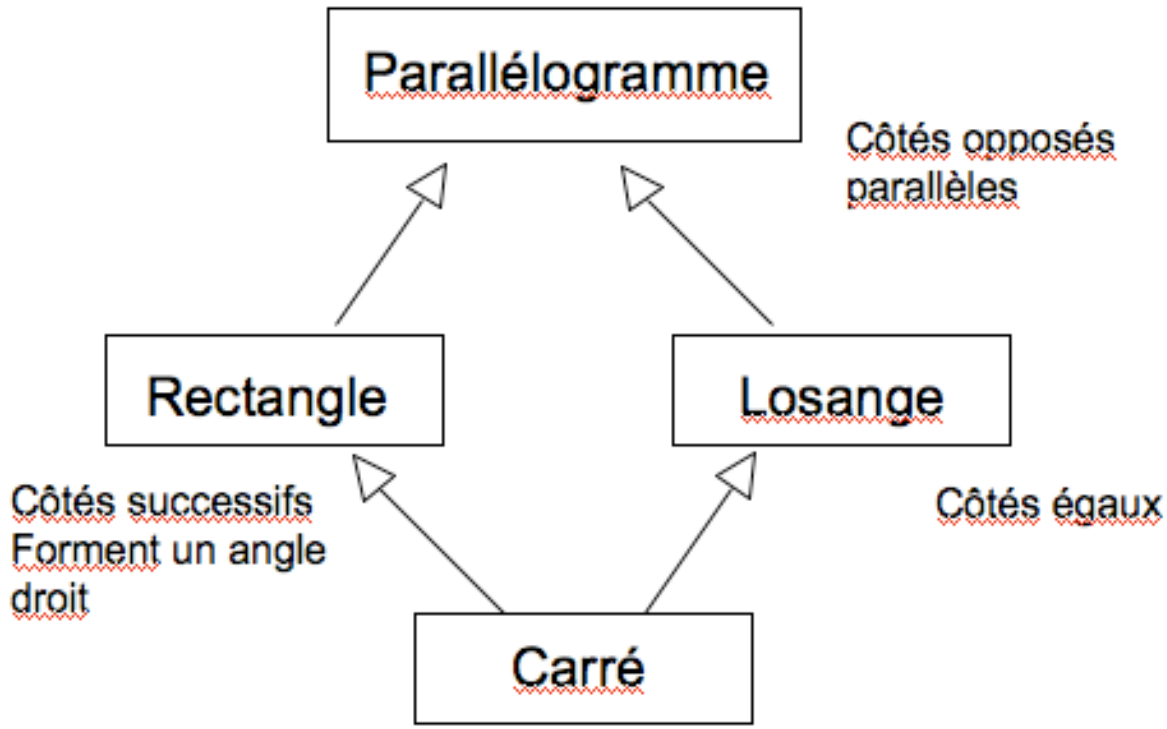


FIGURE : multi-classification de quelques polygones

Une meilleure classification

Multi-classification avec plusieurs critères

Classification des collections sur plusieurs critères : extension possible, indexation des éléments (entiers, clefs)

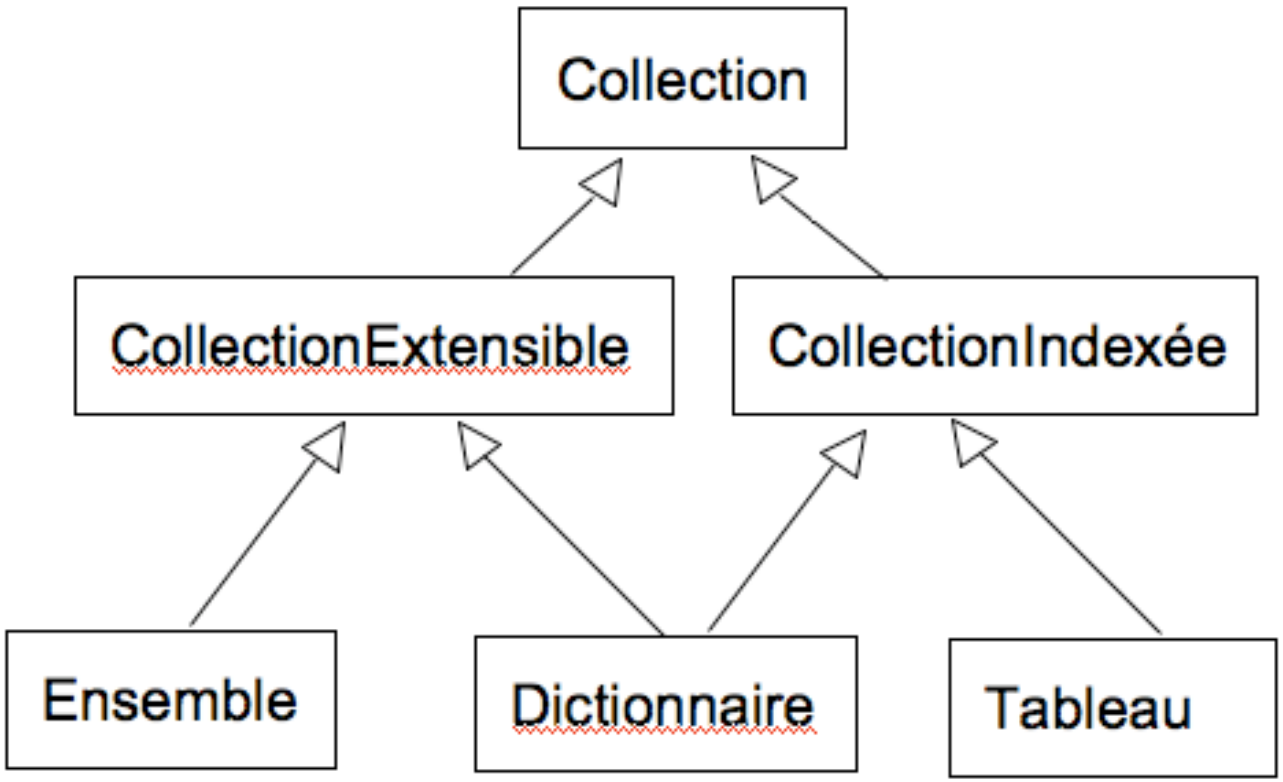


FIGURE : multi-classification de quelques collections

Une meilleure classification

Multi-classification avec un unique critère

Classification basée sur une caractéristique multi-valuée et dont les valeurs peuvent être cumulées par certains objets

Exemple : la caractéristique multi-valuée *zone d'utilisation*

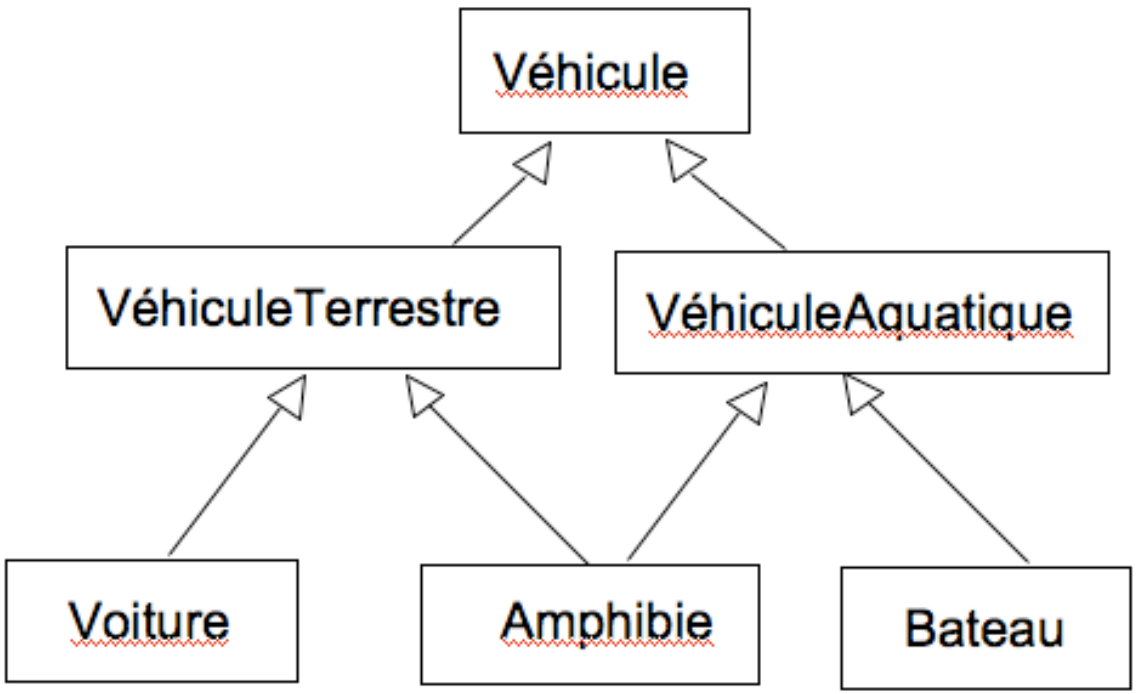


FIGURE : multi-classification de quelques véhicules

Un meilleur partage

- héritage, support à la factorisation des attributs et des méthodes
- évite des redondances dans le code du programme
- facilite les corrections et les modifications puisqu'un seul point du programme contient une déclaration ou un traitement donné
- économie d'écriture

Héritage simple : élimination impossible des redondances

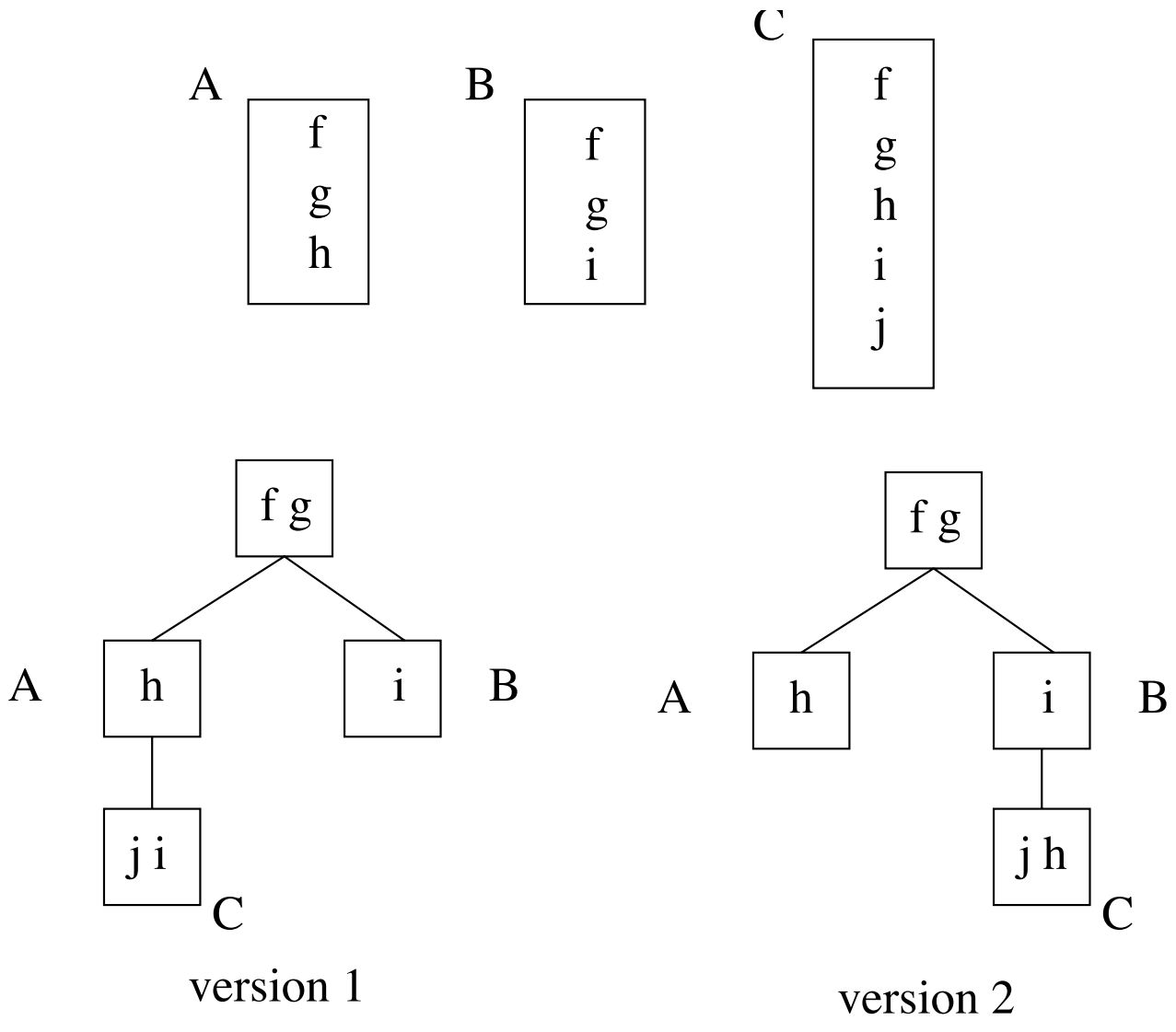


FIGURE : Trois classes et deux factorisations en héritage simple

Héritage multiple : permet d'éliminer les redondances

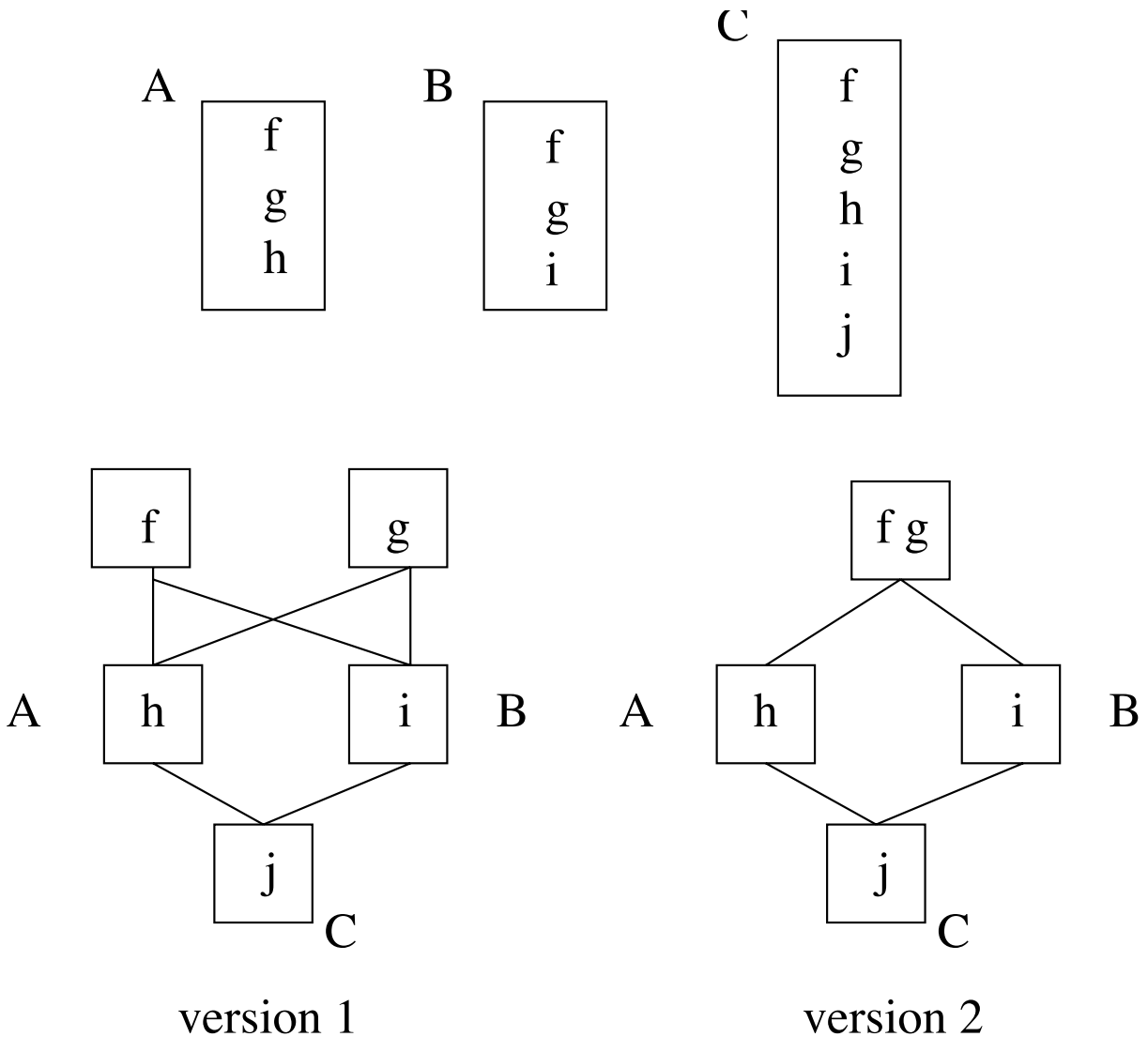
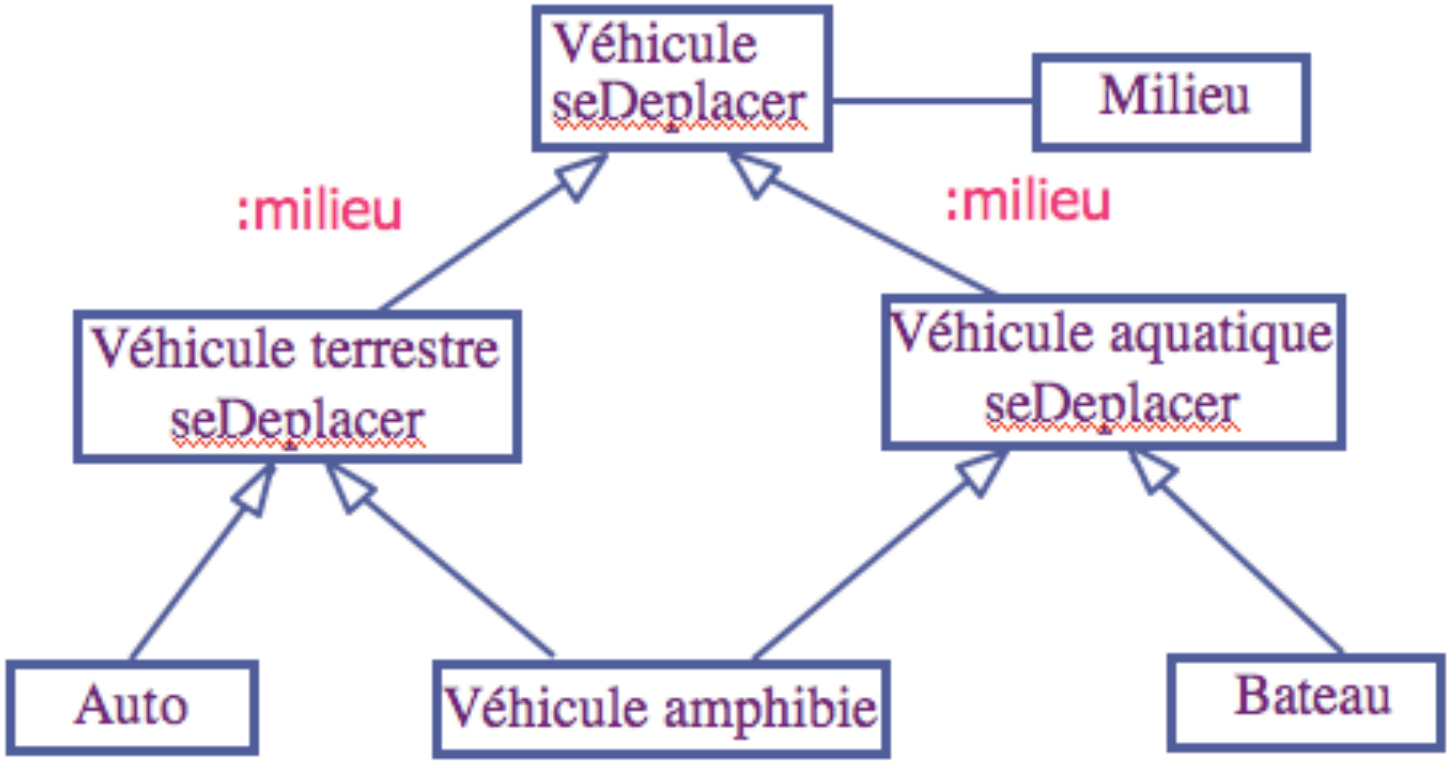


FIGURE : Trois classes et deux factorisations maximales

Conflits de valeurs



- Généralement, les propriétés qui produisent un conflit de valeur spécialisent une même propriété : dans notre exemple, il est logique que la classe des véhicules dispose également d'une méthode `seDeplacer`, même abstraite
- Les propriétés en conflit ont une origine commune

Conflit de nom

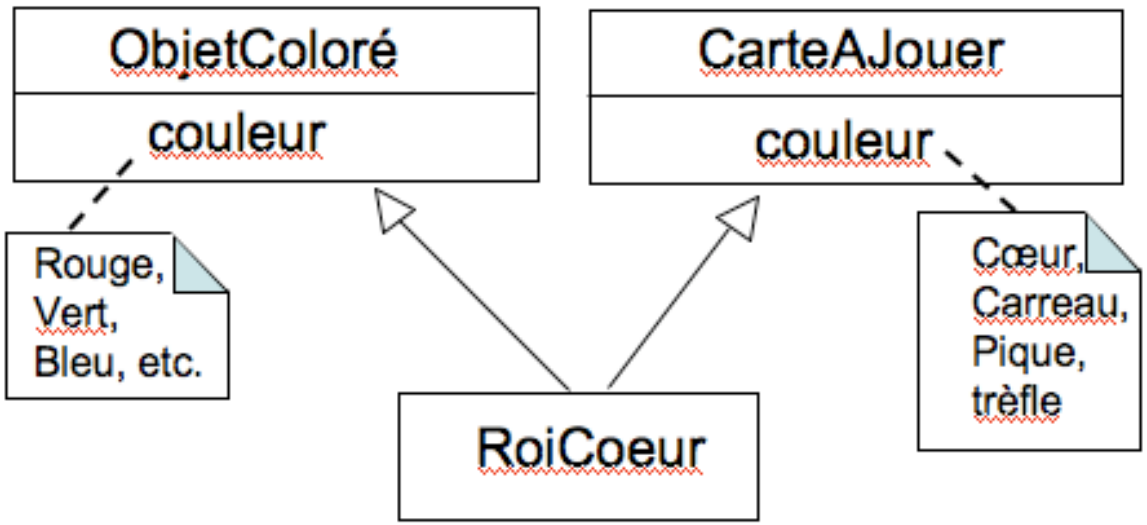


FIGURE : Situation de conflit de nom

Généralement pas d'origine commune de la propriété

Héritage répété

Statuer sur le sort des attributs hérités par plusieurs chemins

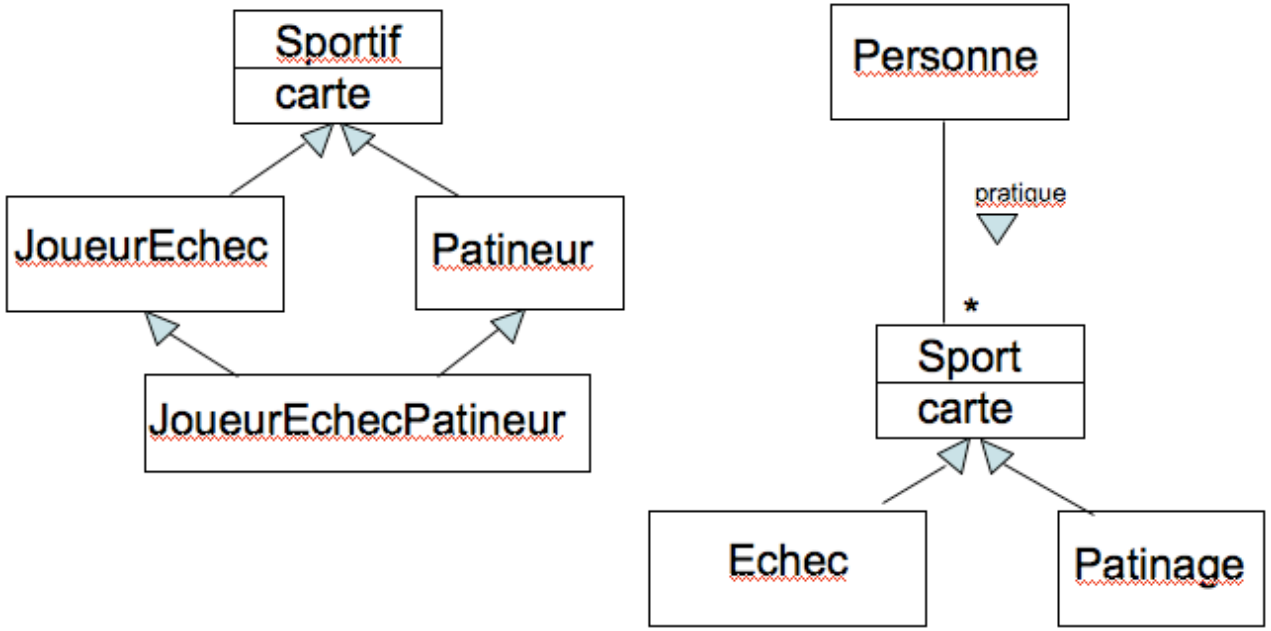


FIGURE : héritage multiple répété (gauche), solution sans héritage répété (droite)

Complexité de la classification

L'héritage multiple est-il vraiment plus complexe ?

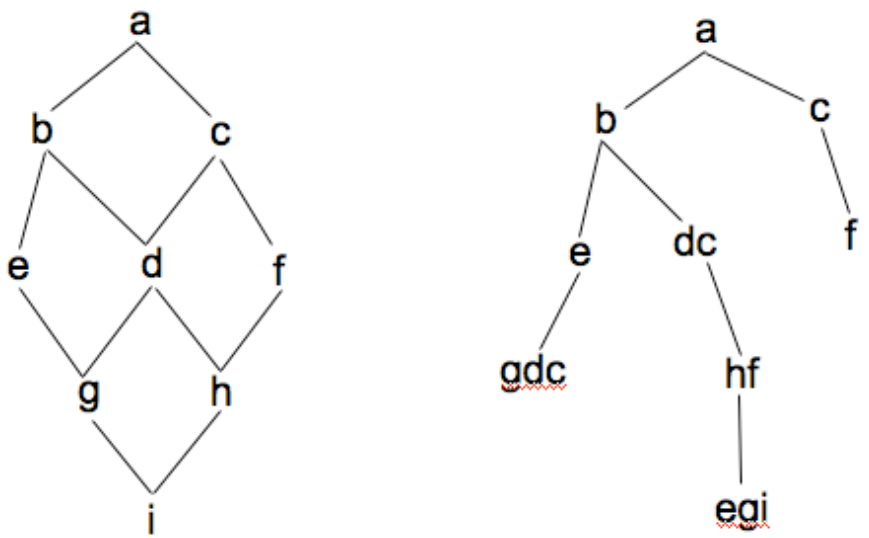


FIGURE : Régularité de la structure avec héritage multiple (gauche), moindre lisibilité de l'héritage simple (droite)

Sommaire

Résolution des conflits par désignation explicite

Avec l'opérateur de résolution de portée ::
et le nom de la classe

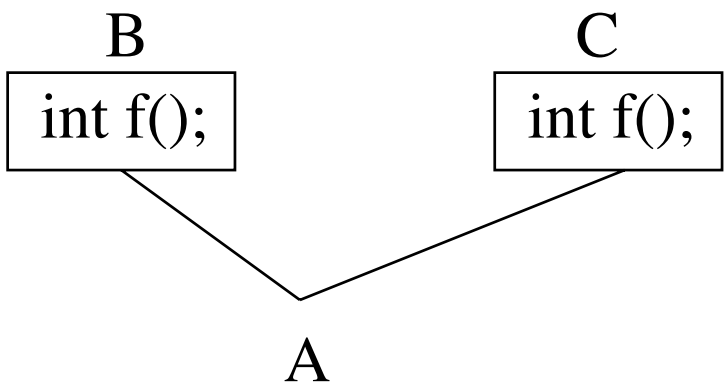


FIGURE : Situation de conflit

Résolution des conflits par désignation explicite

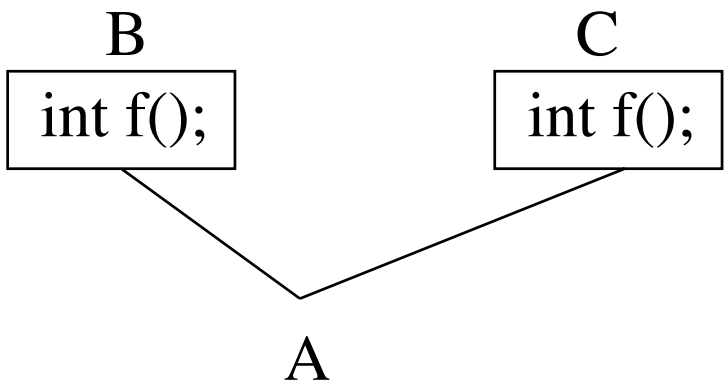


FIGURE : Situation de conflit

Les instructions suivantes se soldent par une ambiguïté :

```
A *instA = new A; instA->f();
```

Résolution des conflits par désignation explicite

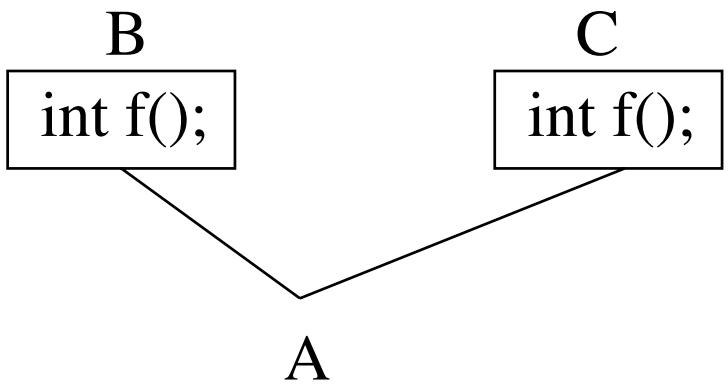


FIGURE : Situation de conflit

Trois solutions sont envisageables :

- écrire `instA->B::f();`
- écrire `instA->C::f();`
- redéfinir `int f()` dans `A`

Résolution des conflits par désignation explicite

Inconvénient 1

`D *instD=new D; instD->f()` et `instD->A::f()` sont ambiguës !

Connaissance de la hiérarchie nécessaire

Contraire aux principes de modularité et de localité

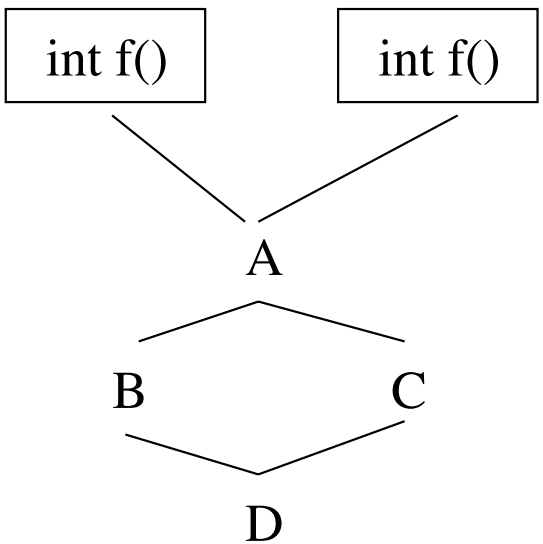


FIGURE : Désambiguisation et connaissance de la hiérarchie

Résolution des conflits par désignation explicite

Inconvénient 2

La règle de spécialisation n'est plus assurée

`C *instC=new C; instC->A::f();` appelle pour les objets de la classe `C` une méthode qui n'est pas adaptée, en l'occurrence pas la plus spécifique

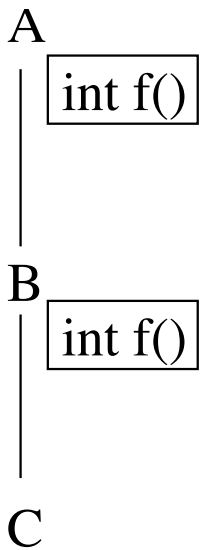


FIGURE : Désambiguisation et règle de spécialisation

Sommaire

Héritage répété versus héritage `virtual`

Les deux sont possibles !

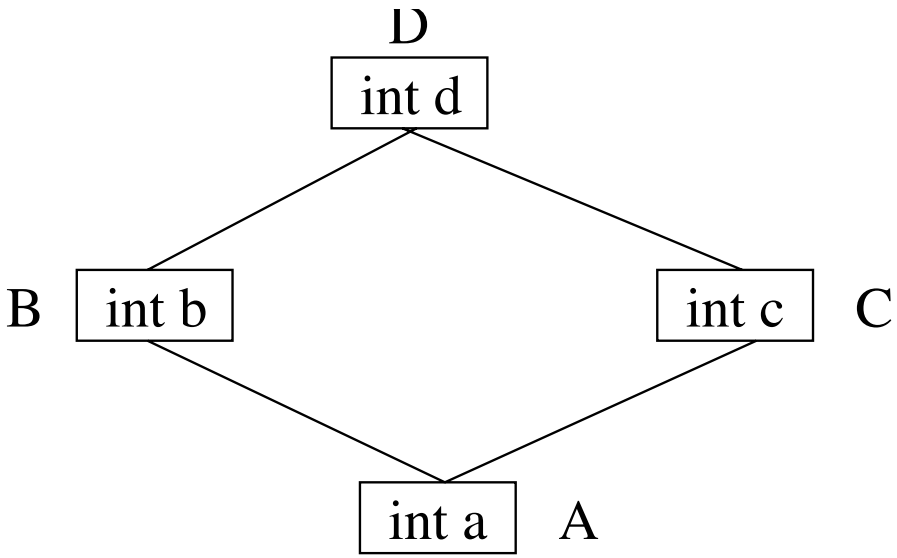


FIGURE : Situation d'héritage : une instance de `A` possède-t-elle 1 ou 2 attributs `d` ?

Héritage répété

Deux attributs d

```
class D {...};
class B : public D {...};
class C : public D {...};
class A : public B, public C {...};
```

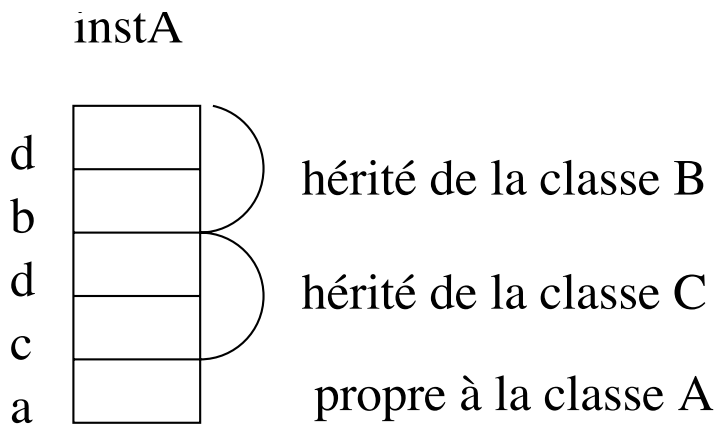


FIGURE : Situation d'héritage répété en C++, vue logique de l'instance

Héritage virtual

Un unique attribut d

```
class D {...};
class B : virtual public D {...};
class C : virtual public D {...};
class A : virtual public B, virtual public C {...};
```

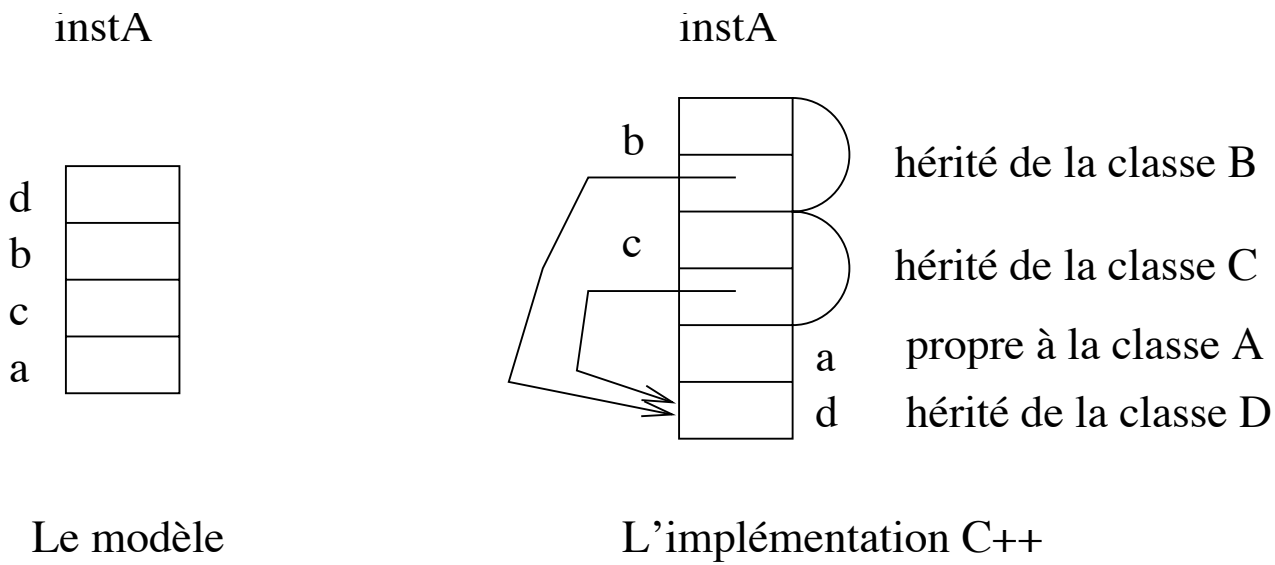


FIGURE : Situation d'héritage virtuel en C++, vue logique de l'instance

Sommaire

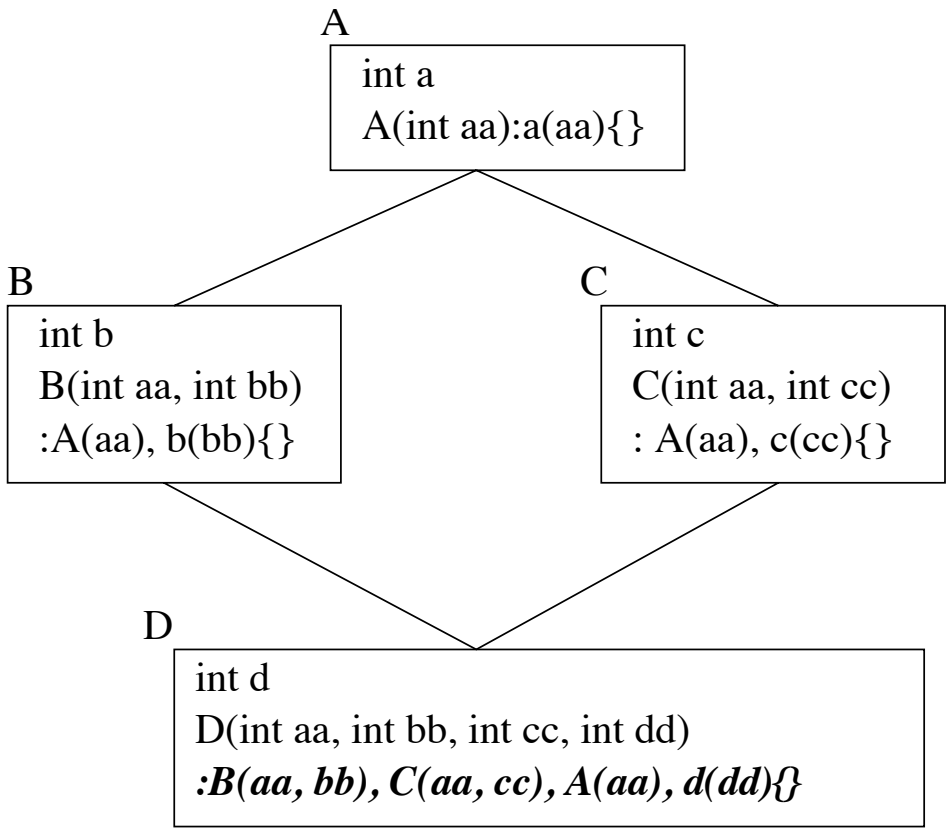
Transmission des paramètres aux constructeurs dans le cas de l'héritage virtuel

Dans le constructeur de `D` :

`B(aa,bb)` et `C(aa,cc)` ne transmettent pas `aa`

Ce pourrait être incoherent, par ex. `B(aa,bb)` et `C(bb,cc)`

Solution : réécrire `A(aa)`



Ordre d'appel des constructeurs et destructeurs

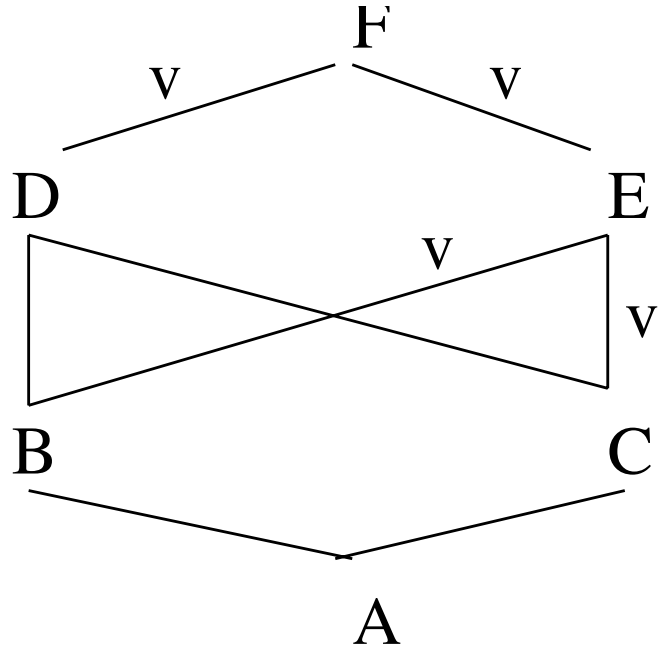
- **Héritage simple** lors de la création d'un objet, les constructeurs des superclasses et de la classe de l'objet sont appelés depuis le haut jusqu'en bas de la hiérarchie.
- **Héritage multiple** les superclasses ne sont pas totalement ordonnées par la relation d'héritage : un algorithme particulier de parcours des superclasses est utilisé pour déterminer un ordre total.
- Les destructeurs sont appelés systématiquement dans le sens inverse de celui des constructeurs.

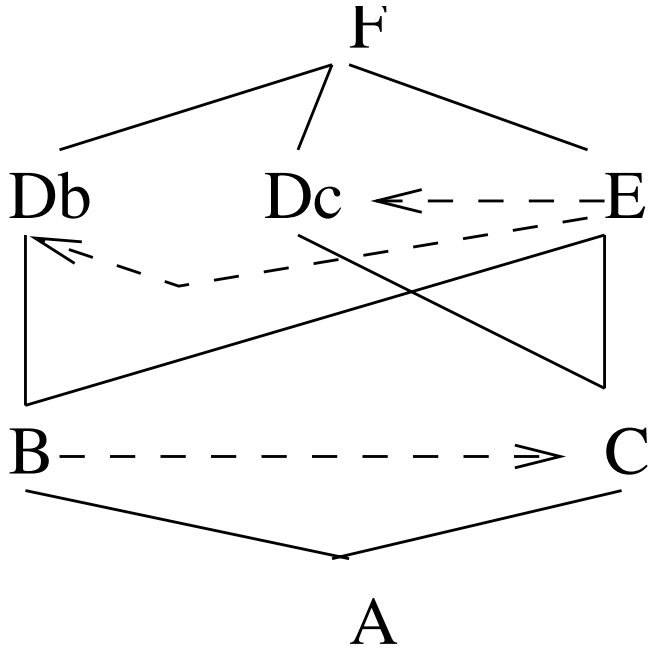
Ordre d'appel des constructeurs et destructeurs

On complète le graphe d'héritage :

- les classes héritées de manière répétée sont dupliquées autant de fois qu'il y a de chemins y menant depuis la classe de l'objet que l'on cherche à construire (ou à détruire).
- un ordre local range par ordre croissant les superclasses directes d'une classe
  - les classes héritées de manière virtuelle avant les classes héritées de manière répétée
  - dans chaque catégorie en suivant l'ordre de déclaration

Ordre d'appel des constructeurs et destructeurs





F E Db B Dc C A

L'ordre local entre les super-classes de `B` est tout d'abord `E` (héritée de manière virtuelle) puis `Db` (héritée de manière répétée). Pour la classe `A`, on suppose que la classe est ainsi déclarée :

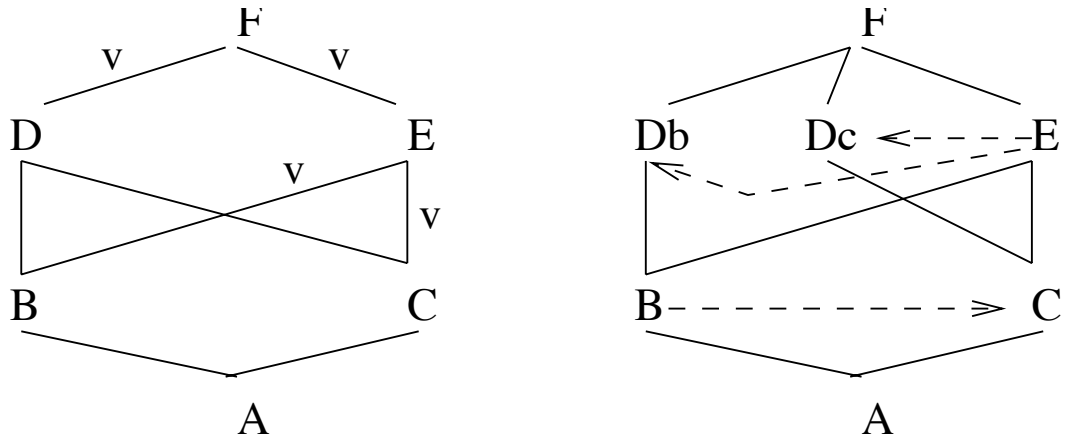
```
class A : public B, public C{.....};
```

Ordre d'appel des constructeurs et destructeurs

L'ordre d'appel des constructeurs est construit :

- par parcours du graphe étendu
- par l'ordre de dépilement d'un parcours en profondeur d'abord (dans lequel on ne repasse pas par les sommets déjà explorés) partant de la classe de l'objet
- en cas de choix, on utilise l'ordre local entre superclasses

Ordre d'appel des constructeurs et destructeurs



F E Db B Dc C A

A A B A B E A B E **F**
A B **E**
A B A B **Db**
A **B**
A A C A C **Dc**
A **C**
A

L'ordre de dépilement est donc : F E Db B Dc C A.

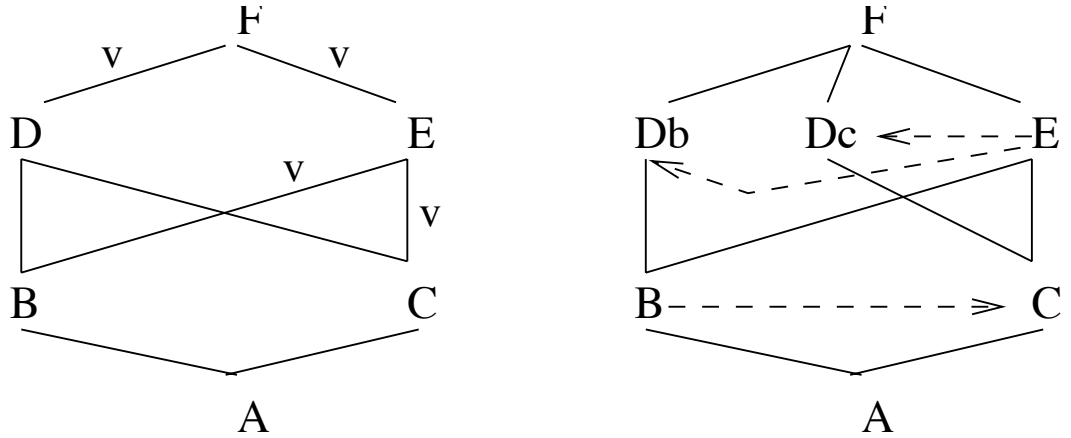
Ordre d'appel des constructeurs et destructeurs

Schéma algorithmique

```
ordreTotal <- vide
p.empiler(racine)
tant que ( not p.vide()) faire
   si (p.sommet() a des successeurs non marqués
            empiler le plus petit successeur
            pour l'ordre local de p.sommet()
   sinon
             s=p.depiler()
             ordreTotal <- ordreTotal + s
             marquer s
```

Cet algorithme calcule une extension linéaire de l'ordre inverse de l'ordre induit par l'héritage et assure ainsi qu'une classe est ramassée (ordre de dépilement) avant toutes ses sous-classes.

Ordre d'appel des constructeurs et destructeurs



F E Db B Dc C A

Un autre algorithme, moins efficace consiste à effectuer une descente en profondeur d'abord en respectant l'ordre inverse de déclaration :

A C Dc F E F **B Db** F **E F**

Dans cet ordre, on ne garde que les dernières occurrences de chaque sommet, puis on inverse le résultat.