TRAVAIL D'ETUDE LICENCE 2002-2003
UNIVERSITE NICE SOPHIA-ANTIPOLIS

Étude comparative des mécanismes d'héritage dans les langages C++, C#, Eiffel et Java

Sujet :

Quelles sont les similitudes, différences, quelles sont les qualités et les défauts de ces 4 langages du point de vue de l'héritage.

Nom :

- ***BOUKHOURROU Sophiane***
- ***GATTI Nicolas***

Encadrement :

- ***CRESCENZO Pierre***

SOMMAIRE :

1 L'importance de l'héritage

- ***1.1 Définition de l'héritage***
- ***1.2 Notion de spécialisation/généralisation***
- ***1.3 Autres intérêts de l'héritage***

2 Différents types d'héritage

- ***2.1 Héritage simple***
- ***2.2 Héritage multiple***
- ***2.3 Héritage multiple d'interface***

3 Héritage simple

4 Héritage multiple

- ***4.1 « Héritage multiple « en JAVA et C#***
- ***4.2 Héritage multiple en C++ et EIFFEL***

5 Conclusion

1 L'IMPORTANCE DE L'HERITAGE

1.1 Définition de l'héritage

Un des grands intérêts des langages orienté objet, c'est de pouvoir définir des dépendances entres classes. Cela permet, en particulier, de réaliser des programmes parfaitement modulaires en disposant de modules réutilisables.
Cette approche est à l'opposé de l'esprit de la programmation objet et du génie logiciel. Dans la mesure où l'on dispose déjà de la classe, les langages orientés objets offre un moyen bien plus simple pour définir cette nouvelle classe. Il s'agit de ***l'héritage*** **.**
L'*héritage* est une caractéristique des langages orientés objet. Une classe obtenue *par héritage* possèdent la totalité des membres de la classe de base ainsi toutes ses méthodes. Une classe B peut donc se définir par rapport une autre classe A. On dira que la sous classe B hérite des attributs et fonctionnalités de la *classe de base* A.
L'ensemble des classes sont organisés de manière hiérarchique permettant de structurer l'ensemble des informations manipulées par un programme.
L'**héritage** permet un *partage hiérarchique* des propriétés (attributs et opérations).
Une **sous-classe** (ou *classe fille* peut incorporer, ou *hériter*, des propriétés d'une **super-classe** (ou *classe mère*). Généralement une super-classe définit les grands traits d'une abstraction, une sous-classe hérite de cette définition et peut la modifier, raffiner et/ou rajouter ses propres propriétés.

1.2 Notion de spécialisation/généralisation

Souvent dans les langages objets, on trouve des caractéristiques communes à différents objets. Le but est de regrouper les objets présentant des similitudes pour former une classe de base (classe mère), on effectue ainsi une **généralisation** des classes**.** Elle revient en mettre en facteur commun certaines structures ou certains comportements. Dans tous les cas de figures il faut essayer de maximiser la généralisation en regroupant autant d'attributs que possible et d'associations que possible.

Réciproquement parfois il est nécessaire de préciser un objet quand des dissimilitudes se présentent, on subdivise alors la classe mère en classes filles (ou sous classes) celles–ci hériteront des comportements et des attributs. Les classes peuvent donc être plus spécifiques, l'héritage permet une **spécialisation** de la classe de base. Outre la possibilité de récupérer toutes les propriétés et méthodes de la super-classe, les classes dérivées peuvent également introduire de nouveaux attributs et méthodes qui les enrichissent. Une classe Personne ayant par exemple comme attribut l'age et le nom pourra se dériver en une classe Employe auquel on ajoutera l'attribut salaire et la méthode qui calcule son salaire.

1.3 Autres intérêts de l'héritage

L'atout essentiel d'un langage orienté objet est que le code est réutilisable. Grâce à l'héritage, on peut faire dériver une nouvelle classe d'une classe existante et ainsi en récupérer les attributs et méthodes, sans avoir à la réécrire totalement. On évite ainsi la

redondance du code. L'héritage permet également de raccourcir le temps de mis au point du code déjà utilisé. Elle facilite la modélisations des objets et la modularité.

Le concept de l'héritage est un concept général aux langages objets, mais ces langages font des choix différents dans la manière de les représenter.

2 Différents types d'héritage

2.1 L'héritage simple

Posons nous déjà la question ***qu'est ce que l'héritage simple ??***

L'héritage simple est un mécanisme de transmission des propriétés d'une classe vers une sous-classe. Pratiquement, cet héritage signifie qu'un nom n'est pas toujours défini dans la classe, mais peut être défini dans un des ancêtres de cette classe. Ainsi pour associer un nom à sa définition, un langage orienté objets sera amené à parcourir la relation d'héritage pour trouver une définition pour ce nom. Comme il s'agit d'associer un nom à sa définition, il s'agit d'un cas particulier du problème général de la portée des noms. Un nom utilisé par un objet d'une classe A, peut avoir plusieurs définitions parmi les ancêtres de la classe A. Pour l'héritage simple, comme l'ensemble des ancêtres de A est totalement ordonné, il suffit de choisir la définition la plus récente pour la relation d'héritage. Avec l'héritage multiple, la situation sera plus complexe et sera étudiée plus loin.
Exemple :
Un avion à réaction est un genre particulier d'avion, un avion est lui-même un véhicule volant etc... La relation d'héritage correspond à la relation *est-un* ou *est une sorte de* .Plus précisément un avion à réaction est un avion + quelques propriétés particulières.

2.2 L'héritage multiple

Posons nous déjà la question ***qu'est ce que l'héritage multiple ??***

L'héritage multiple est mécanisme par lequel une sous-classe hérite des propriétés de plus d'une super-classe .L'héritage multiple est une extension au modèle d'héritage simple où l'on autorise une classe à posséder plusieurs classes mères afin de modéliser une généralisation multiple.
Par héritage multiple on désigne les classes qui peuvent hériter de plusieurs autres classes à la fois. Par exemple on peut très bien imaginer une classe hydravion qui hériterait à la fois de la classe bateau et avion.

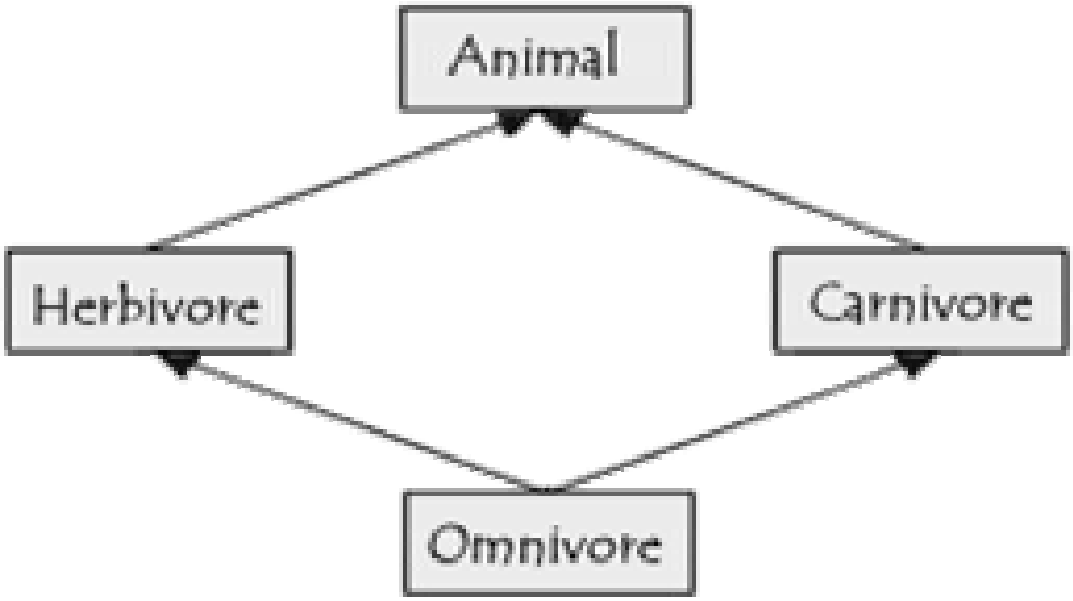
2.2.1 Les concepts

Le principe reste le même que pour l'héritage simple. Il faut que l'espace mémoire qui représente l'objet soit interprété comme il faut par toutes les méthodes des classes dont hérite un objet. C'est l'implémentation qui devient plus complexe avec des problèmes lorsque l'on hérite plusieurs fois d'un même parent (chose qui était techniquement impossible avec l'héritage simple).

Par rapport à un héritage simple on doit placer une indirection supplémentaire pour accéder aux attributs/méthodes.

En effet, prenons un objet X qui hérite de la classe CY et de la classe CZ. Lorsque l'on passe en paramètre l'objet X à une méthode de la classe CY elle doit pouvoir retrouver sa vision d'un objet Y de même pour CZ. Or si on place tout ce qui concerne la classe CY au début de la mémoire de l'objet puis tout ce qui concerne la classe CZ après. Alors les méthodes de CY et CZ ne peuvent plus utiliser l'offset par rapport au début de l'objet car celui-ci est devenu variable. En effet un objet de la classe CZ aura ses attributs/méthodes au début alors qu'un objet de la classe CX ne les aura pas au début. Pourtant ces deux objets doivent pouvoir être traités par la méthode de la classe CZ. D'où la nécessité d'introduire une indirection avant d'accéder aux champs qui nous intéressent. Seulement une classe peut être composée de n'importe quoi, on ne peut donc pas trouver un offset fixe qui indiquerait où aller pour trouver les attributs appartenant à une classe. Il n'y a donc pas plusieurs solutions. Il faut identifier de manière unique les classes et ensuite on devra faire une recherche pour trouver où sont les attributs cherchés en fonction de la classe qui les a déclaré. Pour faire cela, on a évidemment beaucoup de solutions d'implémentation possibles (si vous n'avez pas tout compris cela devrait se clarifier par la suite).

2.2.2 Une implémentation possible



Voici un exemple où la classe Animal hérite de la classe herbivore ainsi que de la classe Carnivore qui elles même héritent aussi de la classe Omnivore.

2.2.2.1 L'héritage

Dans chaque classe on défini un ou des constructeurs qui ont pour but de créer des objets de cette classe (sauf dans le cas des classes abstraites). C'est dans cette procédure que l'on alloue la mémoire nécessaire à l'objet que l'on crée. On alloue la place pour tous les parents plus pour les champs nouvellement définis dans notre classe. On rempli aussi la table des vues qui indiquera à chaque classe ou sont ses champs spécifiques. Ensuite, on initialise l'objet en appelant les constructeurs de chaque classe parentes en commençant par la plus lointaine (classe dont tout le monde hérite) pour que la surcharge puisse fonctionner.
Les constructeurs sont des fonctions qui font pointer les pointeurs de fonctions sur les implémentations (ceci est caché dans les langages objets). C'est aussi ici que l'on initialise les attributs si nécessaire (c'est le programmeur qui détermine les initialisations).

Au final, notre objet possède bien tous les attributs et méthodes des ses parents et de sa classe. On a donc bien l'héritage.

2.2.2.2 Le polymorphisme

Avec notre système d'indirection un objet qui hérite d'une classe peut toujours être considéré comme un objet de cette classe. Les objets d'une classe peuvent donc bien revêtir plusieurs formes réelles.

2.2.2.3La surcharge

La surcharge est une chose assez aisée. Il suffit d'accéder au pointeur sur la fonction que l'on veut surcharger (en y accédant comme n'importe quel attribut par le système d'indirection) et de changer sa valeur pour le faire pointer sur la nouvelle implémentation.

2.2.2.4 Les méthodes abstraites

Dans notre implémentation de l'héritage multiple, cela revient à déclarer un pointeur sur la fonction voulue (comme pour les méthodes normales) en le faisant pointer sur n'importe quoi. Ceci, au niveau de la classe abstraite (celle qui ne déclare pas le code) qui est la première à faire apparaître ce champs même si elle n'en donne pas le code.

Ensuite il suffira pour les héritiers d'accéder à ce champ de leur parent pour le faire pointer sur l'implémentation qu'ils ont faite. C'est exactement le même principe que pour une surcharge de fonction sauf que dans la surcharge, le pointeur pointait sur une adresse de fonction valide.

2.3 L'héritage multiple d'interface

2.3.1 qu'est ce q'une interface

Les interfaces sont des manières plus sophistiquées, d'organiser et de contrôler les objets du système construit, que ce que nous pouvons trouver dans beaucoup de langages de programmation orientés objets. Elles ne contiennent aucun code, juste des déclarations de fonctions.

2.3.2 pourquoi les interfaces

L'héritage multiple (qui lui aussi fait partie d'UML) peut poser des problèmes (problème du Diamant):
imaginons qu'une classe D hérite de classes B et C, chacune héritant d'une même classe A. Si on appelle, sur un objet de D, une méthode de A redéfinie dans B et C, que faut-il faire ?
S'il est possible d'expliquer à un compilateur ce qu'il faut faire, ce genre de situation est à éviter car, lorsque le code va évoluer, on perdra de vue ce genre de problème, et les bugs deviendront difficiles à traquer.
Par conséquent, il convient d'éviter de faire hériter une classe D de deux classes B et C assez proches, par contre, si les classes B et C sont complètement différentes, il n'y a aucun problème.
Bien que leurs fonctionnalités, en elles-mêmes, soient relativement simples leur utilisation relève de la conception.
Les concepteurs de Java sont arrivés à une conclusion différente :
ils ont carrément interdit l'héritage multiple.
Ensuite, ils ont du en rajouter un peu mais juste l'héritage multiple d'interfaces.

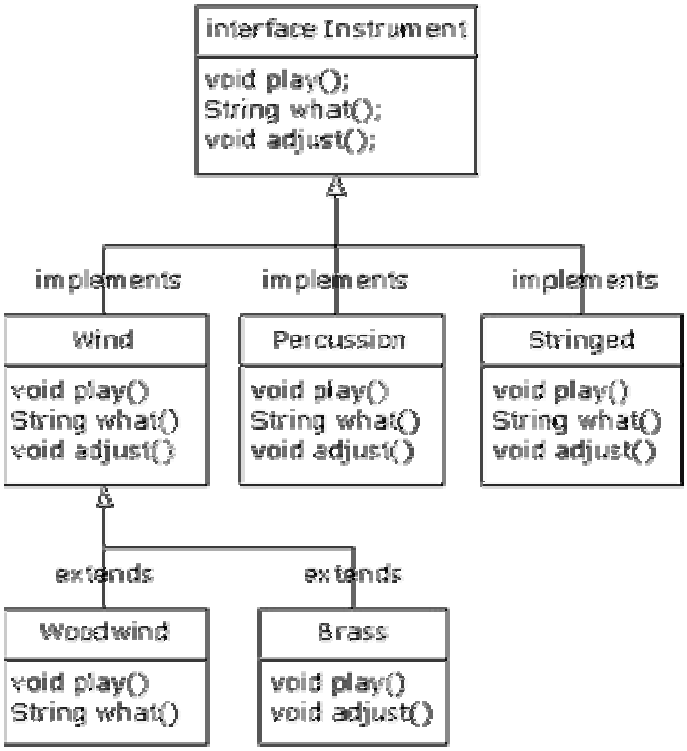
Voici ce que l'on aimerait écrire.

```
class B { ... }
class C { void foo() { ... } ... }
class A extends B, C { ... }
```

Et voici ce que l'on est contraint d'écrire

```
class B { ... }
interface CSimplifiedInterface { void foo(); }
class C implements CSimplifiedInterface { ... }
class A extends B implements CSimplifiedInterface {
  C c;
  void foo() { c.foo(); }
  ...
}
```

Voici un exemple sur l'héritage d'interface :



3 Héritage simple

Par héritage simple on entend qu'une classe ne peut hériter que d'une seule classe (de manière directe). Par contre cela ne limite en rien le nombre de classes qui peuvent hériter d'une classe. C'est le choix du langage Java qui offre quand même quelques possibilités d'héritage multiple avec les interfaces (spécification d'objets dont on a parlé ci-dessus). Avec l'héritage simple, rien de plus simple. On place au début de la mémoire les attributs/méthodes du parent puis après, ceux de la nouvelle classe. Alors l'objet sera
interprété comme il faut si on utilise des méthodes des parents qui ne verront que la partie de l'objet qui les intéressent (la suite leur sera comme masquée par leur méconnaissance).

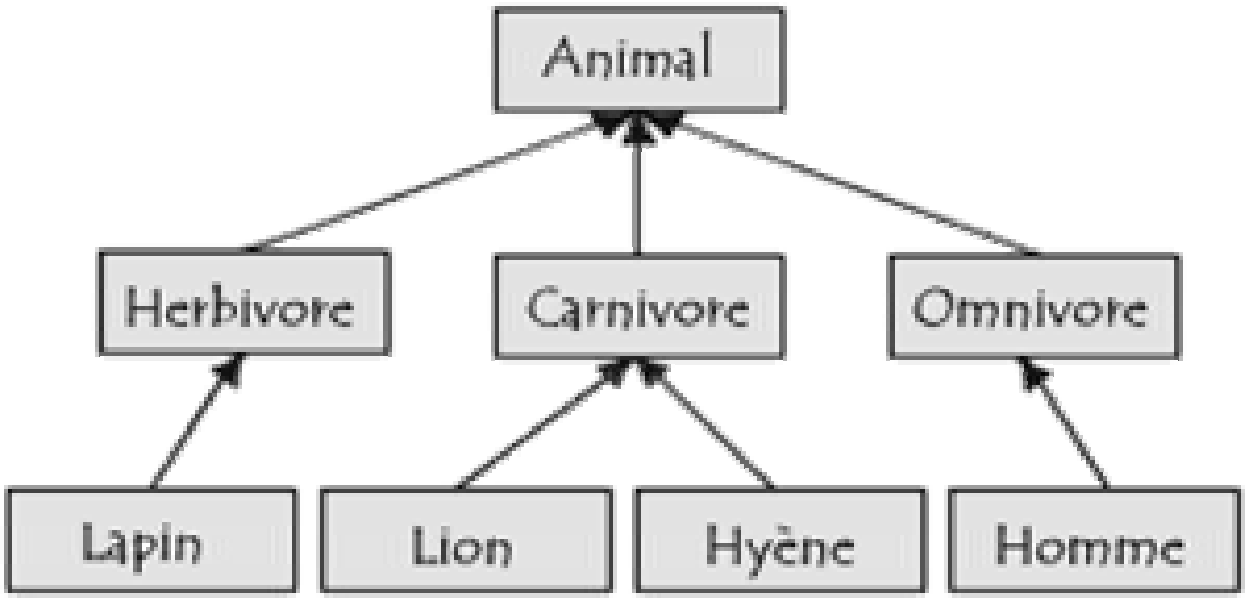
3.1 Les concepts

Un objet ce n'est rien d'autre qu'une zone mémoire où on regroupe un ensemble de variables qui ensemble définissent une entité logique. Un objet peut donc se définir par un pointeur. Tout le reste est dans l'interprétation de ce que l'on trouve à cette adresse. Si je considère qu'il s'agit d'un objet X alors je sais qu'à l'offset 4 par rapport au début

de l'objet (par exemple) je vais trouver un entier qui représente une certaine somme sur un compte bancaire.
Maintenant, ce que l'on veut c'est que si l'objet X hérite de la classe CY alors si j'applique une méthode propre à l'objet CY elle est la bonne interprétation de la zone mémoire où se trouve notre objet X.

3.2 Une implémentation possible

L'héritage simple ne présente pas trop de problème à mettre en oeuvre. Chaque objet ne fait qu'étendre un autre objet. Il suffit donc pour un objet de placer en son début la définition de son parent. Voici un exemple :



La classe Lapin hérite de la classe Herbivore qui hérite elle aussi de la classe Animal.
La classe Lion hérite de la classe Carnivore qui hérite elle aussi de la classe Animal.
La classe Hyène hérite de la classe Carnivore qui hérite elle aussi de la classe Animal.
La classe Homme hérite de la classe Omnivore qui hérite elle aussi de la classe Animal.

Donc toutes les classes du 3éme niveau peuvent accéder aux fonctions du 2éme niveau ainsi que celle du 1er niveau. Au final, nos objets possèdent bien tous les attributs et méthodes des ses parents et de sa classe. On a donc bien l'héritage.

4 Héritage multiple

4.1 « Héritage multiple » en Java et en C#

Le langage *Java* et C# ne permettent pas de l'*héritage multiple* . Ils pallient ce manque par l'introduction des *interfaces* . Le choix délibéré de *Java* et de C# de supprimer l'héritage multiple est dicté par un souci de simplicité.

4.1.1 « Héritage multiple » en Java

Mais qu'est-ce qu'une interface ? Il s'agit essentiellement d'une annonce (ou d'une promesse) indiquant qu'une classe implémentera certaines méthodes. On utilise d'ailleurs le mot clé implements pour signaler qu'une classe tiendra cette promesse. Commençons par voir comment on définit une interface et comment on l'utilise, avant d'en étudier les propriétés en détail. Pour supprimer les problèmes liés à l'héritage multiple, les *interfaces* sont des ``sortes de classes'' qui ne possèdent que des champs static final (autant dire des constantes) et des méthodes abstraites. En fait, les interfaces sont un moyen de préciser les services qu'une classe peut rendre. On dira qu'une classe implante une interface Z si cette classe fournit les implantations des méthodes déclarées dans l'interface Z. Autrement dit, la définition d'une interface consiste se donner une collection de méthodes abstraites et de constantes. L'implantation de ces méthodes devra évidemment être fournie par les classes qui se réclament de cette interface.

Bref, une interface est une classe abstraite dans laquelle, il n'y a ni variables d'instances et ni méthodes non abstraites.

Les interfaces ne fournit pas l'héritage multiple car :

- on ne peut hériter de variables d'instances
- on ne peut hériter d'implantation de méthodes
- la hiérarchie des interfaces est totalement indépendante de celle des classes.

4.1.1.1 Déclaration des interfaces

Les interfaces contiennent uniquement des définitions de constante et des déclarations de méthodes, sans leur définition (autrement dit, les méthodes indiquées dans les interfaces ne comportent que le prototype, et pas le corps). La définition d'une interface se présente comme celle d'une classe. On utilise simplement le mot clé **interface** à la place de **class** :

- Toutes les méthodes qui sont déclarées dans cette interface sont abstraites ; aucune implantation n'est donnée dans la définition de l'interface. Toutes les méthodes étant publiques et abstraites, les mots clés public et abstract n'apparaissent pas : ils sont implicites.
- Toutes les méthodes d'une interfaces sont toujours publiques et non statiques.
- Tous les champs d'une interface sont public, static et final. Ils sont là pour définir des constantes qui sont parfois utilisées dans les méthodes de l'interface. Les mots clés static et final ne figurent pas dans la définition des champs ; ils sont implicites.

La syntaxe de la déclaration d'une interface est la suivante :

```
interface I {
```

```
        Liste des constantes ;
        …
        Liste des prototypes de méthodes ;
        …

}
```

Déclaration sans définition :

```
 interface Visu{

        Void affiche() ;

}
```

Par convention les noms des interfaces commencent par une lettre majuscule et se terminent par ``able'' ou ``ible''. Dans nos exemples, on n'appliquera pas ces conventions.

4.1.1.2 Interface publique

Par essence, les méthodes d'une interface sont abstraites (puisqu'on n'en fournit pas de définition) et publiques (puisqu'elles devront être redéfinies plus tard). Néanmoins, il n'est pas nécessaire de mentionner les mots clés public et abstract (on peut quand même le faire). En l'absence de ce qualifier, elle ne peut être utilisée que par les seules classes appartenant au même *package* que l'interface.

Une interface peut être dotée des mêmes droits d'accès qu'une classe (public ou droit de paquetage). Dans le cas où on désire avoir une interface publique :

```
 Public interface Visu{

        Void affiche() ;

}
```

Contrairement aux classes, on ne peut qualifier une interface de private ou protected.

Les champs d'une interface sont des champs static. Toutes les règles d'initialisation des champs statiques s'appliquent ici.

Les qualifiés transcient, volatile ou synchronized ne peuvent être utilisés pour les membres des interfaces. De même, les qualifiés private et protected ne doivent être utilisés pour les membres des interfaces.

4.1.1.3 Implanter des interfaces

Les interfaces définissent des ``promesses de services''. Mais seule une classe peut rendre effectivement les services qu'une interface promet. Autrement dit, l'interface

toute seule ne sert à rien : il nous faut une classe qui implante l'interface. Une classe qui implante une interface le déclare dans son entête

```
Public class A implements I{
        Ici on doit (re)définir les méthodes déclarées dans l'interface I
}

public class Fiche implements Visu{
private String  Nom , Prenom ;
int Age ;


public Fiche(String n , String p , …){
        ...
}

Définition de la méthode affiche déclarée dans l'interface Visu
public void Affiche(){
        ...
}
```

Alors, la classe A est contrainte à définir toutes les méthodes prototypées dans l'interface I ; plus exactement, si la classe A ne définit pas toutes les méthodes prototypées dans I, elle doit être déclarée abstraite et ne pourra donc pas être instanciée

Sachant que la classe A implémente l'interface I, on sait (si A n'est pas abstraite) qu'elle dispose de toutes les méthodes de cette interface I ; on a un renseignement sur ce dont on peut disposer avec les instances de cette classe (c'est à dire un renseignement sur "l'interface" au sens usuel du terme). Concrètement, en reprenant l'exemple de la classe Fiche avec l'interface Visu, on peut écrire :

```
Visu pers ;

Pers = new Fiche(« moi », »toto » ,…) ;

Pers.Affiche() ;
```

On peut donc définir des variables de type interface, ici de type Visu ; on peut alors invoquer uniquement la méthode déclarée dans Visu sur un objet référencé par une variable de type Visu ou si cela avait été le cas utiliser les constantes définies dans l'interface. Cela a des conséquences fondamentales sur les possibilités de programmation.

```
Interface I {

        Void methode() ;
```

```
}

class A implements I{

….

Void methode(){

.....

}

class B extends A{

        ....

}

I i = new B();

i.methode();
```

La signature de la méthode doit évidemment être la même que celle promise par l'interface. Dans le cas contraire, la méthode est considérée comme une méthode de la classe et non une implantation de l'interface.

Si une méthode de même signature existe dans la classe mais avec un type de retour différent une erreur de compilation est générée.

4.1.1.4 Variables de type interface et polymorphisme

Bien que la vocation d'une interface soit d'être implémentée par une classe, on peut définir des variables de type interface :

```
Public interface I{ …}

        I i ; // i est une référence à un objet d'une classe implémentant l'interface I
```

Bien entendu, on ne pourra pas affecter à i une référence à quelque chose de type I puisqu'on ne peut pas instancier une interface (pas plus qu'on ne pouvait instancier une classe abstraite !). En revanche, on pourra affecter à i n'importe quelle référence à un objet d'une classe implémentant l'interface I :

```
Public class A implements I{ … }

                I i  = new A(..); // compile
```

De plus, à travers i, on pourra manipuler des objets de classes quelconques, non nécessairement liées par héritage, pour peu que ces classes implémentent l'interface I.

Voici un exemple illustrant cet aspect. Une interface Affichable comporte une méthode affiche. Deux classes Entier et Flottant implémentent cette interface (aucun lien d'héritage n'apparaît ici). On crée un tableau hétérogène de références de type Affichable qu'on remplit en instanciant des objets de type Entier et Flottant.

```
Interface Affichable {void Affiche() ;}

Class Int implements Affichable{

        Private int nombre ;

        Public Int (int n){

                nombre = n ;

        }

        void Affiche(){

System.out.println(« Je suis un nombre entier : «  + nombre) ;

        }

}

public Double implements Affichable{

        Private float nombre ;

        Public Double (float n){

                nombre = n ;

        }

        void Affiche(){

        System.out.println(« Je suis un nombre réel : «  + nombre) ;

        }

}

public class Test{

        public static void main(String args[]){

        Affichable[ ] tab ;

         tab = new Affichable[3];
```

```
        tab[0] = new Int(12);

        tab[1] = new Double(1.2f);

        tab[2] = new Int(42);

        for(int i  = 0 ; i< tab.length ; i ++)

        tab[i].Affiche();

}

}
```

Résultat

```
        Je suis un nombre entier : 12

        Je suis un nombre réel : 1.2

        Je suis un nombre entier : 42
```

Cet exemple est restrictif puisqu'il peut se traiter avec une classe abstraite. Voyons maintenant ce que l'interface apporte de plus.

Remarque

Ici, notre interface a été déclarée avec un droit de paquetage, et non avec l'attribut public, ce qui nous a permis de placer sa définition dans le même fichier source que les autres classes (les droits d'accès des interfaces sont en effet régis par les mêmes règles que ceux des classes). En pratique, il en ira rarement ainsi, dans la mesure où chaque interface disposera de son propre fichier source.

4.1.1.5 Un des principaux intérêts

Une interface permet de mettre en connexion plusieurs hiérarchies de classes, qui à priori n'ont aucuns lien communs entre elles, par l'intermédiaire de méthodes spécifiques.

4.1.1.6 Interfaces et constantes

L'essentiel du concept d'interface réside dans les en-têtes de méthodes qui y figurent. Mais une interface peut aussi renfermer des constantes symboliques qui seront alors accessibles à toutes les classes implémentant l'interface :

```
interface I{
```

```
        static final int MAX = 100 ;

        void f(int n) ;

        void g() ;

}

class A implements I{

        // doit définit les méthodes f et g

        // dans toutes les méthodes de A on aura accès a la constante MAX

}
```

Ces constantes sont automatiquement considérées comme si elles avaient été déclarées static et final. Il doit s'agir obligatoirement d'expressions constantes. Elles sont accessibles en dehors d'une classe implémentant l'interface. Par exemple, la constante MAXI de l'interface I se notera simplement I.MAXI.

4.1.1.7 Dérivation d'une interface

On peut définir une interface comme une généralisation d'une autre. On utilise là encore le mot clé extends, ce qui conduit à parler d'héritage ou de dérivation, et ce bien qu'il ne s'agisse en fait que d'emboîter simplement des déclarations :

```
interface I1{

        static final int MAX = 100 ;

        void f(int n) ;

}

interface I2 extends I1{

        static final int MIN= 20 ;

        void g() ;

}
```

En fait, la définition de I2 est totalement équivalente à :

```
interface I2{

        static final int MAX = 100 ;
```

```
    static final int MIN= 20;

    void f(int n) ;

    void g() ;

}
```

Une interface dérivée hérite de toutes les constantes et méthodes des interfaces ancêtres; à moins qu'un autre champ de même nom ou une autre méthode de même signature soit redéfinie dans l'interface dérivée.

4.1.1.8 Utiliser des interfaces

Une même classe peut implémenter plusieurs interfaces :

```
Class A implements I , J {

    A doit obligatoirement définir les méthodes prévues dans I et J

}
```

Dans ce cas d'utilisation, les interfaces permettent de compenser en grande partie l'absence d'héritage multiple.

4.1.1.9 Redéfinition des champs

Tout comme pour les classes, les champs des interface peut être redéfinis dans une interface dérivée.

```
interface A { int info = 1; }
interface B extends A { int info = 2; }
```

La définition du champ info dans l'interface B masque la définition du champ info de l'interface A. Pour parler du champ info de l'interface A, on le notera A.info.

4.1.1.10 Héritage diamant

Un même champ peut être hérité de plusieurs manière pour une même interface:

```
interface A { char infoA = 'A'; }
interface B extends A { char infoB = 'B'; }
interface C extends A { char infoB = 'C'; }
interface D extends B, C { char infoD = 'D'; }
```

Le champ infoA est hérité par l'interface D de deux manières: une fois par l'interface B et une autre fois par celle de C. Mais il n'existera pas deux champs infoA dans l'interface D; il n'y en aura qu'un

4.1.2 « Héritage multiple » en C#

C #, comme Java, soutient le concept d'une interface qui est apparentée à une classe abstraite pure. De même C # et Java tous les deux permettent seulement la transmission simple des classes mais la transmission multiple (ou l'exécution) des interfaces.

Parfois quand une classe met en application une interface il est possible pour qu'il y ait des collisions portant sur le nom de méthode. Par exemple, une classe de FileRepresentation qui a une interface utilisateur graphique peut mettre en application un IWindow et une interface d'IFileHandler.. Tous les deux classes peuvent avoir une méthode étroitement liée où dans le cas d'IWindow elle permet de fermer la fenêtre de GUI tandis que dans le cas d'IFileHandler il permet de fermer le dossier.C# nous permet de lier des réalisations de méthode aux intefaces spécifiques. Ainsi dans l'exemple, la classe de FileRepresentation aurait de différentes méthodes selon quelles soient appelées si la classe était traitée comme IWindow ou IFileHandler

NOTE : Les méthodes explicites d'interface sont privées et peuvent seulement être consultées si l'objet est moulé au type exigé avant d'appeler la méthode.

```
interface Vehicule{

   void afficheToi();

}

interface Robot{

   void afficheToi();

}

class TestRobot : Robot, Vehicule{

   string model;
   short  annee;
   string nom;


   TestRobot(String nom, String model,short annee){

   this.nom  = nom;
   this.model = model;
   this.annee  = annee;

   }
```

```
    void Robot.IdentifySelf(){

    Console.WriteLine("Mon nom est :  " + this.nom);
    }

    void Vehicule.IdentifySelf(){

      Console.WriteLine("Model:" + this.model +  " Annee:" + this.annee);
    }


    public static void Main(){

    TestRobot tr = new TestRobot("SedanBot", "Toyota", 2001);

    // tr.IdentifySelf(); ERROR

    IVehicle v = (IVehicle) tr;

    IRobot r   = (IRobot) tr;

    v.IdentifySelf();

    r.IdentifySelf();

   }
}
```

4.2 Héritage multiple en C++ et EIFFEL

4.2.1 Héritage multiple en C++

Le C++ offre une implémentation très complète de l'héritage car il propose aussi bien l'héritage simple que multiple ainsi que des options avancées d'héritage sélectif des attributs aussi bien que des méthodes. Signalons également comme aspects positifs le chaînage automatique des constructeurs et destructeurs.

4.2.1.1 Généralités

Voici la syntaxe générale :

NomClasseDérivée : [public|private] ClasseBase1 {, [public|private] ClasseBaseI}$^{1..n}$
déclaration de classe

Le C++ met en œuvre 3 types de dérivation :

- Si on veut que la classe dérivée autorise à ses utilisateurs l'accès aux outils accessibles de la classe de base.

Class Dérivée : **public** Base {…}

- Si on veut que la classe dérivée interdit à ses utilisateurs l'accès aux outils accessibles de la classe de base.

Class Dérivée : **private** Base {…}

- Si on veut limiter à ses propres classes dérivées l'accès aux outils accessibles de la classe de base.

Class Dérivée : **protected** Base {…}

Nous pouvons dériver une classe à partir de plus d'une seule classe de base ce procédé est l'héritage multiple. Cette caractéristique imposante favorise des formes intéressantes de réutilisation de logiciels, bien qu'elle puisse également engendrer une pléiade de problèmes ambigus.

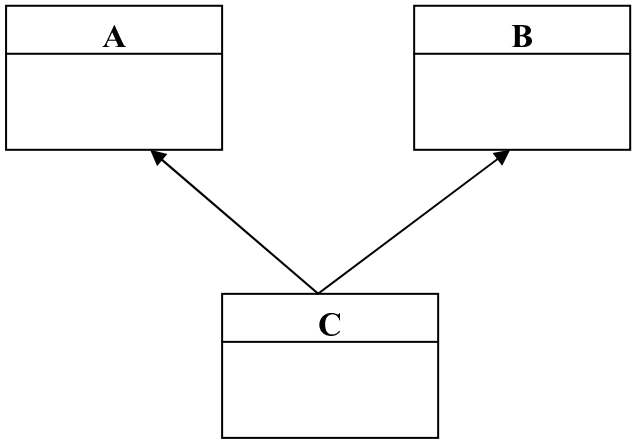
En langage C++, il est possible d'utiliser l'héritage multiple. Il permet de créer des classes dérivées à partir de plusieurs classes de base. Pour chaque classe de base, on peut définir le mode d'héritage. Voici un exemple d'héritage multiple en C++

```
class A {
  public:
    void fa() { /* ... */ }
  protected:
    int _x;
};

class B {
  public:
    void fb() { /* ... */ }
  protected:
    int _x;
};

class C: public B, public A {
  public:
    void fc();
};

void C::fc() {
  int i;
  fa();
  i = A::_x + B::_x; // résolution de portée pour lever l'ambiguïté
}
```



L'héritage multiple entraîne quelques conflits :

- La gestion des membres homonymes hérités des classes parentes.
- La gestion de l'héritage à répétition

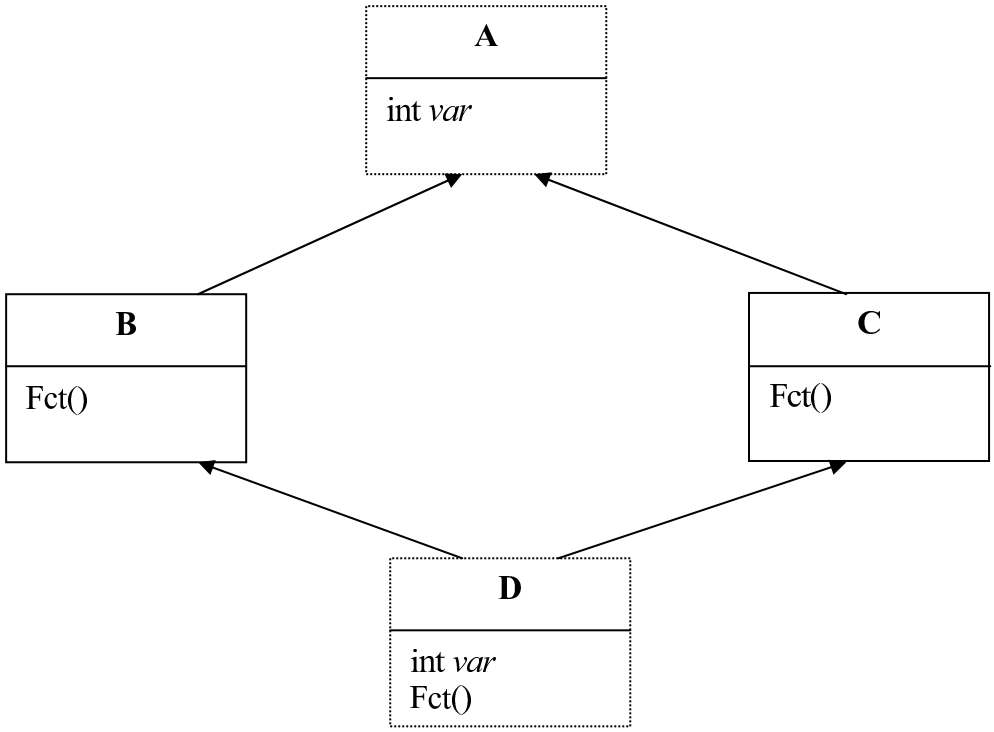
4.2.1.2 Conflits de nom et perte de polymorphisme

Par le biais de l'héritage multiple il est possible qu'une classe hérite d'une même superclasse par deux, ou plusieurs, chemins distincts.

Supposons un objet de la classe D qui contient deux fois les données héritées de la classe de base A, une fois par héritage de la classe B et une autre fois par C. Il y a donc deux fois le membre var dans la classe D.

Ceci pose un premier problème au niveau de la définition du langage. Dans une instance de la classe D, les données et algorithmes définis dans l'ancêtre A doivent-ils être hérités une ou deux fois? Selon la situation que l'on désire modéliser, l'une ou l'autre de ces variantes sera la plus adéquate. Dans le premier cas, si les entités issues de A ne figurent qu'à un exemplaire dans l'instance de D, on parlera d'**héritage commun**; on parlera, par contre d'**héritage répété lorsqu'elles figurent à deux exemplaires.**

Le graphe d'héritage a la forme d'un diamant



L'accès au membre var la classe A se fait en levant l'ambiguïté

```
void main() {
  D un;
  D* deux = new C ;
  un.var = 0;    // ERREUR, ambiguïté
  un.B::var = 1; // OK
  un.C::var = 2; // OK
  deux->fct() ; // ERREUR ambiguïté
  deux->C ::fct() ; //OK
  deux->B ::fct() ; //OK
}
```

Qualifier explicitement la fonction à appeler n'est pas une solution:
Les fonctions virtuelles ne sont plus virtuelles

4.2.1.3 L'héritage virtuel, solution au polymorphisme et à l'héritage répété

C++ est un langage à typage statique, nécessaire pour une programmation fiable, mais offre la possibilité de liaison (édition de lien) dynamique -nécessaire à la programmation objets. Le mécanisme de fonctions virtuelles sert à implanter ce dernier aspect.

Une fonction virtuelle est une fonction membre d'une classe qui peut être redéfinie dans ses sous classes avec la bonne fonction appelée pour chaque instance à l'exécution. Elles sont déclarées **virtual** dans la classe. Ce mot indique que la fonction possède différentes versions dans différentes classes dérivées et qu'il s'agit, pour le compilateur, d'appeler à chaque fois la bonne fonction. Reprenons l'exemple de la fonction affiche() des classes article et ses dérivée.

```
class article{
// ...
virtual void affiche() {
cout <<numero<<" "<< nom<<" " <<pht<<" "<<qte;
}
// ...
};

class audioVisuel: article{
// ...
void affiche(){
article::affiche();
cout<<" "<<dureeGarantie;
}
// ...
};
```

La classe article déclare affiche() comme une fonction virtual, et en fournit une version de base. On doit toujours fournir une version de base pour une fonction virtuelle. La sous classe audioVisuel la redéclare et en fournit une autre version (le mot virtual n'est pas nécessaire à redéclarer pour les sous classes).

Où joue le fait virtuel par rapport au cas non virtuel? C'est lié au polymorphisme et à la liaison dynamique. Considérons les déclarations

```
article *a = new article (...);
audioVisuel *b = new audioVisuel(...);
```

et la séquence de code

```
a->affiche(); (*)b->affiche();
```

Dans chaque cas c'est la fonction correspondante qui est appelée: pour a, la version de affiche() définie dans article, et pour b, la version de affiche() définie dans audioVisuel. Ceci est normal bien sûr, que ce soit virtuel ou non. Mais maintenant, considérons l'affectation

```
a = b;
```

qui fait pointer a , déclaré pointeur article, sur une instance d'une *autre* classe, en l'occurrence une *sous* classe. Le typage statique l'autorise exceptionnellement mais entre classes et sous classes et dans le sens

instance de classe <- instance sous-classe.

C'est le ***polymorphisme***. Car en effet, comme un objet audioVisuel est aussi (*isa*) un article, il peut être désignée par une variable article. Par contre l'inverse est incorrecte: un article n'est pas forcément un genre spécifique comme vêtement ou TV...

Si maintenant on fait
a->affiche();
c'est la fonction affiche() (la bonne) de la classe audioVisuel qui est appelée, ce qui est normal. Alors que dans l'appel précédent (*) c'est la fonction de base qui était appelée. En d'autres termes, la méthode à appliquer à un objet dépend de son instance actuelle. Ce mécanisme n'est possible que par la liaison dynamique. En outre, il faut que les variables désignant les objets soient des pointeurs. C'est pour cela qu'on a déclaré a et b comme des pointeurs.

Si la fonction affiche() n'était par virtual, ce qui était le cas précédemment, cela ne gênerait en rien le programme, seulement, on doit appeler affiche() sur des variables de la classe appropriée et éviter le polymorphisme. Car dans ce cas
a->affiche();
sera toujours traduit (dû au typage statique) par un appel à la méthode de base.

Le polymorphisme et le mécanisme de fonctions virtuelles est intéressant, car il est souvent nécessaire d'avoir une entité qui peut revêtir plusieurs formes (d'où le mot polymorphisme), et donc avoir à appeler des méthodes qui s'appliquent de façon appropriée à chaque cas. Imaginer par exemple un tableau (caddie) d'articles qui peut contenir tout type d'article, ou une fonction à paramètre article qui peut recevoir tout aussi tout article. Les méthodes à appeler sur les éléments du tableau doivent, si redéfinies, correspondre au type de l'élément en cours. C'est ce qui fait la souplesse et le charme de la programmation objets.

Fonctions Virtuelles Pures
Les fonctions virtuelles pures, sont des fonctions membres sans corps déclarées
virtual *entete_fonction* =0;

Par exemple
virtual void f() =0;

Elles caractérisent une classe pour laquelle il n'y aura pas d'instances. Une telle classe est dite *abstraite*. Elle sert à détenir des fonctions, dont le corps devra être fourni par des classes dérivées éventuelles qui, elles, seront instanciées. Justement, l'indicateur =0; pour une fonction virtuelle, diffère sa définition vers des sous classes. L'utilité en est que chacune de ces sous classes pourra donner sa version de la fonction virtuelle.
Imaginons par exemple une classe figure qui englobe plusieurs types de figures géométriques pour lesquelles le calcul de la surface, le dessin etc. est différent selon que c'est un carre, un cercle ou tout autre figure. On pourra écrire:

```
class figure{
virtual void rotation(float)=0;
virtual void afficher()=0;
virtual float surface()=0;
```

```
// ...
};
```

On voit bien que cette classe n'a pas besoin d'instance, cela n'aurait aucun sens. Par contre, on peut en avoir pour des classes dérivées:

```
class rectangle : figure {
rectangle(float hg, float bd){ ... }
virtual void rotation(float alpha) {
// rotation d'un rectangle
}
virtual void afficher() //...
// ...
};
class cercle : figure {
// ...
};
```

Les classes cercle et rectangle donneront chacune leur version des fonctions virtuelles. C++ exige que *toutes* les fonctions virtuelles données dans la classe mère soient implantées dans les classes dérivées, car sinon il ne pourra pas instancier ces dernières (il y aurait une caractéristique héritée dont il ne connaît pas la réalisation).

Remarque: Cela n'empêche pas de mettre des fonctions membres normales dans une classe avec fonctions virtuelles. Elles seront héritées comme a l'accoutumé. Néanmoins cela diminuerait la caractéristique abstraite de la classe.

4.2.2 Héritage multiple en EIFFEL

4.2.2.1 Généralités

Comme C++, Eiffel met en œuvre l'héritage multiple. Cependant les difficultés liées à la transmission multiple sont résolues de manière très différente. Eiffel permet à l'utilisateur d'adapter les membres hérités de la classe mère selon les besoins de la classe fille.

Voici la syntaxe générale pour effectuer de l'héritage :

```
Class Polygone
        inherit Figure
        end ;
feature …
end ;
```

Après la clause inherit on peut ajouter un nombre illimité de classes mères pour réaliser de l'héritage multiple.

Après chaque nom de classe mère on peut rajouter des clauses d'adaptation facultative. Ces options doivent impérativement apparaître dans l'ordre suivant :

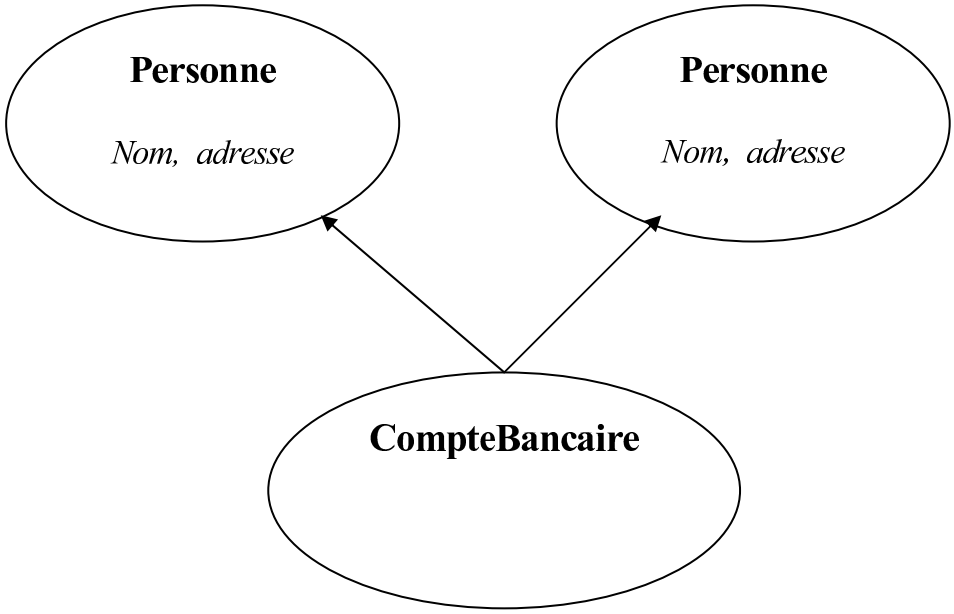
- **rename** permet à l'utilisateur de modifier les membres hérités. Cette option est particulièrement utile quand des conflits de noms se présentent (voir plus bas).
- **export** permet à l'utilisateur de changer le mode de dérivation des membres hérités.
- **undefine** utiliser pour résoudre les conflits de noms lors d'une transmission multiple.
- **redefine** pour redéfinir le membre hérité.
- **select** pour déterminer la méthode à appliquer quand deux méthodes de même nom sont héritées.

Nous avons vu dans le langage C++ que l'héritage multiple posé des problèmes de conflits de noms. Nous allons maintenant nous intéresser à la manière dont Eiffel traite ce problème majeur dans la dérivation multiple.

4.2.2.2 Gestion de l'héritage répété

Lors d'une transmission multiple, une classe peut hériter du même parent plusieurs fois directement ou par différents chemins. Ce qui conduit à produire de l'héritage répété.
Considérons l'exemple suivant :

Un compte bancaire peut être commun à un couple, 2 personnes ont un numéro de compte et une seule adresse en revanche ils ont des noms différents. Si on a une classe *CompeBancaire* celle-ci hérite deux fois d'une même classe *Personne*. Contrairement au C++ on peut dériver plusieurs fois d'une même classe On alors la situation suivante représenté sous forme de graphe :



On remarque alors que la classe *CompteBancaire* hérite deux fois de l'attribut nom et adresse. Comment Eiffel gère t-il ce type de problème ?

Eiffel transforme les membres hérités répétés (ceux qui n'ont pas été renommés dans les classes ancêtres) en membre simple. Si un membre n'est pas modifié entre une classe ancêtre et une classe fille alors cette dernière a un membre (fusionné) simple. Des membres dupliqués qui sont hérités sans changement, sont fusionnés pour obtenir un membre simple.

Dans notre exemple, les attributs noms et adresse qui sont hérités deux fois deviennent des membres simples dans la classe *CompteBancaire*. Hors ce n'est pas notre intérêt nous voulons bien un seul attribut adresse par contre il nous faut deux noms différents. Eiffel permet d'éviter la fusion des membres, il suffit d'introduire la clause **rename** suivit des membres à renommer.
Voici le code de notre exemple :

```
class  Personne
        feature
                nom :STRING
                adresse :STRING
end;



class CompteBancaire
        inherit
        Personne rename nom as nom_1;
        Personne rename nom as nom_2 ;
        feature
                num_compte : INTEGER
                deposer(x : REAL) is
                do
-- …
                end;
retirer(x : REAL) is
do
     -- …
end;
        -- …
        end;
```

Eiffel autorise l'héritage **répété**, il suffit alors de renommer les membres répétés. Si on désire faire de l'héritage en **commun**, Eiffel fusionne de manière automatique tous les membres hérités non renommés par les classes ancêtres.

4.2.2.3 Gestion du polymorphisme

Prenons l'exemple suivant :

```
class B
```

```
        inherit A
                rename
                        m as k
end ;
feature …
end ;
```

Si une variable x est déclaré du type A mais de type réel B. Si on a x.m alors c'est la méthode k qui sera exécuté.

Si maintenant on veut redéfinir la méthode m dans B en utilisant la version de A. On aurait pu écrire le code suivant :

```
class B
        inherit A
                rename
                        m as old_m
                redefine m
                end ;
feature
        m(…) is
                do … old_m …end ;
```

Le code ci-dessus peut présenter des soucis lors de l'utilisation du polymorphisme. C'est la routine old_m qui sera appliqué ici et non m.

Pour faire du polymorphisme il faut hériter deux fois de la classe A et sélectionner la routine qui sera redéfini. Voici le code qu'il aurait fallu écrire :

```
class B
        inherit A
                rename
                        m as old_m
        inherit A
                redefine m
                select m
                end ;
feature
        m(…) is
                do … old_m …. End ;
```

Quand deux versions différentes de la même méthode sont héritées dans une classe, alors quelle méthode sera appliquée lors d'un polymorphisme? Pour décider la méthode à appliquer, il faut introduire la clause **select.**

5 CONCLUSION

C# met en œuvre comme Java l'héritage multiple d'interface et simple de classes. Parmi les défauts, nous remarquons l'impossibilité depuis le code de savoir si on hérite d'une classe ou d'un interface. Pour Microsoft c'est le nom qui remplie ce rôle. Ainsi que l'obligation de déclarer les fonctions pouvant être surcharger

En JAVA, l'héritage est simplifié au maximum (surcharge implicite, distinction entre interface et classe). L'héritage multiple d'interface est relativement simples mais leur utilisation relève de la conception. A noter également il n'y a qu'un seul mode d'héritage(implicite) à la différence de C++ où trois modes sont disponibles(public, protected et final).

Le mécanisme d'heritage multiple proposée par C++ est trop rudimentaire et complexe en ce qui concerne le renommage, la redéfinition et la visibilité des méthodes héritées pour etre utilisable (d'ailleurs, de nombreuses règles de programmation C++ recommandent de ne pas utiliser l'heritage multiple). Les méthodes n'étant pas virtuelles par défaut, il faut précéder du mot clé virtual les méthodes de la classe mère pour réaliser du polymorphisme mais encore faut-il savoir quand mettre des méthodes virtuelles.

Eiffel , quant à lui maîtrise le puissant potentiel de l'héritage multiple par un mécanisme plus clair et plus complet. L'héritage répété est possible comme l'héritage commun, le renommage permet d'éviter tout conflits de noms.