Université Nice SOPHIA ANTIPOLIS

Travail d'Etude
Licence d'Informatique
16 Juin 2003

BOUCHEREZ Kévin
SAUSSIER Pascal

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

Encadrement de ce travail par : Pierre Crescenzo

SOMMAIRE

1 Introduction 4

2 L'héritage Simple 7

2.1 Le C# 7

2.1.1 L'héritage de classes et sa syntaxe 7
2.1.2 Les niveaux d'accessibilité 7
2.1.3 Les constructeurs 8
2.1.4 Le Polymorphisme 10
2.1.4.1 Définition 10
2.1.4.2 Masquer ou écraser une méthode de la classe de base. 11
2.1.4.3 Construire des classes abstraites. 14
2.1.4.4 Empêcher qu'une classe puisse être transformée en sous classe. 14
2.1.5 Interface: 15
2.1.5.1 Définition 15
2.1.5.2 Héritage et Interface 16
2.1.5.3 Rencontrer une interface abstraite. 17

2.2 JAVA, C++ et Eiffel 18

2.2.1 L'héritage de classes et sa syntaxe 18
2.2.1.1 En Java 18
2.2.1.2 En C++ 18
2.2.1.3 En Eiffel 18
2.2.2 Les niveaux d'accessibilité 18
2.2.2.1 En Java 19
2.2.2.2 En C++ 19
2.2.2.3 En Eiffel 20
2.2.3 Les constructeurs 20
2.2.3.1 En Java 20
2.2.3.2 En C++ 21
2.2.3.3 En Eiffel 21
2.2.4 Le Polymorphisme 22
2.2.4.1 Définition 22
2.2.4.1.1 En Java 22
2.2.4.1.2 En C++ et en Eiffel 22
2.2.4.2 Masquer ou écraser une méthode de la classe de base 22
2.2.4.2.1 En Java 22
2.2.4.2.2 En C++ 23
2.2.4.2.3 En Eiffel 23
2.2.4.3 Construire des classes abstraites 23
2.2.4.3.1 En Java 23
2.2.4.3.2 En C++ 23
2.2.4.3.3 En Eiffel 24
2.2.4.4 Empêcher qu'une classe puisse être transformée en sous classe 24
2.2.4.4.1 En Java 24
2.2.4.4.2 En C++ 24
2.2.4.4.3 En Eiffel 24
2.2.5 Interface 24
2.2.5.1 En Java 24
2.2.5.2 En C++ et en Eiffel 25

3 L'héritage Multiple 26

3.1 Introduction 26

3.2 En C++ 29

3.2.1 Héritage commun et héritage répété 29

3.2.2 Comment accéder à quoi ? 31

3.3 En Eiffel 34

3.3.1 Clauses de renommage, redéfinition 34

3.3.2 Les exceptions 35

3.4 En Java et en C# 39

4 Conclusion 40

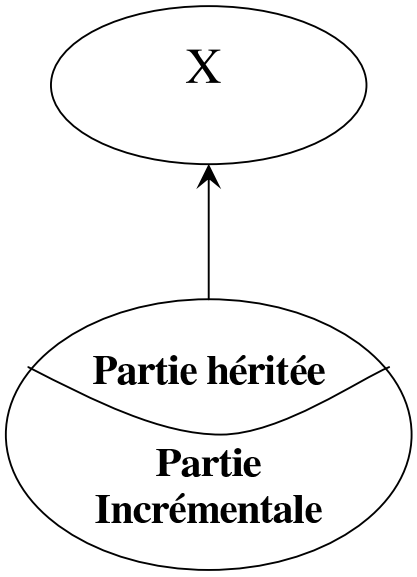
1 Introduction

L'héritage est une relation entre différentes classes permettant de définir une nouvelle classe en se basant sur des classes existantes. Le concept d'héritage est l'un des fondements de la programmation orientée objet (POO).

Par exemple, un fils demande à son père "Qu'est-ce que c'est qu'un canard ?", le père répond : "C'est un oiseau qui fait coin-coin". Cet exemple bien qu'amusant révèle une quantité considérable d'informations sur l'héritage. Le fils sachant ce qu'est un oiseau, il sait maintenant qu'un canard possède toutes les propriétés d'un oiseau plus la propriété supplémentaire : "faire coin-coin".

Il en va de même pour les langages objet. Lorsqu'une classe Y hérite directement ou indirectement d'une classe X, on dit que la classe Y est une classe descendante ou dérivée de X. On dit encore que X est l'ancêtre de Y. Si une telle relation d'héritage existe entre X et Y, alors Y se compose de deux parties:

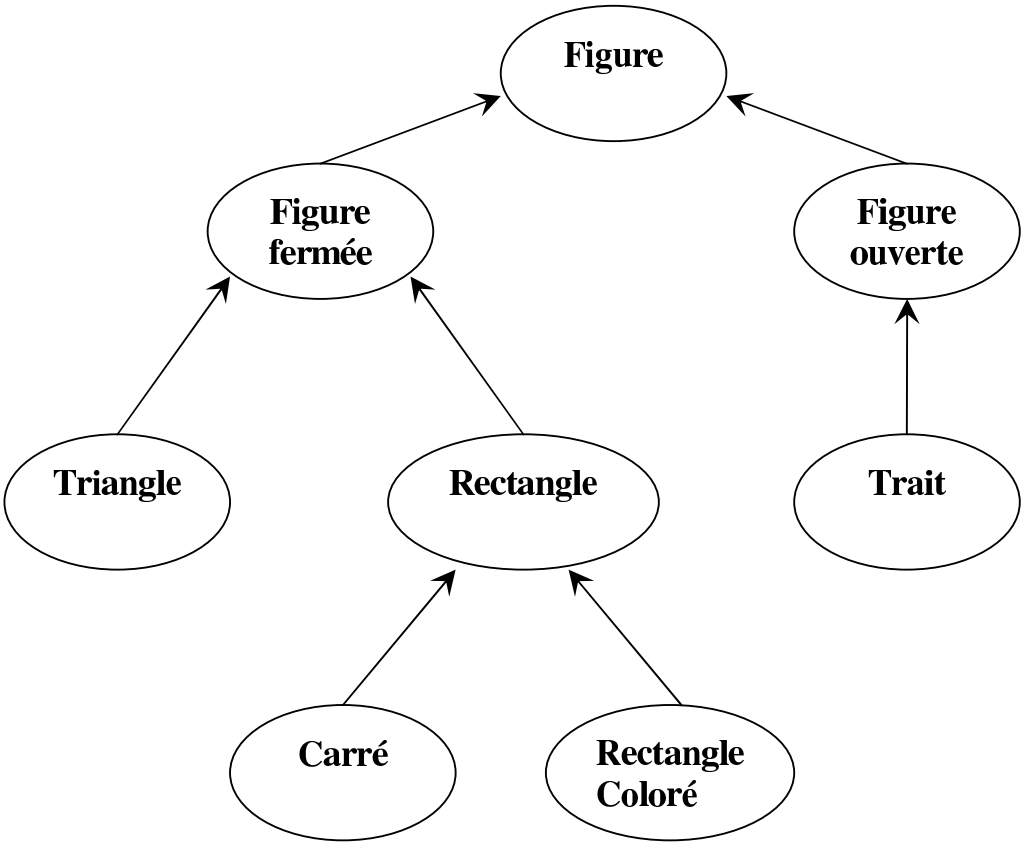
- La partie héritée ("C'est un oiseau")
- La partie incrémentale ("Il fait coin-coin")



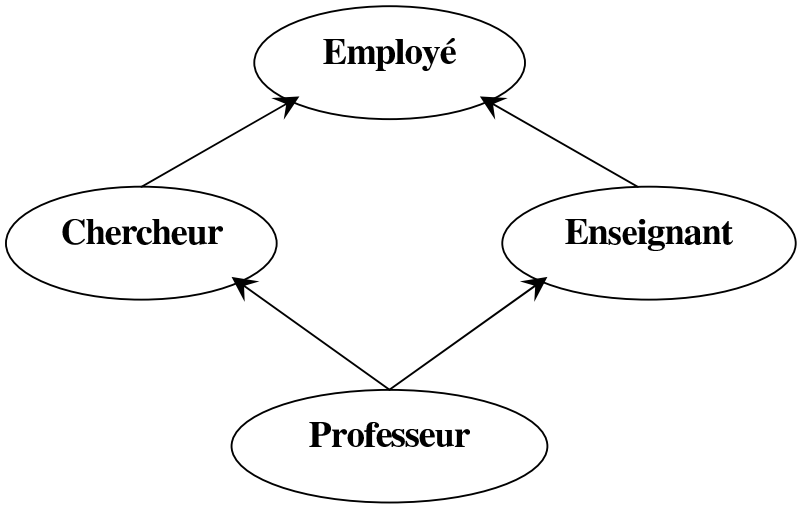
La relation d'héritage peut être vue comme une relation de type "Est_Un", car la classe Y dérivée de X hérite de tout de X et *est* donc *un* X. Mais Y est plus qu'un X, car lorsqu'on crée une nouvelle classe par héritage, on lui adjoint généralement d'autres attributs et méthodes (ici, "faire coin-coin").

L'héritage peut donc être considéré à la fois comme une relation d'extension et comme une relation de spécialisation. Si l'on voit une classe comme un type, alors on parlera de spécialisation. Un rectangle est un type de figure, mais un rectangle coloré est plus spécialisé, plus particulier qu'un rectangle. C'est un processus de spécialisation.

Si l'on voit cette classe comme un module, par exemple si Y hérite de X, alors le module Y contient les méthodes et les champs de X en plus de ses extensions propres. C'est un processus d'extension.



Il se peut aussi qu'une classe hérite de plusieurs classes. Dans ce cas, on parle d'héritage multiple. Toutefois, ce type d'héritage est différent selon le langage étudié, et certains langages imposent même de fortes contraintes et restrictions concernant son utilisation.



Dans un premier temps, nous étudierons les différents mécanismes de l'héritage simple en s'appuyant sur le langage C#. Nous verrons ensuite les différences et les similitudes avec Java, C++ et Eiffel.

Ensuite, nous traiterons des mécanismes de l'héritage multiple avec comme support le langage C++. Nous verrons ensuite les différences et les similitudes principalement avec Eiffel. Puis un bref descriptif des possibilités offertes par Java et C# pour simuler l'héritage multiple. Ces deux langages ont en effet choisi de ne pas traiter l'héritage multiple, le considérant comme trop complexe.

2 L'héritage Simple

2.1 Le C#

2.1.1 L'héritage de classes et sa syntaxe

Un exemple (la relation EST_UN):

public class Voiture : Véhicule // Voiture hérite de la classe Véhicule.

Par conséquent la classe Voiture hérite de la classe mère Véhicule, la classe fille Voiture peut donc utiliser les méthodes de la classe Véhicule. L'identificateur d'héritage est le symbole *" : "*, le membre de gauche (la classe fille) est la classe descendante, le membre de droite est donc la classe ascendante (ou classe mère, ou super-classe).

2.1.2 Les niveaux d'accessibilité

C# offre différents niveaux d'accessibilité pour la gestion des attributs et des méthodes de ses classes :

- Un membre défini *public* est accessible par toutes les classes du programme.
- Un membre défini *private* n'est accessible que par la classe dans laquelle il est déclaré.
- Un membre défini *protected* n'est accessible que dans la classe où il est déclaré ainsi que dans toutes ses sous-classes (ou classes fille).
- Un membre défini *internal* est accessible par toutes les classes du même espace de noms (essentiellement, par tout groupe de modules C# que l'on aura spécifié pour cela, c'est-à-dire tous les modules que l'on aura écrits pour le programme. Un espace de noms en C# ressemble un peu au *package* de JAVA).

- Un membre *internal protected* est accessible par la classe dans laquelle il est déclaré et toutes ses sous-classes, ainsi que par les classes du même module.

Remarque: Nous n'avons pas l'obligation de créer un seul fichier par classe publique comme en JAVA. On peut mettre toute les classes en *public* dans un même fichier (car en Java la classe qui est *public* doit être son nom de fichier. En Java s'il y a plusieurs classes dans un même fichier il ne peut y en avoir qu'une seule qui soit *public*).

2.1.3 Les constructeurs

Lors de la fabrication d'un objet correspondant à une classe, une de ses méthodes est automatiquement appelée : le constructeur. Ce constructeur doit porter le même nom que la classe à laquelle il appartient. Il peut posséder un nombre quelconques d'arguments. S'il n'est pas défini par le programmeur, un constructeur par défaut est donné à la classe, celui-ci n'a pas de paramètres.

Un exemple :

```
using System;
namespace HeritageEtConstructeurs
{
        public class Class
        {
                public static int Main(string[] args)
                {
                        Console.WriteLine("Création d'un objet Vehicule");
                        Vehicule ve = new Vehicule();
                        Console.WriteLine("Création d'un objet Voiture ");
                        Voiture vo = new Voiture();
                        Console.WriteLine("Fin du main…");
                        return 0;
                }
        }
        public class Vehicule
        {
                public Vehicule()  // Création d'un constructeur pour la classe Vehicule
                {
                        Console.WriteLine("Construction de Vehicule");
                }
        }
        public class Voiture : Véhicule
        {
                public Voiture()  // Création d'un constructeur pour la classe Voiture
                {
```

```
                Console.WriteLine("Construction de Voiture");
            }
        }
}
```

Ce programme donne le résultat suivant :

```
Création d'un objet Vehicule
Construction de Vehicule
Création d'un objet Voiture
Construction de Vehicule
Construction de Voiture
Fin du main…
```

Lors de la fabrication d'un objet qui possède une super-classe, il se pose alors la question du devenir du constructeur de la classe mère. Ce constructeur est en fait exécuté avant le constructeur de la classe fille. Mais si le constructeur de la classe mère possède des arguments, il faut que la classe fille puisse lui en transmettre quelques uns. Ceci se fait par l'intermédiaire du mot clé *" base "*.

Un exemple :

Imaginons que la classe Vehicule ait un attribut poids, lors de l'appel au constructeur de Voiture, il faut retransmettre l'information poids au constructeur de Vehicule.

```
public class Vehicule
{
        public int poids;

        public Vehicule(int p)  // Création d'un constructeur pour la classe Vehicule
        {
                poids = p;
                Console.WriteLine("Construction de Vehicule de poids {0}", poids);
        }
}

public class Voiture : Vehicule
{
        public Voiture(int poids) : base(poids)
        {
                Console.WriteLine("Construction de Voiture");
        }
}
```

Si pour le main on a :

```
public static int Main(string[] args)
{
        Console.WriteLine("Création d'un objet Vehicule");
        Vehicule ve = new Vehicule(1000);
```

```
        Console.WriteLine("Création d'un objet Voiture ");
        Voiture vo = new Voiture(2000);
        Console.WriteLine("Fin du main…");
        return 0;
}
```

On obtiendrait :

```
        Création d'un objet Vehicule
        Construction de Vehicule de poids 1000
        Création d'un objet Voiture
        Construction de Vehicule de poids 2000
        Construction de Voiture
        Fin du main…
```

2.1.4 Le Polymorphisme

2.1.4.1 Définition

Le pouvoir du mécanisme d'héritage repose sur le fait qu'une sous-classe n'est pas obligée d'hériter à l'identique de toutes les méthodes de la classe de base. Une sous-classe peut hériter de l'essence des méthodes de la classe de base tout en réalisant une implémentation différente de leurs détails.

Un exemple :

Une Voiture *est un* Vehicule, elle possède tous les attributs et les comportements de Vehicule. Pourtant une Voiture n'aura pas forcément la même façon de prendre un virage qu'un Vehicule, mais une Voiture reste un Vehicule.

```
public class Test
{
    public static void coloration(Vehicule v, Color c)//En supposant que Color existe en C#
        {
            // Cette procédure colorie un Véhicule de couleur c
        }

        public static void main(string[] args)
        {
            Vehicule v = new Vehicule("toto");.
            Voiture a = new Voiture("Ferrari");
            coloration(v, Color.red);       // OK
            coloration(a, Color.blue);      // OK car une Voiture EST_UN Vehicule
        …
        }
}
```

Il serait toutefois utile dans certains cas de pouvoir discerner si un objet est un Vehicule ou est une Voiture. Puisque l'on ne met pas forcément le type d'objet en question "à cause" du polymorphisme, il existe le mot-clé *is* qui permet de savoir si v de type Véhicule est bien de type Véhicule ou autre.

Un exemple:

```
public void coloration(Vehicule v, Color c)
{
        if (v is Vehicule)
        {
                Console.WriteLine("v est un Vehicule");
                v.affiche();
                …
        }
        else if (v is Voiture)
        {
                Console.WriteLine("v est une Voiture");
                Voiture a = (Voiture) v;
                a.affiche();
                …
        }
        else
        {
                Console.WriteLine("v est inconnu");
        }
}
```

Inconvénient s'il l'on rajoute de nouvelles classes filles (avec comme classe mère, la classe Vehicule), par exemple la classe Avion, ou Moto il faudra ne pas oublier de rajouter des "else if" dans la procédure coloration.

2.1.4.2 Masquer ou écraser une méthode de la classe de base.

Toutefois, une Voiture n'aura pas forcément la même façon de prendre un virage qu'un Vehicule comme nous l'avons dit plus haut. Il faudrait pouvoir donner un comportement différent à une Voiture qu'à un Vehicule. Il faudrait donc **surcharger** une méthode héritée.

Nous avons la possibilité de redéfinir une méthode de classe. Une méthode d'une classe peut surcharger une autre méthode de la même classe en ayant des arguments différents. En C# on peut même avoir deux fonctions portant le même nom qui diffèrent seulement par le type retourné. De même, une méthode peut aussi surcharger une méthode de sa classe de base (c'est-à-dire sa classe mère).

surcharger ⇔ redéfinir ⇔ cacher une méthode

Dans le cas ou l'on redéfinirait accidentellement une méthode de la classe de base (donc qui a la même signature) :

Un exemple:
Dans la classe mère Vehicule une fonction virage().
Dans la classe fille Voiture une fonction virage().

C# sait détecter le problème à la compilation, il génère un avertissement en disant qu'il est probable qu'il y ait un problème. La solution apportée consiste en le mot clé *new*.

```
new public int virage()
{
        // corps de la méthode
}
```

```
public class Vehicule
{
        ….
  public String affiche()
  {
    Console.WriteLine("Je suis un Vehicule");
   }
}
```

```
public class Voiture : Vehicule
{
        ….
   new public String affiche()
   {
     Console.WriteLine("Je suis une Voiture");
    }
}
```

Ainsi, le mot-clé *new* indique au compilateur que cette méthode est redéfinie intentionnellement.

Remarque: Ce mot-clé new n'a rien à voir avec l'utilisation du même mot clé pour créer un objet.

C# permet aussi une autre utilisation du mot clé *base*:
base.nomDeLaFonction() appelle la fonction *nomDeLaFonction*() qui se trouve dans la classe mère.

Un autre exemple :

```
new public int virage()    // On est donc ici dans la classe fille
{
        base.virage();
}
```

Ainsi, dans cette fonction je veux appeler la méthode virage() de la classe mère Avion.

Pour reprendre l'exemple de coloration de Vehicule de tout à l'heure, nous ne voudrions pas connaître tous les différents types de Vehicule. Nous voudrions que ce soit aux programmeurs qui utilisent coloration(…) de connaître leurs types de Vehicule.

C# permet de prendre les décisions sur les méthodes à invoquer en fonction du type de l'objet à l'exécution.

Par exemple, pour dire à C# de faire le choix à l'exécution de la version de affiche() à utiliser, il faut marquer la fonction de classe de base avec le mot-clé *virtual*, et la fonction de chaque sous-classe avec le mot-clé *override*.

Ce qui donne le résultat suivant d'après l'exemple précédent:

Exemple:

```
public class Vehicule
{
        ….
  virtual public String affiche()
  {
    Console.WriteLine("Je suis un Vehicule");
   }
}
```

```
public class Avion : Vehicule
{
        ….
    override new public String affiche()
    {
      Console.WriteLine("Je suis un Avion");
     }
}
```

```
public void coloration(Vehicule v, Color c)
        {
                if (v is Vehicule)
                {
                        Console.WriteLine("v est un véhicule");
                        v.affiche();
                        …
                }
                else if (v is Avion)
                {
                        Console.WriteLine("v est un avion");
                        v.affiche();   // permet de faire cela grâce au mot clé virtual et override.
                        …
                }
                else
                {
                        Console.WriteLine("v est inconnu");
                }
        }
```

Ainsi, v.affiche() suffit dans n'importe quelles parties du *if*.

2.1.4.3 Construire des classes abstraites.

En se penchant un peu plus sur l'exemple du Vehicule, on se rend compte qu'il serait stupide d'écrire :

```
Vehicule v = new Vehicule(…);
```

En effet, un Vehicule correspond à tout les attributs et comportements communs à tous les Véhicules mais ne représente pas un objet concret tel qu'une Voiture, une Moto ou un Avion, mais cette classe s'apparenterait plutôt à un concept.

Il faudrait donc pouvoir l'interdire, la solution consiste à rendre cette classe abstraite grâce au mot clé *abstract*.

Ainsi, si la classe Vehicule est abstraite et que l'on écrit :

```
Vehicule v  = new Vehicule(…);
```

Cela va générer une erreur à la compilation.

Une classe ainsi que des méthodes peuvent être abstraites, ainsi les méthodes déclarées comme abstraites (mot-clé *abstract*) ne doivent pas avoir d'implémentation.

Le fait qu'une des méthodes d'une classe soit déclarée abstraite entraîne le fait que cette même classe est abstraite.

Une classe peut être déclaré abstraite qu'elle comporte ou non des membres abstraits; mais une classe ne peut être concrète que lorsque toutes ses méthodes abstraites ont été redéfinies par des méthodes réelles. Une classe abstraite est automatiquement virtuelle.

2.1.4.4 Empêcher qu'une classe puisse être transformée en sous classe.

Vous pouvez très bien décider que vous ne voulez pas que les générations futures de programmeurs puissent étendre une de vos classes. Dans ce cas vous pouvez la verrouiller en utilisant le mot-clé *sealed*.

Une classe scellée ne peut donc être utilisée comme classe de base pour une autre chose.

2.1.5 Interface:

2.1.5.1 Définition

L'*interface* de C# implémente également une autre association, tout aussi importante: la relation PEUT_ETRE_UTLISE_COMME.

Une description d'interface ressemble beaucoup à une classe sans données dans laquelle toutes les méthodes seraient abstraites.

Un exemple:

```
interface IRecordable
{
        void PrendreUneNote(string sNote);
}
```

La méthode PrendreUneNote (…) est écrite sans implémentation.
Le mot-clé *interface* remplace le mot-clé *class*. Une interface ne contient aucune définition de membre donnée. Les mots-clés *public* et *virtual* ou *abstract* sont implicites et ne sont donc pas nécessaires.

Un exemple:

```
public class Voiture : IFrein
```

Par convention on met un I devant l'interface car cela s'écrit pareil que lors d'un héritage classique, il n'y a pas de distinctions entre l'héritage d'interfaces et l'héritage de classes.

```
public class Stylo : IRecordable
{
        …
}

public class PDA : IRecordable
{
        …
}

public class Portable : IRecordable
```

```
{
        …
}

public class Class1
{
        static public void RecordShoppingList(IRecordable recordObject)
        {
                string sList = GenerateShoppingList();
                recordObject.PrendreUneNote(sList);
        }

        public static void main(string[] args)
        {
                PDA pda = new PDA();
                RecordShoppingList(pda);   // ou Stylo ou Portable.
        }
}
```

Donc, finalement on peut mettre n'importe quelles classes qui implémentent Irecordable, n'importe laquelle de ces classes PEUT_ETRE_UTILISEE_COMME un Recordable.

2.1.5.2 Héritage et Interface

Une interface peut "hériter" des méthodes d'une autre interface. Il ne s'agit en fait pas d'un véritable héritage, même s'il en a pourtant l'air.

```
public interface IComparable
{
        int comparerA();
}

public interface ICompare : IComparable
{
        int getValeur();
}
```

L'interface ICompare hérite de IComparable l'exigence d'implémenter la méthode comparerA(). A cela, elle ajoute l'exigence d'implémenter getValeur(). Un objet ICompare peut être utilisé comme un objet IComparable, car par définition, le premier implémente les exigences du second. Toutefois, il ne s'agit pas là d'un héritage complet au sens C#, orienté objet, de ce terme. Le polymorphisme n'est pas possible.

De plus, les relations de constructeurs ne s'appliquent pas.

2.1.5.3 Rencontrer une interface abstraite.

Afin d'implémenter une interface, une classe doit redéfinir chaque méthode de celle-ci. Toutefois, une classe peut redéfinir une méthode d'une interface par une méthode abstraite(naturellement, bien sûr, une telle classe est abstraite).

En résumé :

Dans la classe qui implémente l'interface, on doit redéfinir toutes les méthodes, même celles dont on ne se sert pas.

Par conséquent, pour toute les méthodes qui ne nous intéressent pas il y a 2 choix à adopter :

1) On écrit rien dans la méthode que l'on redéfinit

Exemple:

```
public int getToto()
{
        return 0;
}
```

2) On déclare cette méthode en une méthode *abstract* donc pas besoin de redéfinir :

Exemple:

```
abstract public int getToto();
```

Inconvénient:

La classe dans laquelle on fait cela doit devenir une classe abstraite.

2.2 JAVA, C++ et Eiffel

2.2.1 L'héritage de classes et sa syntaxe

2.2.1.1 En Java

Un exemple (la relation EST_UN):

public class Voiture **extends** Vehicule // Voiture hérite de la classe Vehicule.

L'identificateur d'héritage est le symbole *" extends "*, le membre de gauche (la classe fille) est la classe descendante, le membre de droite est donc la classe ascendante (ou classe mère, ou super-classe).

2.2.1.2 En C++

class Voiture **: public** Vehicule // Voiture hérite de la classe

L'identificateur d'héritage est le symbole *" : "*, le membre de gauche (la classe fille) est la classe descendante, le membre de droite est donc la classe ascendante (ou classe mère, ou super-classe), précédé par un mot clé définissant le type de la relation d'héritage avec la classe mère (dans l'exemple **public**).

2.2.1.3 En Eiffel

class Voiture
inherit
    Vehicule // Voiture hérite de la classe

L'identificateur d'héritage est le symbole *" inherit "*, le membre de gauche (la classe fille) est la classe descendante, le membre de droite est donc la classe ascendante (ou classe mère, ou super-classe).

2.2.2 Les niveaux d'accessibilité

2.2.2.1 En Java

JAVA offre différents niveaux d'accessibilité pour la gestion des attributs et des méthodes de ses classes :

- *public* toutes les classes sans exception y ont accès.
- *private* seule la classe dans laquelle il est déclaré y a accès (à ce membre ou constructeur).
- Un membre défini *protected* n'est accessible que dans la classe où il est déclaré ainsi que dans toutes ses sous-classes (ou classes fille) mais, *protected* autorise en plus à la différence de C# l'utilisation par les classes du même paquetage que la classe où est défini le membre ou le constructeur.

Si rien n'est spécifié, seules les classes du même paquetage que la classe dans lequel il est déclaré y ont accès (un paquetage étant un regroupement de classes).

Remarque :
La classe publique doit être du même nom que le fichier dans lequel elle se trouve. Si plusieurs classes se trouvent dans le même fichier, il ne peut donc y avoir qu'une seule classe publique portant le nom du fichier.

2.2.2.2 En C++

C++ offre différents niveaux d'accessibilité pour la gestion des attributs et des méthodes de ses classes :

- *public* toutes les classes sans exception y ont accès.
- *private* seule la classe dans laquelle il est déclaré a accès (à ce membre ou constructeur).
- Un membre défini *protected* n'est accessible que dans la classe où il est déclaré ainsi que dans toutes ses sous-classes (ou classes fille) , ou classes amies.

En plus de cela, C++ permet de définir le type de la relation d'héritage avec la classe mère, qui peut être soit **public**, **protected** ou **private**.

En résumé :

Mode de dérivation	Statut du membre dans la classe ancêtre	Statut du membre dans la classe dérivée
private	private protected public	inaccessible private private
protected	private protected public	inaccessible protected protected
public	private protected public	inaccessible protected public

Il est possible aussi de définir un héritage partiellement public, on peut rendre son niveau d'accès initial (protected ou public), à un membre hérité non publiquement grâce à une using-declaration :

```
class Tableau
{
        public :
                int taillemax();
                …
};
```

```
class Pile : private Tableau
{
        public:
                using Tableau::taillemax;
                ...
};
```

Dans la classe Pile qui hérite de la classe Tableau, le mot clé **private** rend impossible l'accès à la méthode taillemax() de la classe Tableau, effet qui est annulé grâce à "**using** Tableau::taillemax;" .

2.2.2.3 En Eiffel

Par défaut, toute méthode est publique et accessible par n'importe quelle classe. Les attributs sont quand à eux en lecture seule pour les autres classes, l'affectation leur est interdite. Il est toutefois possible de définir des membres de manière **private,** il suffit pour cela de déclarer ces membres après une clause **feature {NONE}**.

2.2.3 Les constructeurs

2.2.3.1 En Java

Le constructeur doit avoir comme en C# le même nom que la classe à laquelle il appartient. Il peut aussi y avoir plusieurs constructeurs avec différents paramètres. Il existe aussi un constructeur par défaut sans paramètre.

Comme en C#, il est possible d'accéder au constructeur de la classe mère grâce au mot clé **super(…). super(…)** doit être la première instruction du constructeur de la classe fille. Si **super** n'apparaît pas en première instruction, un appel implicite est fait au constructeur sans paramètre de la classe mère.

Si un constructeur de **A** est déclaré **protected**, ce constructeur peut être appelé depuis un constructeur d'une classe fille **B** par un appel à **super**() mais B ne peut créer d'instance de **A** par **new A**()

2.2.3.2 En C++

Lors de la définition d'un constructeur pour une classe dérivée, la liste d'initialisation doit mentionner les constructeurs choisis :

- Pour l'initialisation de chacune des bases
- Pour l'initialisation de chacun des objets membres spécifiques à la classe dérivée

Un exemple :

```
class Employe
{
        public :
                Employe(int n=0) : num(n)
                {std::cout << "initialisation Employe
                \n";}
                int num;
                …
};
```

```
class Cadre : public Employe
{
        public:
                Cadre(int n, int e, int nb) :
                        Employe(n), echelon(e)
                {std::cout << "initialisation
                Cadre\n";}
                int echelon;
                ...
};
```

2.2.3.3 En Eiffel

Les routines de création de classes sont déclarée grâce au mot clé **creation**, puis lorsqu'un objet est créé grâce à " !! ", soit la classe n'a pas de méthodes de **creation**, et on peut créer l'objet e par exemple en faisant !!e, soit la classe a au mois une méthode de création et il faut la spécifier explicitement par exemple : !!e.init(123).

Un exemple :

```
local
        e : TOTO
do
        !!e.init(123)
end
```

Un objet e de type TOTO est créé, initialisé, attaché à e et ensuite la méthode de création init est appelée pour e.

Il est aussi possible de créer une instance d'une sous classe de la classe principale.

Un exemple :

```
local
        e : TOTO
do
        !TITI!e.make(…)
end
```

Un objet e de type TITI (une sous-classe de TOTO) est créé, initialisé, attaché à e et ensuite la méthode de création make est appelée pour e.

2.2.4 Le Polymorphisme

2.2.4.1 Définition

2.2.4.1.1 En Java

Il existe un équivalent au mot clé *is* de C# : *instanceof*. Il permet donc de savoir à quel type de classe nous avons affaire.

2.2.4.1.2 En C++ et en Eiffel

Il n'existe pas d'équivalent au mot clé *is* de C#.

2.2.4.2 Masquer ou écraser une méthode de la classe de base

2.2.4.2.1 En Java

La redéfinition de méthodes ne se passe pas exactement comme en C#, le compilateur n'averti pas si une classe mère et une classe fille possèdent deux méthodes du même nom avec les mêmes paramètres, la méthode est juste redéfinie.

Il est aussi possible de redéfinir une méthode dans la classe fille (par exemple toto()), de même nom qu'une méthode de la classe mère, tout en gardant accès à la méthode de la classe mère grâce à **super (**ici super.toto()).

2.2.4.2.2 En C++

Comme en Java, le compilateur n'averti pas si une classe mère et une classe fille possèdent deux méthodes du même nom avec les mêmes paramètres, la méthode est juste redéfinie.

Il est en revanche possible de redéfinir une méthode dans la classe fille (par exemple toto()), de même nom qu'une méthode de la classe mère, tout en gardant accès à la méthode de la classe mère par "fille.mère::toto();" .

2.2.4.2.3 En Eiffel

En Eiffel il est impossible de surcharger une méthode, il est toutefois possible de renommer (**rename**) une méthode héritée ou bien de la redéfinir (**redefine**), voire même les deux grâce à l'héritage multiple (voir 3.3). Il est même possible pour une classe fille d'éliminer des membres de sa classe mère grâce à **undefine**.

2.2.4.3 Construire des classes abstraites

2.2.4.3.1 En Java

Les mécanismes des classes abstraites sont assez semblables à ceux de C#. Une méthode est abstraite (modificateur **abstract**) lorsqu'on la déclare sans donner son implémentation. Une classe doit être déclarée abstraite (**abstract class**) si elle possède une méthode abstraite. Il est impossible de créer une instance d'une classe abstraite.

2.2.4.3.2 En C++

Les mécanismes des classes abstraites sont assez semblables à ceux de C#. Une méthode est abstraite ou viruelle pure lorsqu'elle n'a pas d'implémentation et que sa signature est suivie de "=0". Il est impossible de créer une instance d'une classe abstraite. De telles classes ne sont pas pourvues de constructeurs.

2.2.4.3.3 En Eiffel

Le mot clé **deffered** peut être utilisé pour des classes et des méthodes. Une méthode **deffered** ne doit pas avoir d'implémentation. Si c'est une méthode qui est **deffered** alors la classe à laquelle elle appartient doit être déclarée **deffered**. Si c'est la classe qui est **deffered** alors elle doit avoir au moins une méthode **deffered**. Il est à noter qu'une classe **deffered** n'a pas de partie **creation**.

2.2.4.4 Empêcher qu'une classe puisse être transformée en sous classe

2.2.4.4.1 En Java

Une classe **final** ne peut avoir de classes filles. Il est à noter que si une méthode est **final** elle ne peut être non plus redéfinie.

2.2.4.4.2 En C++

Il n'existe pas d'équivalent aux mots clé **sealed**, **final** ou **frozen**.

2.2.4.4.3 En Eiffel

Le mot clé correspondant à **final** est **frozen,** il est valide pour les attributs et les méthodes des classes.

2.2.5 Interface

2.2.5.1 En Java

Une interface est une classe purement abstraite dont toutes les méthodes sont **abstract** et **public.** Le mot clé permettant de faire « l'héritage » d'interfaces est **implements.**

Un exemple :

public class Voiture **extends** Vehicule **implements** Transport

La classe Voiture hérite de la classe Vehicule et implémente l'interface Transport, c'est-à-dire que la Voiture PEUT_ETRE_UTILISEE_COMME un Transport.

2.2.5.2 En C++ et en Eiffel

La notion d'interface n'est pas présente.

3 L'héritage Multiple

3.1 Introduction

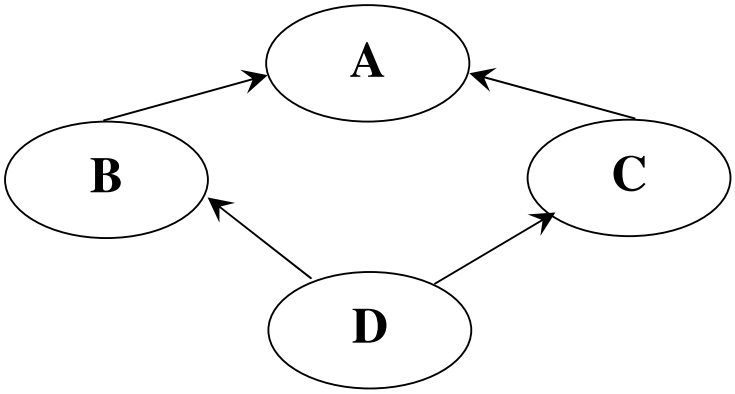
Plusieurs langages à objets, par exemple C++ et Eiffel, offrent le concept d'héritage multiple. Dans ces langages, une sous-classe peut hériter de deux ou de plusieurs super-classes immédiates.

Ceci pose un certain nombre de problèmes non triviaux, aussi bien au niveau de la définition du langage qu'à celui de son implémentation. On suppose, en effet, qu'il n'est pas possible de définir une relation d'héritage circulaire dans laquelle une classe pourrait hériter (directement ou indirectement) d'elle-même. Il est par contre possible qu'une classe hérite d'une même super-classe par deux, ou plusieurs, chemins distincts.

Considérons, dans un langage hypothétique, les déclarations suivantes:

```
class A { /* Une classe racine... */ };
class B inherit A { /*Héritage simple de A... */ };
class C inherit A { /*Héritage simple de A... */ };

class D inherit B,C
{
  Héritage multiple de B et C...
};
```

La sous-classe D possède deux super-classes immédiates B et C; on remarque de plus qu'elle hérite indirectement de la classe racine A par les deux relations d'héritage distinctes D->B->A et D->C->A. On a un graphe d'héritage en forme de losange:



Dans une instance de la classe D, les données et méthodes définies dans l'ancêtre A doivent-elles être héritées 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.

L'ordre de spécification des super-classes immédiates d'une classe donnée est significatif. Lors de la recherche d'un identificateur, ces super-classes sont visitées dans l'ordre où elles ont été spécifiées.

```
class A { /* Une classe racine... */ };
class B { /* Une classe racine... */ };

class C inherit A,B
{
  Héritage multiple de A et B...
};


class D inherit B,A
{
  Héritage multiple de B et A...
};

class E inherit C,D
{
  Héritage multiple de C et D...
};
```

Admettant, pour les besoins de la discussion, qu'il y a héritage commun de A et B dans les instances de E, la classe A devrait être visitée avant la classe B pour tenir compte de la clause d'héritage de C, mais après cette dernière pour tenir compte de celle de D.

Un autre problème, en cas d'héritage commun, est celui d'une méthode déclarée dans un ancêtre commun et redéfinie dans un chemin d'héritage, mais pas dans un autre. Faut-il utiliser la variante originale ou redéfinie de la méthode?

Un exemple:

```
class A
{ /* Une classe racine... */
  method m() { /*Méthode originale... */ };
};
```

```
class B inherit A { /*Héritage simple de A... */ };

class C inherit A
{ /*Héritage simple de A... */
  redefine m()
  { /*Méthode redéfinie pour les instances de C
      et de ses descendants...
    */
  };
};

class D inherit B,C
{
  Héritage multiple de B et C...
};
```

On a de nouveau le cas du graphe d'héritage en losange vu plus haut. On y ajoute une méthode m déclarée dans la classe racine A et redéfinie dans sa sous-classe C. Sous l'hypothèse d'un héritage commun de A, quelle version de cette méthode est applicable aux instances de D?

D'autre part, il se pose la question, en cas d'héritage répété, de spécifier la manière d'accéder, sans ambiguïté, aux entités héritées de la super-classe instanciée plusieurs fois dans l'objet composé. Ainsi, si l'on suppose un héritage répété de A dans les instances de D, y a-t-il moyen de distinguer, dans cette dernière classe, les entités héritées de l'instanciation de l'ancêtre A via B de celles héritées de son instanciation via C.

Voyons comment C++ et Eiffel répondent à ces questions

3.2 En C++

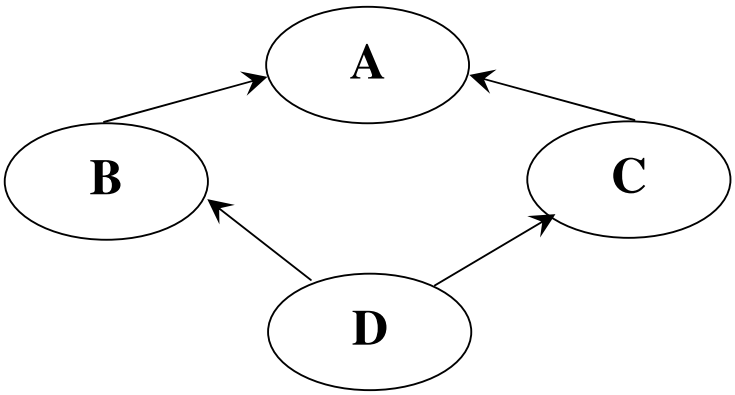
3.2.1 Héritage commun et héritage répété

En C++, la distinction entre l'héritage commun et l'héritage répété d'une super-classe héritée par des chemins distincts doit être spécifiée non pas dans la sous-classe finale où intervient l'héritage multiple concerné, mais dans les classes qui héritent directement de la super-classe concernée.

En cas d'héritage commun, les clauses d'héritage concernées doivent être préfixées du symbole virtual. Ainsi, reprenant l'exemple du graphe d'héritage en losange, on a les deux possibilités suivantes.

Héritage commun

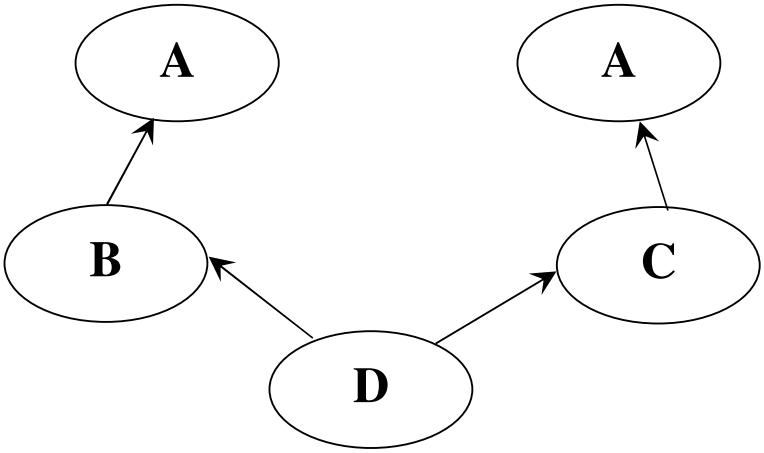
```
class A { /* Une classe racine... */ };
class B : virtual public A { /*Héritage simple de A... */ };
class C : virtual public A { /*Héritage simple de A... */ };

class D : public B, public C
{
  Héritage multiple de B et C;
  Héritage commun de A...
};
```



Héritage répété

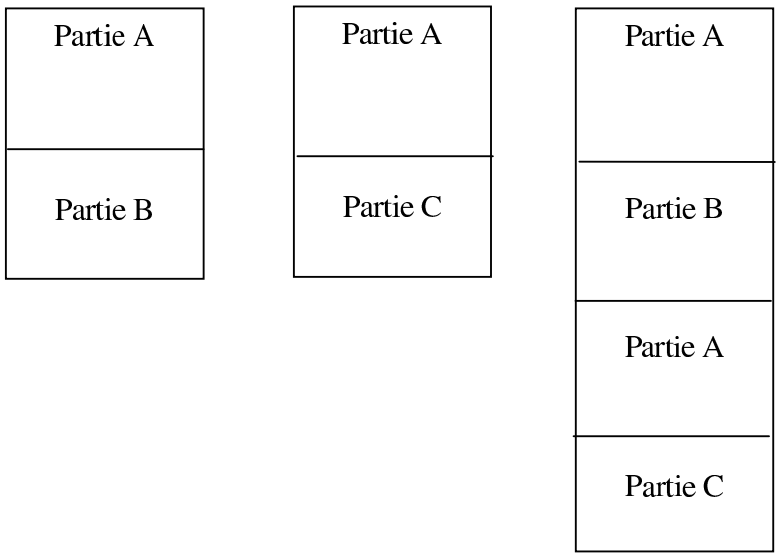
```
class A { /* Une classe racine... */ };
class B : public A { /*Héritage simple de A... */ };
class C : public A { /*Héritage simple de A... */ };

class D : public B, public C
{
```

```
  Héritage multiple de B et C;
  Héritage répété de A...
};
```



À priori, il apparaîtrait plus naturel de spécifier dans la déclaration de la sous-classe finale D, plutôt que dans les classes intermédiaires B et C, si A doit être héritée de manière commune ou répétée. Sémantiquement, cette spécification n'a, en effet, d'incidence que sur les instances de D. Le choix retenu est donc lié à la représentation interne des objets composés correspondants.

Dans le cas de l'exemple précédent, des instances des classes B, C et D pourront avoir les structures respectives suivantes:



3.2.2 Comment accéder à quoi ?

En C++, les accès ambigus intervenant lorsque le même identificateur est hérité de plusieurs super-classes distinctes doivent être qualifiés explicitement du nom d'une classe qui lève l'ambiguïté.

Un exemple:

```
class A
{ /*Une classe racine... */
  public:
    virtual void m() { /*Une méthode redéfinissable... */ };
};

class B
{ /*Une classe racine... */
  public:
    virtual void m() { /*Une méthode redéfinissable... */ };
};

class C: public A, public B
{ /*Une sous-classe de A et de B... */
};
```

L'identificateur m est hérité de chacun de ses deux ancêtres A et B au sein de C. Ceci est admis. Par contre, il n'est pas autorisé d'utiliser cet identificateur sans le qualifier explicitement à l'intérieur de C; il est nécessaire d'y utiliser les formes A::m ou B::m selon l'effet désiré.

On note qu'un identificateur hérité d'une super-classe commune n'a pas besoin d'être qualifié explicitement; par contre, s'il est hérité d'une super-classe répétée, il est nécessaire de la qualifier au moyen d'une classe intermédiaire susceptible de lever l'ambiguïté.

Héritage commun

```
class A
{ /* Une classe racine... */
  public:
    virtual void m() { /*Une méthode redéfinissable... */ };
};

class B : virtual public A { /*Héritage simple de A... */ };
class C : virtual public A { /*Héritage simple de A... */ };

class D : public B, public C
{
```

```
  /* Héritage multiple de B et C;
     héritage commun de A...
  */
};
```

À l'intérieur de la classe D, il y a héritage commun de la classe A; l'emploi non qualifié de l'identificateur m y est autorisé.

Héritage répété

```
class A
{ /* Une classe racine... */
  public:
    virtual void m() { /*Une méthode redéfinissable... */ };
};


class B : public A { /*Héritage simple de A... */ };
class C : public A { /*Héritage simple de A... */ };


class D : public B, public C
{
  /* Héritage multiple de B et C;
     héritage répété de A...
  */
};
```

À l'intérieur de la classe D, il y a héritage répété de la classe A; l'identificateur m qui en est hérité doit être explicitement qualifié B::m ou C::m .

En cas d'héritage commun, il reste la situation décrite par les déclarations:

```
class A
{ /* Une classe racine... */
  public:
    virtual void m() { /*Une méthode redéfinissable... */ };
};

class B : virtual public A { /*Héritage simple de A... */ };
class C : virtual public A
{ /*Héritage simple de A... */
  public:
    virtual void m() { /*Méthode redéfinie... */ };
};

class D : public B, public C
{
```

```
  /*Héritage multiple de B et C;
    héritage commun de A...
  */
};
```

La méthode m, est héritée de la super-classe commune A une fois sous sa forme originale via B et une fois sous sa forme redéfinie dans C. L'usage de l'identificateur m non qualifié au sein de la classe D est autorisé: il implique l'accès à la méthode redéfinie; l'accès à sa version originale devrait y être fait au moyen de la forme qualifiée A::m. Par contre, si cette méthode avait été redéfinie dans chacune des super-classes immédiates, il devrait être qualifié explicitement dans tous les cas A::m, B::m et C::m selon l'effet désiré.

3.3 En Eiffel

3.3.1 Clauses de renommage, redéfinition

La manière de résoudre, en Eiffel, les problèmes liés à l'héritage multiple diffèrent, en plusieurs points, des solutions retenues pour C++.

En Eiffel, il n'est pas autorisé, dans le cas général, d'hériter sans autres le même identificateur de plusieurs super-classes immédiates distinctes. Le cas échéant, d'éventuels homonymes doivent être rendus distincts au moyen de clauses de renommage.

Exemple:

```
class A
feature
  -- ...
  proc(k: INTEGER)is
  do
    -- ...
  end; --proc
  -- ...
end; --class A

class B
feature
  -- ...
  proc(x,y: REAL)is
  do
    -- ...
  end; --proc
  -- ...
end; --class B

class C
inherit
  A rename proc as a_proc end;
  B redefine proc end
feature
  -- ...
  proc(x,y: REAL)is
  do
    -- ...
  end; --proc
  -- ...
end; --class C
```

Il est défini deux classes racines A et B. Chacune d'entre elles exporte une méthode redéfinissable proc. Il est défini de plus la sous-classe C qui admet

les deux super-classes immédiates A et B. Ceci nécessite que proc soit renommée dans l'une, au moins, des deux clauses d'héritage.

Dans le cas présent, la méthode proc héritée de A est renommée a_proc au sein de C et en est exportée sous ce dernier nom. La méthode proc héritée de B n'est par contre pas renommée; elle est, par contre, redéfinie dans C et c'est cette version redéfinie qui est applicable aux instances de C.

3.3.2 Les exceptions

Il existe cependant des exceptions à l'interdiction d'hériter sous le même nom des entités exportées de plusieurs super-classes immédiates. En particulier, il est possible d'hériter d'une super-classe une méthode différée (abstraite) et d'hériter d'une autre super-classe une méthode concrète de même nom et de même profil. Dans ce cas, il est admis que la version concrète de la méthode réalise l'implémentation effective de sa version différée pour les instances de la sous-classe.

Un cas très particulier est celui des identificateurs hérités d'une même classe ancêtre par deux (ou plusieurs) chemins d'héritage distincts (par exemple dans le cas d'un graphe d'héritage en losange).

Si une entité exportée d'une classe est héritée dans une sous-classe sous le même nom par deux plusieurs chemins d'héritage distincts, elle sera héritée dans cette dernière sous forme commune et n'y existera qu'à un exemplaire.

Par contre, si une même entité est héritée dans une sous-classe sous des noms distincts (à la suite de clauses de renommage), elle sera héritée sous forme répétée à raison d'un exemplaire pour chaque identificateur distinct. Ainsi, la distinction entre héritage commun et répété ne se fait pas, à priori, en bloc pour l'ensemble d'une classe héritée par plusieurs chemins d'héritage distincts, mais isolément pour chacune de ses entités, ce qui semble déconcertant, voire douteux dans le cas des variables.

Exemple:

```
class A
feature
  -- ...
  x,y,z: REAL;
  -- ...
  proc is
  do
    x:=x+1; y:=y+1; z:=z+1;
  end; --proc
```

```
  -- ...
end; --class A

class B
inherit
  A rename x as bx end
feature
  -- ...
end; --class B

class C
inherit
  A rename z as cz end
feature
  -- ...
end; --class C

class D
inherit
  B; C
feature
-- ...
end; --class D
```

On reconnaît le graphe d'héritage en losange. La sous-classe B redéfinit, sous le nom bx la variable x exportée de A. Cette variable sera héritée à deux exemplaires dans la sous-classe D, une fois sous le nom bx via B et une fois sous son nom original via C. De même, la variable z est héritée à deux exemplaires, une fois sous son nom original via B et une fois sous le nom cz via C. Il y a par contre héritage commun de la variable y et de la méthode proc.

Les instances de D contiennent donc les cinq variables réelles x, bx, y, z et cz. La question est de savoir quels exemplaires de x et de z seront incrémentés lors de l'application de la méthode proc à une instance de D; la chose est loin d'être claire. Sous cette forme, la hiérarchie de classes est illégale; il faut lever l'ambiguïté précitée de qui se fait au moyen de clauses **select** incorporées à une clause d'héritage. L'exemple précédent peut être modifié comme suit.

```
class A
feature
  -- ...
  x,y,z: REAL;
  -- ...
  proc is
  do
    x:=x+1; y:=y+1; z:=z+1;
  end; --proc
  -- ...
end; --class A

class B
inherit
  A rename x as bx select x end
feature
```

```
  -- ...
end; --class B

class C
inherit
  A rename z as cz select cz end
feature
  -- ...
end; --class C

class D
inherit
  B; C
feature
-- ...
end; --class D
```

Les clauses **select** x dans la sous-classe B implique qu'en cas de polymorphisme portant sur une version répétée de l'entité x, c'est la version originale x qui sera retenue. Par contre, en cas de polymorphisme portant sur une version répétée de l'entité y, c'est sa version cz renommée dans C qui sera retenue. L'application de la méthode proc à une instance de D aura pour effet d'incrémenter ses variables x, y et cz.

Si cette manière de faire apparaît artificielle, elle doit être utilisée dans le cas fréquent où une classe redéfinit une méthode tout en gardant un accès à la méthode héritée.

Exemple:

```
class A
feature
  -- ...
  proc(k: INTEGER)is
  do
    -- ...
  end; --proc
  -- ...
end; --class A

class B
inherit
  A rename proc as super_proc end;
  A redefine proc select proc end
feature
  -- ...
  proc(x,y: REAL)is
  do
    -- ...
    super_proc(x,y);
    -- ...
  end; --proc
  -- ...
end; --class B
```

La sous-classe B hérite deux fois directement la classe A, ce qu'Eiffel autorise. La plupart des entités sont héritées de manière commune; par contre, proc est répétée une fois sous le nom super_proc, une autre fois sous son nom original. Dans ce dernier cas, elle est redéfinie; du fait de la clause select, c'est cette version redéfinie qui sera appliquée aux instances de B en cas d'invocation polymorphe de cette méthode.

3.4 En Java et en C#

Ces deux langages permettent l'héritage multiple d'interface combiné à l'héritage simple de classe. Toutefois, une simulation d'héritage multiple reste possible : il faut choisir une des classes mère correspondant le plus au critère EST_UN, et utiliser les autres classes mère candidates en tant que membre de la nouvelle classe (relation A_UN).

4 Conclusion

Les mécanismes d'héritages permettent de modéliser de manière simple et intuitive le monde qui nous entoure, il rend l'informatique plus proche de l'Homme. En effet, de nos jours deux types d'héritages coexistent.

D'une part l'héritage classique peut être implémenté par tous les langages objets, en effet dans « l'univers » de la programmation objet il est fondamental de pouvoir réutiliser des objets déjà existants et permettre ainsi de créer des objets plus spécifiques ayant les mêmes caractéristiques que l'objet principal tout en développant ses propres caractéritisques ou en développant d'une manière différente les comportements de l'objet principal.

C# et Java qui sont tous les deux des langages utilisant l'héritage classique sont très similaires, en effet la différence se fait surtout par les mots clés. C# propose en plus une sécurité supplémentaire dans le mécanisme de redéfinition, en effet il faut préciser si une méthode est redéfinie intentionnellement par le mot clé **new.**

C++ permet quand à lui de spécifier d'une manière plus particulière la relation d'héritage, il est possible d'hériter d'une classe de manière **public**, **protected** ou **private, c**hose qui est absente dans les autres langages. Il manque à C++ la possibilité de « finaliser » (au sens Java du terme **final)** un membre d'une classe ou même une classe, ce qui pose des problèmes de sécurité dès lors qu'une classe peut être réutilisée par d'autres programmeurs (comme cela se passe en Java).

Au niveau des constructeurs, Eiffel oblige l'utilisateur à spécifier explicitement le constructeur invoqué dès lors qu'il y en a un d'implémenté. Il est aussi possible de créer une instance d'une sous classe de la classe principale.

D'autre part, l'héritage multiple peut etre implémenté par ces quatre langages mais de manière différente pour Java et C#. En effet, ces deux langages simulent l'héritage multiple par le biais des interfaces.

L'héritage multiple permet de mieux modéliser le concept objet tel qu'il existe dans la nature, mais il est faut être sûr de sa programmation car il est source de nombreuses erreurs pour le programmeur.

C++ et Eiffel qui sont tous les deux des langages utilisant l'héritage multiples sont néanmoins très différents et n'ont rien à voir au niveau de la syntaxe. C++ premet de choisr entre héritage commun (graphe en losange) et héritage répété. Eiffel impose l'héritage commun, il est toutefois possible de procéder à un héritage répété grâce aux clauses de renommage.

Toutefois la programmation par objet s'essoufle et il devient difficile de trouver de nouvelles solutions. Cependant depuis peu il existe un nouveau mode de pensée qui est la programmation par composant.

Ainsi, l'héritage n'est il pas menacé par la programmation par composant ?

Bibliographie

[1] Nino Silverio – Langage C++ - Troisième édition deuxième tirage 1999.

[2] Stephen Randy Davis – C# pour les Nuls – 2002.

[3] http://www.irisa.fr/pampa/EPEE/oosewe.pdf

[4] Richard Grin – Cours de Java (pour Licence Informatique) – 2003.

[5] Raphaëlle Chaine – Cours de C++ (pour Maîtrise Informatique) – 2002 2003.