TOULOUSE

Informatique

iut Midi-Pyrénées

UNIVERSITÉ TOULOUSE III

Concepts de la programmation par objets

Christian Percebois

Département Informatique
IUT A Paul Sabatier

janvier 2014

TABLE DES MATIERES

Avant-propos

Ce document introduit les **concepts de la programmation par objets**, conformément au module M2103 (Bases de la programmation orientée objets), et couvre les points suivants :

- classes et objets,
- héritage de classes,
- composition d'objets,
- et polymorphisme.

Compte tenu des différentes déclinaisons de ces concepts, le cours ne s'appuie pas sur un langage à objets spécifique, mais tente plutôt de présenter chaque notion telle qu'elle est le plus souvent admise par la communauté.

Chapitre 1 : Les classes et les objets

Le développement de logiciels de plus en plus complexes nécessite l'exigence d'une certaine qualité de réalisation et cette réalisation est souvent suivie d'une phase dite de maintenance qui consiste à corriger le logiciel et à le faire évoluer.

La programmation par objets vise à apporter des solutions à au moins ces deux aspects en se focalisant sur des facteurs de qualité du développement du code : la réutilisabilité et l'extensibilité. La réutilisabilité correspond à la capacité d'exploiter tel quel un module logiciel déjà développé dans une nouvelle application. L'extensibilité traduit la capacité d'un logiciel à intégrer facilement de nouvelles spécifications, qu'elles soient demandées par les utilisateurs ou imposées par un événement extérieur.

On se propose, après une définition des termes classe et objet, d'étudier les trois principes qui caractérisent la programmation par objets [1, 3, 4, 5, 6 et 7] : l'instanciation (ce chapitre), l'héritage (chapitre 2) et l'envoi de message (chapitre 3). L'exposé ne s'appuie pas sur un langage spécifique[1] ; toutefois, chaque concept sera exprimé en pseudo-Java comme illustration, mais il ne saurait bien évidemment être question de syntaxe dans ces chapitres.

1. L'OBJET

Définition

Un objet est une structure de données, présente à l'exécution, constituée de valeurs appelées champs et sur laquelle on peut appliquer des opérations appelées méthodes.

Exemple

Un hypermarché désire cataloguer ses articles à la vente[2]. Il préconise pour cela de représenter chacune de ses ventes par un objet fournissant sa désignation, son prix hors taxes, son prix de vente et son coût de livraison au client lorsque ce service est disponible.

On peut considérer par exemple la chemise *laLacoste* dont les données sont (en mémoire) :

désignation	"Chemise"
prixHT	80.0

[1] Simula, Smalltalk-80, C++, C#, Ceyx, CLOS, Ruby, Eiffel, Python, Hybrid, Diesel, Ada 95, Java, New Flavors, Objective-C, Objective Caml, Object Pascal, ObjVlisp, ...

[2] L'exemple des ventes d'un hypermarché servira de fil conducteur à ce cours. Il s'inspire de l'ouvrage de G. Masini et al. [4], mais en diffère dans le sens où nous ne modélisons pas un type d'article pour le service de comptabilité, mais un article à la vente.

munie notamment des opérations *prixDeVente* (pour calculer le prix de vente d'une chemise) et *coûtLivraison* (pour calculer son coût de livraison au client).

Un autre article, l'eau gazeuse *laBadoit*, muni aussi des opérations *prixDeVente* et *coûtLivraison* pourra se représenter par :

désignation	"EauGazeuse"
prixHT	1.0

Dans la terminologie des objets, les valeurs sont appelées champs (comme pour un enregistrement). Un objet possède une identité qui lui est propre et l'ensemble des valeurs de ses champs définit son état. Un objet se caractérise aussi par un comportement incarné par ses méthodes. Un objet est donc un triplet (identité, état, comportement).

Deux objets ayant des identités différentes peuvent avoir des champs identiques. Réciproquement, les champs d'un objet donné peuvent évoluer durant l'exécution d'un système, mais cela n'affecte en rien l'identité de l'objet.

2. LA CLASSE

Définition

Une classe est la description d'une famille d'objets ayant même structure et même comportement. Chaque classe possède donc une double facette :

- des attributs auxquels sont associés les champs des objets,
- des opérations, appelées méthodes qui représentent le comportement commun des objets appartenant à la classe.

L'ensemble des attributs et des méthodes d'une classe définit ses propriétés.

Exemple

Les ventes du magasin ont fait apparaître deux objets *laLacoste* et *laBadoit* ayant même structure et même comportement. On définit donc une première classe d'objets, *Article*, décrivant un article disponible en rayon.

```
class Article
{
      private String désignation ;
      private Float prixHT ;

      // calcule le prix de vente d'un article
      public Float prixDeVente ()
      {
            return (prixHT * 1.2) ;              // TVA = 20 %
      }
```

```
        // calcule le coût de livraison d'un article
        public Float coûtLivraison ()
        {
            return (prixHT * 0.05) ;            // taxe livraison = 5 %
        }

        // fournit la désignation d'un article
        public String getDésignation ()
        {
            return (désignation) ;
        }

        // fournit le prix hors taxes d'un article
        public Float getPrixHT ()
        {
            return (prixHT) ;
        }

        // modifie le prix hors taxes d'un article
        public void setPrixHT (Float p)
        {
            prixHT = p ;
        }
    }
```

Remarques

1) Noter que les méthodes de la classe *Article* ne possèdent pas en paramètre un article car le code de la classe correspond au code d'un objet, appelé code de l'instance courante. Le code de la classe *Article* s'applique de facto aux deux articles que nous avons introduits : la chemise *laLacoste* et l'eau gazeuse *laBadoit*.

2) La classe cache au monde extérieur ses détails d'implantation. On parle ainsi de masquage d'information (ou d'encapsulation). Les attributs et méthodes préfixés par le mot-clé ***private*** ne sont accessibles qu'au sein de la classe ; à l'inverse, le mot-clé ***public*** spécifie que l'entité associée est exportée. En pratique, les attributs d'une classe sont privés à la classe (mot-clé ***private***).

3. LES OBJETS : INSTANCES DES CLASSES

Une classe établit un modèle de construction pour ses représentants appelés instances ou objets. Le code d'une classe correspond dès lors à un moule permettant de reproduire autant d'exemplaires que nécessaire ; on parle dans ce cas d'instanciation par moulage.

3.1. Le constructeur

La structure d'un objet est fournie par les attributs de la classe. On dote dès lors la classe d'une opération spécifique, appelée constructeur, ayant en charge la création

de l'objet. Nous prendrons pour convention que le constructeur d'une classe est une méthode spécifique ayant pour nom celui de la classe. En outre, un constructeur ne renvoie aucun résultat.

Exemple

Le constructeur de la classe *Article* accepte en paramètres les informations indispensables à la création d'un article bien constitué : une désignation *d* et un prix hors taxes *p*. On écrira dès lors :

```
class Article
{
        …

        // constructeur : initialise les champs d'un article
        public Article (String d, Float p)
        {
                désignation = d ;
                prixHT = p ;
        }

        // autres méthodes
        …
}
```

3.2. La relation d'instanciation (*new*)

La création d'un objet nécessite la définition d'une classe d'appartenance et d'un constructeur pour cette classe. Elle utilise la relation d'instanciation, nommée ***new*** par la suite. Pour créer les deux objets *laLacoste* et *laBadoit*, à partir de la classe *Article*, on doit initialement les déclarer, puis invoquer explicitement le constructeur *Article*. On écrira donc respectivement :

```
// déclaration de l'instance laLacoste
Article laLacoste ;

// création de l'instance laLacoste
laLacoste = new Article ("Chemise", 80.0) ;
```

```
// déclaration de l'instance laBadoit
Article laBadoit ;

// création de l'instance laBadoit
laBadoit = new Article ("EauGazeuse", 1.0) ;
```

Les instances *laLacoste* et *laBadoit* ont pour type *Article*. La structure de données décrite par les attributs de cette classe *Article* est dupliquée par l'instruction ***new*** pour former les champs de chaque instance ; cette structure est ensuite affectée

par les valeurs propres à l'objet que l'instance représente grâce au constructeur. Ne pas confondre l'attribut (relatif à la classe) et le champ (relatif à l'objet).

En pratique, l'instanciation d'un objet se fait généralement en deux temps. Premièrement, un espace mémoire de la taille requise par les attributs de la classe est alloué afin de matérialiser l'objet. Ensuite, l'état de l'objet est initialisé par le constructeur de la classe. Le constructeur confère à la classe un statut de fabrique d'objets.

On obtient ainsi pour l'instance *laLacoste* :

désignation	"Chemise"
prixHT	80.0

et pour l'instance *laBadoit* :

désignation	"EauGazeuse"
prixHT	1.0

Par la suite, on manipulera un objet via son nom, à savoir l'identificateur *laLacoste* pour la chemise et l'identificateur *laBadoit* pour l'eau gazeuse pour l'exemple qui nous concerne.

Remarques

1) Dans un système à objets, il n'existe pas d'autres objets que des instances de classe : tout objet *o* est une instance d'une classe *C et C* est le type de l'objet *o*. On obtient alors module = classe. En pratique, une classe s'apparente à un type.

2) La classe est une structure statique, alors que l'objet est dynamique : un objet est créé dynamiquement durant l'exécution du programme, grâce à la relation d'instanciation (mot-clé ***new***).

3) Outre les services qu'elle peut proposer, une classe possède en général cinq principales catégories d'opérations :

- les constructeurs qui créent des objets,
- les destructeurs qui détruisent des objets,
- les sélecteurs[3] (ou opérations de consultation) qui renvoient tout ou partie de l'état d'un objet,
- les modificateurs[4] (ou opérations de modification) qui changent tout ou partie de l'état d'un objet,
- les itérateurs qui permettent d'effectuer un traitement sur tous les composants de l'objet.

[3] souvent préfixés par *get*

[4] souvent préfixés par *set*

Chapitre 2 : L'héritage de classes et la composition d'objets

Il existe deux types de relation entre classes : une relation "être" et une relation "avoir" (ou "posséder"). La première correspond à la notion d'héritage ; la seconde à la notion de client ou d'utilisation. La relation "avoir" est à la base de la composition d'objets.

1. L'HERITAGE DE CLASSES : LA RELATION *ETRE*

1.1. Définition

Avec l'héritage, une classe partage les propriétés (attributs et méthodes) d'une autre classe, appelée sa superclasse. Les classes sont alors spécialisées et les propriétés des classes les plus générales sont héritées par les classes dérivées.

Exemple

Il est possible de gérer l'ensemble des ventes avec la seule classe *Article*. Cependant, les articles se distinguent car ils n'ont pas exactement les mêmes caractéristiques. Par exemple, l'attribut *taille* est nécessaire pour enregistrer la taille d'une chemise, mais il est superflu pour une bouteille d'eau gazeuse. Dans un même ordre d'idée, on peut penser que le *prixDeVente* d'un article est fonction de la TVA.

Il existe deux façons de spécialiser une classe : l'enrichissement et la substitution.

1) L'enrichissement

L'enrichissement dote une sous-classe de nouveaux attributs et/ou de nouvelles méthodes propres au sous-ensemble d'objets ainsi décrit. Par exemple, pour modéliser la taille et le coloris des chemises en vente, il suffit de définir une sous-classe *Chemise* de la classe *Article* (mot-clé ***extends***) en précisant les attributs *taille* et *coloris* :

```
class Chemise extends Article
{
        private Int taille ;
        private Couleur coloris ;

        // constructeur de la classe Chemise
        public Chemise (String d, Float p, Int t, Couleur c)
        {
                super (d, p) ;
                taille = t ;
                coloris = c ;
        }
}
```

Par héritage, une chemise possède les champs *désignation*, *prixHT*, *taille* et *coloris* et les méthodes *prixDeVente*, *coûtLivraison*, *getDésignation*, *getPrixHT* et *setPrixHT*.

2) La substitution

La substitution propose de redéfinir un traitement hérité. Les objets de la sous-classe ont alors un autre comportement que celui spécifié par la classe ancêtre. Par exemple, pour une bouteille de Badoit, le prix doit tenir compte d'une TVA égale à 5,5 % car il s'agit d'un produit alimentaire ; la TVA ne représente plus 20 % du prix comme dans le cas d'un article standard. Ce calcul du *prixDeVente* d'une bouteille de Badoit introduit dès lors la classe *EauGazeuse*, sous-classe de la classe *Article* (mot-clé ***extends***) :

```
class EauGazeuse extends Article
{
        // constructeur de la classe EauGazeuse
        public EauGazeuse (String d, Float p)
        {
                super (d, p) ;
        }

        // redéfinition du calcul du prix de vente de la classe Article
        public Float prixDeVente ()
        {
                return (getPrixHT () * 1.055) ;          // TVA = 5,5  %
        }
}
```

La sous-classe *EauGazeuse* détient les mêmes informations que la classe *Article*, sauf pour la méthode redéfinie *prixDeVente*. On dit que la nouvelle méthode masque celle qui a été héritée. Par convention, nous considérons dans ce cours qu'une redéfinition est effective lorsque la sous-classe fournit un corps de méthode d'en-tête strictement identique à l'en-tête de la méthode héritée[5].

A partir de ces deux nouvelles classes *Chemise* et *EauGazeuse*, on peut créer les deux objets *laLacoste* (instance de la classe *Chemise*) et *laBadoit* (instance de la classe *EauGazeuse*) en écrivant respectivement :

```
Chemise laLacoste ;
laLacoste = new Chemise ("Chemise", 80.0, 1, rouge) ;
```

```
EauGazeuse laBadoit ;
laBadoit = new EauGazeuse ("EauGazeuse", 1.0) ;
```

[5] Il est parfois possible de modifier le type des paramètres ou le type de retour d'une méthode que l'on redéfinit. Le nouveau type de retour peut, par exemple, être un sous-type du type déclaré dans la méthode héritée (principe de covariance).

L'état initial de l'objet *laLacoste* sera désormais :

désignation	"Chemise"
prixHT	80.0
taille	1
coloris	rouge

Remarques

1) Bien entendu, on peut dans une même sous-classe, utiliser les mécanismes d'enrichissement et de substitution d'une classe ancêtre.

2) Les attributs *désignation* et *prixHT* ainsi que les méthodes *prixDeVente*, *coûtLivraison*, *getDésignation*, *getPrixHT* et *setPrixHT* ont exactement la même définition dans les deux classes *Article* et *Chemise*. Ils n'ont pas besoin d'être répétés dans la classe *Chemise*, d'après la clause ***extends*** *Article* qui spécifie qu'une chemise est un article.

3) De par le principe du masquage d'information, la classe *Chemise* ne peut directement accéder au champ *prixHT* d'un article. Pour ce faire, elle doit invoquer l'accesseur *getPrixHT*.

4) Le constructeur de la classe *Chemise* possède désormais quatre paramètres. Il fait référence au constructeur de la classe *Article* (mot-clé ***super***) pour initialiser les attributs *désignation* et *prixHT*. Créer une bouteille d'eau de Badoit revient simplement à invoquer le constructeur *Article* (mot-clé ***super***).

1.2. La sémantique de l'héritage

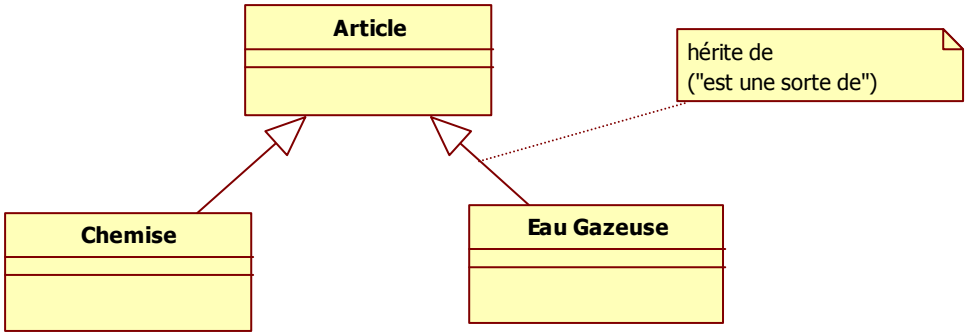
L'héritage n'a pas de sémantique formelle et dès lors correspond à un concept pouvant donner lieu à de multiples interprétations. Cependant, deux points de vue peuvent être dégagés de cette relation, celui des objets (extension) et celui de la classe (intention). L'inclusion des extensions s'appuie sur le fait qu'on considère une classe comme un ensemble d'objets. Avec l'inclusion des intentions, la classe est perçue comme un ensemble de propriétés.

Remarque

Ces deux points de vue duaux confèrent à une sous-classe le statut de sous-type. Si *B* hérite de *A*, *B* est considéré comme un sous-type de *A* et tout objet déclaré de type *A* pourra désigner à l'exécution un objet de type *B*.

1.3. Le graphe d'héritage

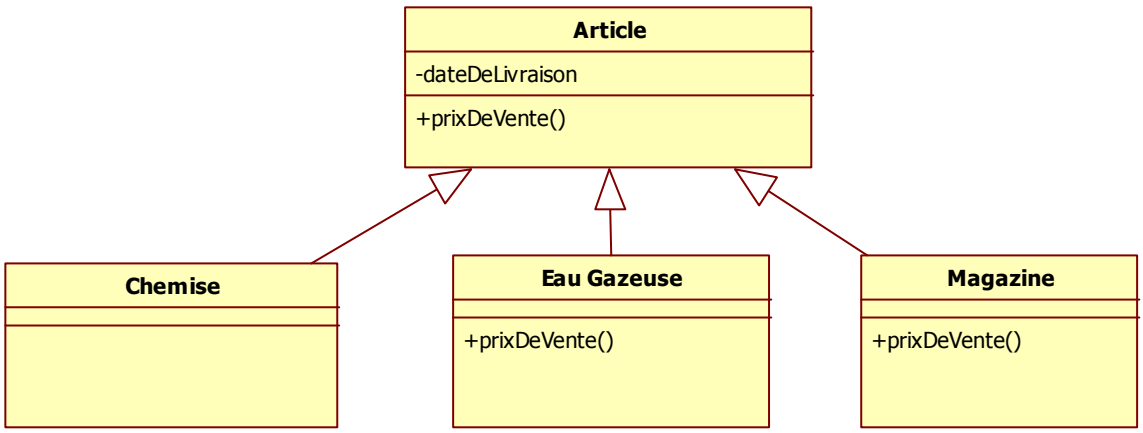
La relation d'héritage lie une classe à sa superclasse. Les classes sont organisées hiérarchiquement à la manière des taxonomies utilisées en biologie ou en zoologie. La représentation graphique de la relation forme un graphe, appelé graphe d'héritage. Pour notre problème, le graphe d'héritage avec la syntaxe UML [6] du diagramme de classes est donc :



Remarques

1) La structuration en classes et sous-classes entraîne une modularité importante. La modification de l'une d'elles n'entraîne qu'un minimum de modifications des autres.

Par exemple, compléter la classe *Article* par un nouvel attribut *dateDeLivraison* fait automatiquement bénéficier ses sous-classes *Chemise* et *EauGazeuse* de cette nouvelle information. On peut également introduire un nouveau type d'article, les magazines, dont le taux de TVA est fixé à 2,1 %, ce qui conduit à redéfinir la méthode *prixDeVente* héritée, à l'instar de ce que nous avons fait pour la classe *EauGazeuse*. Cet exemple illustre la capacité des objets à prendre en compte les facteurs de réutilisabilité et d'extensibilité du génie logiciel.



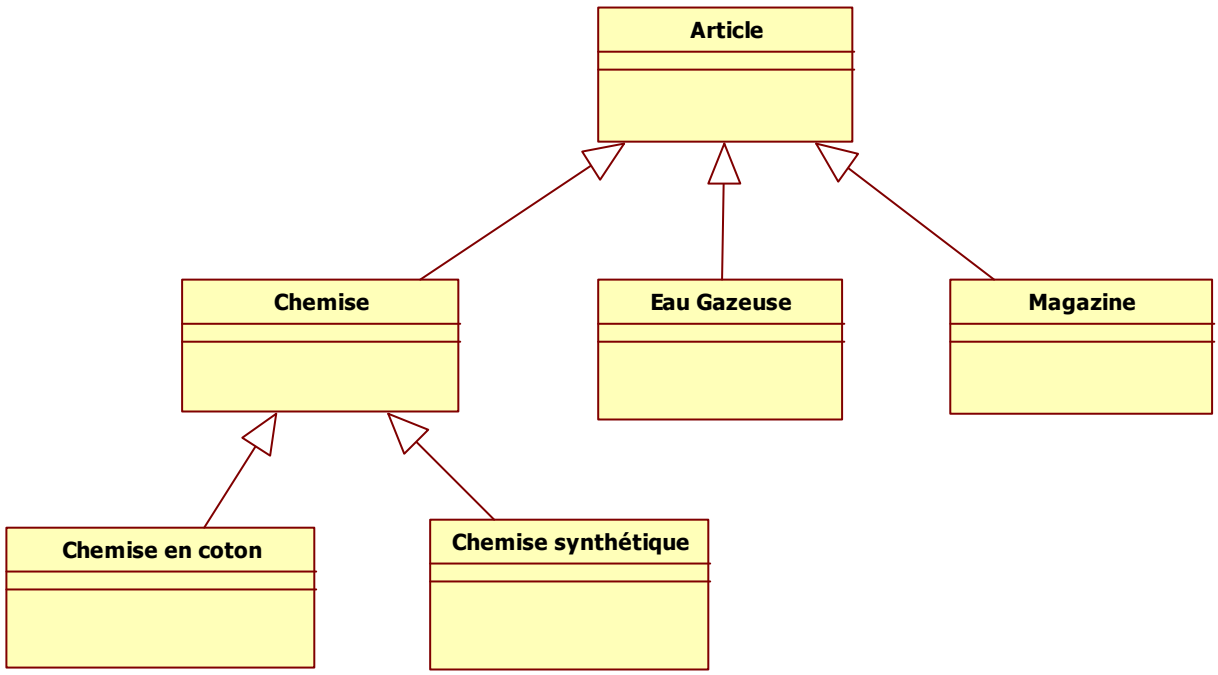
Concernant le code pseudo-Java associé à cette hiérarchie de classes, il vient :

```
class Article
{
        …
        private Date dateDeLivraison ;
        …
}
```

```
class Magazine extends Article
{
        …

        // redéfinition du calcul du prix de vente de la classe Article
        public Float prixDeVente ()
        {
                return (getPrixHT ( ) * 1.021) ;          // TVA = 2,1 %
        }
}
```

Bien entendu, il est tout à fait possible d'enrichir ce schéma par de nouvelles classes. Par exemple, par la définition des sous-classes *ChemiseEnCoton* et *ChemiseSynthétique*, héritières de la classe *Chemise*.



2) La relation d'héritage est transitive : les caractéristiques des classes supérieures sont héritées par les classes inférieures, qui sont d'autant plus spécialisées qu'elles sont proches des feuilles de l'arbre. Elle est par contre antiréflexive et antisymétrique.

3) Par extension de la définition donnée précédemment, le terme superclasse désigne toute classe dont hérite une classe donnée. Ainsi, la classe *ChemiseEnCoton* a pour superclasses *Chemise* et *Article*.

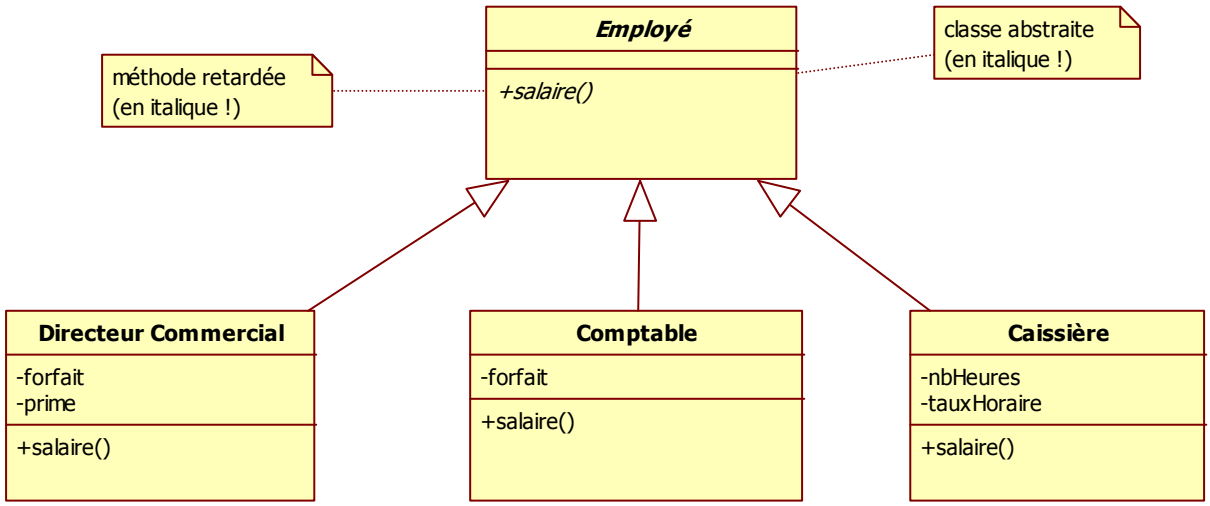
1.4. Classes abstraites et méthodes retardées

On doit parfois déléguer l'écriture d'une méthode à l'ensemble de ses classes descendantes.

Exemple

Considérons la classe *Employé* de notre hypermarché et ses sous-classes *DirecteurCommercial*, *Comptable* et *Caissière*. En pratique, le salaire est spécifique à chacun des employés selon qu'il est assujetti à un forfait, qu'il bénéficie de primes ou que sa rémunération dépend de son nombre d'heures de travail. Dès lors, il devient illusoire de chercher à écrire le corps de la méthode *salaire* dans la classe *Employé* car les différentes réalisations possibles sont définies dans les sous-classes correspondantes *DirecteurCommercial*, *Comptable* et *Caissière*.

Une telle méthode est dite retardée ; elle est simplement déclarée au niveau de la classe *Employé (*mot-clé ***abstract)***. Lorsqu'une classe possède au moins une méthode retardée, elle est dite abstraite (mot-clé ***abstract***). On définira donc pour le graphe d'héritage UML :



Et pour le pseudo-code Java correspondant :

```
abstract class Employé
{
        …
        abstract public Float salaire () ;          // pas de corps
        …
}
```

```
class DirecteurCommercial extends Employé
{
        private Float forfait ;
        private Float prime ;

        //  constructeur
        public DirecteurCommercial (Float f, Float p)
        {
                forfait = f ;
                prime = p ;
        }

        // détermine le salaire d'un directeur commercial
        public Float salaire ()
        {
                return (forfait + prime) ;
        }
}
```

```
class Comptable extends Employé
{
        private Float forfait ;

        // constructeur
        public Comptable (Float f)
        {
                forfait = f ;
        }

        // détermine le salaire d'un comptable
        public Float salaire ()
        {
                return (forfait) ;
        }
}
```

```
class Caissière extends Employé
{
        private Int nbHeures ;
        private Float tauxHoraire ;

        // constructeur
        public Caissière (Int n, Float t)
        {
                nbHeures = n ;
                tauxHoraire = t ;
        }
```

```
        // détermine le salaire d'une caissière
        public Float salaire ()
        {
            return (nbHeures * tauxHoraire) ;
        }
    }
```

Remarques

1) Appliquer la méthode *salaire* à un objet de la classe abstraite *Employé* n'a pas de sens puisque la rémunération n'est pas calculée dans cette classe. Par contre, le code peut déclarer statiquement un *Employé* et désigner à l'exécution un *Comptable,* puis calculer son salaire.

2) Une classe qui hérite d'une classe abstraite sans définir toutes les méthodes retardées est elle-même, de fait, abstraite.

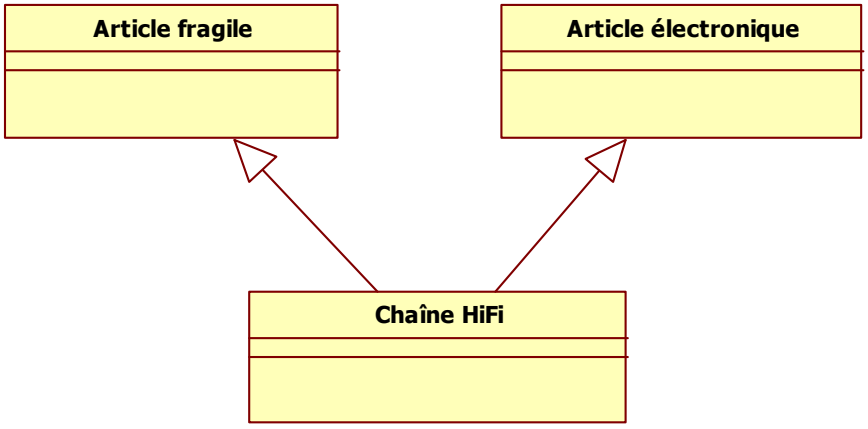
3) S'il est impossible de créer un objet en tant que représentant d'une classe abstraite, la classe peut cependant proposer un constructeur qui sera invoqué par les constructeurs des classes descendantes.

1.5. L'héritage simple et l'héritage multiple

Lorsqu'une classe ne possède qu'une seule superclasse directe, l'héritage est dit simple et la relation entre classes se traduit par une arborescence. Le partage de propriétés se limite au partage des différentes branches de l'arbre, parcourues depuis la sous-classe feuille jusqu'à la classe racine en suivant les liens d'héritage. En héritage multiple, une classe peut hériter directement de plusieurs classes et l'arbre devient un graphe : plusieurs chemins sont alors possibles, chacun offrant la possibilité de partager des données (attributs) et des traitements (méthodes).

Exemple

Ainsi, par exemple, la classe *ChaîneHiFi* peut hériter à la fois des classes *ArticleFragile* et *ArticleElectronique*. La première détaille les conditions d'emballage et de transport, alors que la seconde précise les éléments électroniques et acoustiques destinés à la restitution de sources sonores.



La clause ***extends*** de la classe *ChaîneHiFi* énumère alors les deux classes *ArticleFragile* et *ArticleElectronique* dont elle hérite :

```
class ChaîneHiFi extends ArticleFragile, ArticleElectronique
{
        …
}
```

L'héritage multiple facilite la mise en commun de données et de traitements, qu'il faudrait sinon dupliquer dans plusieurs classes. Il permet également un gain de place. En contrepartie, il impose de bien cerner le contour de chacune des classes et leur agencement doit être élaboré avec le plus grand soin.

Remarques

1) En héritage simple, le graphe est en fait un arbre !

2) Tous les langages à objets ne proposent pas le mécanisme d'héritage multiple[6], essentiellement car :

- la sémantique de l'héritage est peu claire,
- peu d'exemples pertinents exploitant l'héritage multiple existent,
- et des problèmes théoriques et pratiques, inévitables, subsistent.

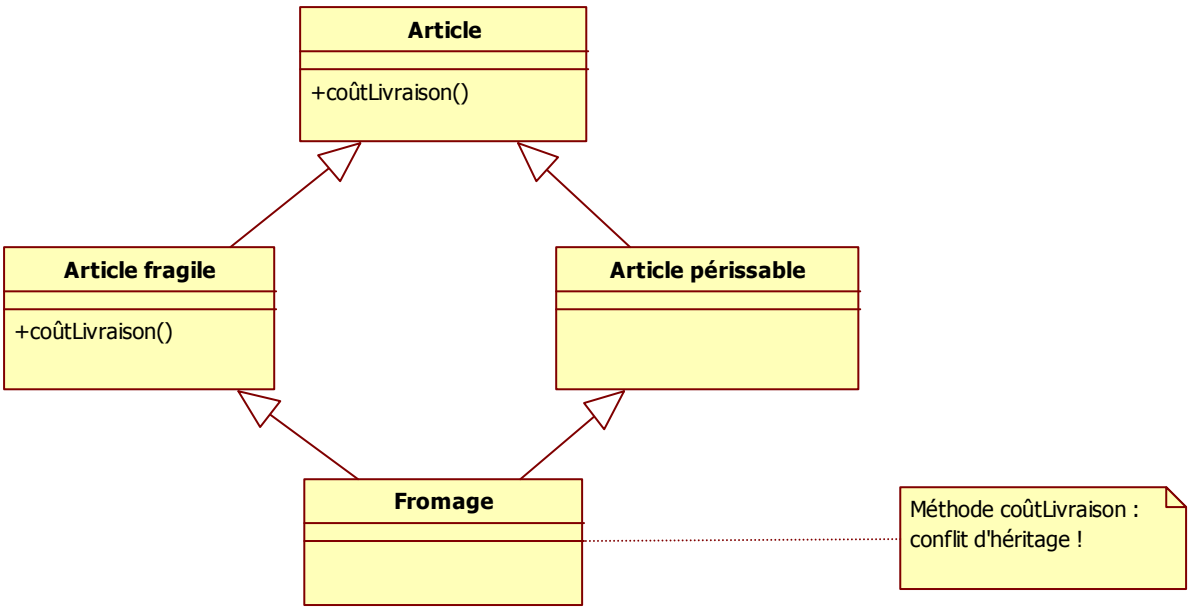
1.6. Conflits d'héritage

Avec l'héritage multiple apparaît le problème des conflits de nom, exprimé par la l'énoncé suivant : lorsqu'une même méthode est héritée de plusieurs classes, quelle méthode héritée choisir ?

Exemple

Supposons le graphe d'héritage suivant :

[6] Java par exemple qui propose une alternative à l'héritage multiple avec son concept d'interface que l'on peut voir comme la spécification d'un ensemble de méthodes publiques retardées indiquant les services rendus par les classes qui implémentent ces méthodes. Spécifier un service n'induit pas pour autant un comportement… cause du conflit en héritage multiple.



Deux méthodes *coûtLivraison* sont héritées par la classe *Fromage* : une par le biais de la classe *ArticleFragile* et l'autre par le biais de la classe *ArticlePérissable*. Ces deux méthodes sont en conflit car elles permettent toutes deux de répercuter au client les frais de livraison d'un fromage, mais de deux manières différentes. Notons que ce conflit est inévitable, dans le cas où la méthode *coûtLivraison* est héritée de la classe *Article*, puis redéfinie dans l'une des deux sous-classes *ArticleFragile* ou *ArticlePérissable*.

Il n'existe pas de technique simple pour résoudre de tels conflits et déterminer la méthode à appliquer à un objet. On utilise soit un parcours du graphe « câblé » dans le langage, soit des annotations du programmeur.

1) Parcours du graphe d'héritage

Un parcours du graphe d'héritage ordonne[7] les classes, ce qui permet ensuite de rechercher la première classe qui propose un corps pour ladite la méthode. Aucun parcours de la hiérarchie de l'objet n'est satisfaisant a priori. On distingue plusieurs heuristiques, par exemple :

- premier parcours
Fromage, *ArticleFragile*, *ArticlePérissable*, *Article*

- deuxième parcours
Fromage, *ArticlePérissable, Article, ArticleFragile*

Pour le premier parcours, c'est la méthode *coûtLivraison* de la classe *ArticleFragile* qui sera appelée ; pour le second, c'est celle de la classe *Article*.

[7] Le graphe d'héritage définit un ordre partiel des classes : certaines classes ne sont pas comparables. Le parcours du graphe (en largeur d'abord, en profondeur d'abord, …) permet d'établir un ordre total des classes qui deviennent alors toutes comparables.

L'important est que le parcours soit précisé dans la sémantique du langage et soit clairement défini pour l'utilisateur.

2) Annotations du programmeur

Une autre technique consiste à laisser le soin au programmeur de régler le conflit, considérant que c'est lui seul qui en connaît la cause. On peut soit invalider une méthode héritée, soit imposer la redéfinition de la dite méthode, soit renommer la méthode héritée dans la sous-classe, soit régler au coup par coup le conflit à chaque utilisation de la méthode, …

2. LA COMPOSITION D'OBJETS : LA RELATION *AVOIR*

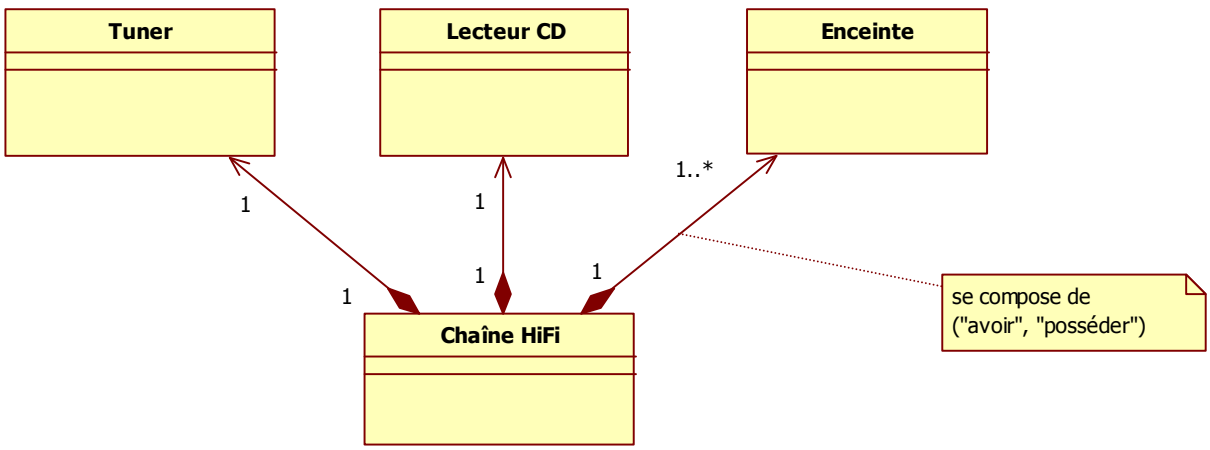
Il existe deux façons d'utiliser une classe. Une classe peut être descendante d'une autre classe : elle spécialise ou étend la classe parente (relation *être*). Une classe peut aussi utiliser les méthodes d'une autre classe dont elle cliente (relation *avoir* ou *posséder*). Ce dernier mode intervient notamment dans le cas de la composition d'objets.

2.1. Définition

Un objet composite est un objet formé de l'assemblage d'autres objets. La composition d'objets revient donc à construire de nouveaux objets plus complexes à partir d'objets existants dont l'interface est clairement spécifiée. Elle décrit une contenance structurelle entre instances.

2.2. Exemples

1) Une chaîne haute-fidélité comporte plusieurs éléments spécifiques comme un tuner, un lecteur CD et plusieurs enceintes. Ce n'est pas une spécialisation de ces différents éléments, mais plutôt un savant amalgame entre plusieurs composants spécifiques, qu'ils soient électroniques ou acoustiques. En programmation par objets, un objet chaîne hi-fi sera pourvu de champs désignant des instances des classes *Tuner*, *LecteurCD* et *Enceinte*. On dit alors qu'une chaîne est un objet composite, formé de l'assemblage d'autres objets. Un tel objet est composé entre autres d'un ensemble d'enceintes modélisé par l'objet *e* de type *Tableau <Enceinte>*.



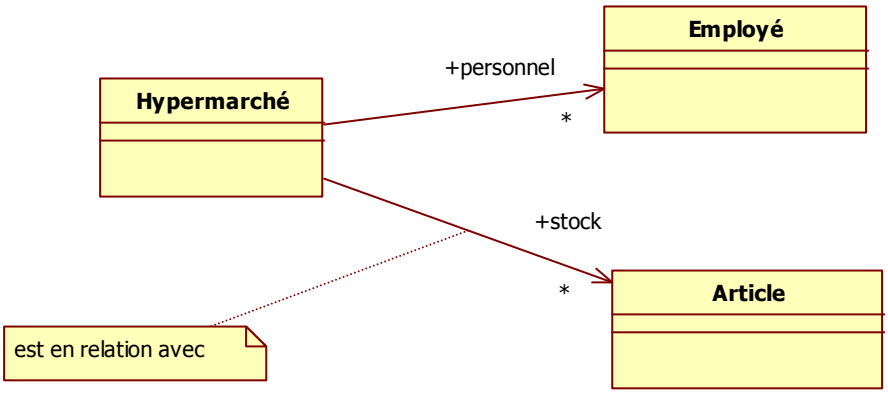
Pour cet exemple, on écrira :

```
class ChaîneHiFi
{
        private Tuner t ;
        private LecteurCD l ;
        private Tableau<Enceinte> e ;

        // constructeur : initialise les champs d'une chaîne haute-fidélité
        public ChaîneHiFi (Int nb)
        {
                t = new Tuner () ;
                l = new LecteurCD () ;
                e = new Tableau<Enceinte> (nb) ;
        }

        // autres méthodes d'une chaîne haute-fidélité
        …
}
```

2) On peut coder le *stock* et le *personnel* d'un *hypermarché* respectivement par un tableau d'articles et une liste d'employés. Cependant, le stock et le personnel ne sont pas assujettis à l'hypermarché : la relation n'est plus une composition (un losange noir), mais simplement une association (une simple flèche). Le couplage entre les objets *hypermarché*, *stock* et *personnel* s'avère dès lors plus lâche que pour une composition[8], car il n'y a pas nécessairement subordination des durées de vie du stock et du personnel vis-à-vis de l'hypermarché. On peut donc considérer ce type de relation comme un affaiblissement de la composition.



On codera souvent en pratique une association comme une composition car les langages à objets n'établissent pas nécessairement une distinction entre ces deux notions, contrairement à UML. Ce qui conduit au code suivant :

8 Dans la pratique, les chaînes haute-fidélité se présentent soit de manière compacte, un seul boîtier regroupant tous les éléments à l'exception des enceintes, soit sous la forme d'éléments séparés qui peuvent être de marques différentes. Le lien de composition ne s'impose plus si les éléments sont séparés ; il en est de même pour les enceintes quelle que soit la configuration.

```
class Hypermarché
{
        private Tableau<Article> stock ;
        private Liste<Employé> personnel ;

        // méthodes d'un hypermarché
        …
}
```

Remarques

1) En programmation par objets, le langage s'accompagne d'une bibliothèque de classes permettant, entre autres, d'implanter les types de données usuels (nombres, chaînes de caractères, tableaux, piles, files, …) et de réaliser les entrées-sorties. La classe *Tableau* fait partie de cette bibliothèque en tant que type abstrait de données. Cette classe générique offre bien évidemment tous les services indispensables à l'utilisation d'un tableau.

2) D'après le principe de masquage d'information, un objet composite n'appliquera à un de ses composants que les méthodes définies par son interface - les méthodes publiques. On dit alors que l'objet composite utilise les services de l'objet composant. Il en résulte alors la mise en place d'une relation client-fournisseur entre l'objet composite et ses constituants.

3) Une classe *A* est dite cliente de la classe *B* (et *B* est le fournisseur de la classe) lorsque *A* :

- contient la déclaration d'un attribut *b* de type *B*,
- possède une méthode ayant un paramètre d'entrée ou de retour *b* de type *B*
- ou utilise un objet local *b* de type *B* pour l'une de ses méthodes.

Le premier cas correspond à une dépendance structurelle des objets possédés par l'objet englobant (composition) ; les deux autres expriment une dépendance non-structurelle vis-à-vis des objets utilisés.

Chapitre 3 : L'envoi de message et le polymorphisme

1. L'ENVOI DE MESSAGE

1.1. Définition

L'invocation d'une méthode se traduit par une requête adressée à l'objet par un client. La satisfaction de la requête est à la charge de l'objet auquel elle est adressée. Ce mode de communication est appelé envoi de message.

L'envoi de message est l'unique façon de communiquer entre les objets. Il a pour but d'effectuer une opération attachée à un objet ; il doit toujours mentionner l'objet concerné, appelé le receveur du message, le sélecteur de la méthode à activer et ses paramètres.

1.2. Cas général (*send*)

Dans beaucoup de langages, l'envoi de message est réalisé par la primitive ***send***. On peut le noter :

send (objetReceveur, sélecteur (paramètre$_1$, ..., paramètre$_n$))

ou, plus synthétiquement :

objetReceveur.sélecteur (paramètre$_1$, ..., paramètre$_n$)

Par la suite, on adoptera la notation pointée pour traduire un envoi de message.

Exemples

1) Considérons l'objet *laLacoste*, instance de la classe *Chemise*, et l'écriture *prix = laLacoste.prixDeVente()*. Si le prix hors taxes de la chemise est de 80 €, la valeur affectée à *prix* sera égale à 96 € (TVA = 20 %).

2) Soit l'écriture *prix = a.prixDeVente()*. Si *a*, de type *Article*, mémorise une instance de la classe *Chemise* (par exemple une chemise *laLacoste*) avec un prix hors taxes de 80 €, le résultat retourné par le message est 96 €, car la méthode *prixDeVente* de la classe *Chemise* est héritée de la classe *Article* (TVA = 20 %). En revanche, si *a* est une instance de la classe *EauGazeuse* (par exemple *laBadoit*) avec un prix hors taxes de 1 €, c'est la méthode redéfinie de la classe *EauGazeuse* qui est activée. Le résultat est alors 1,05 € (TVA = 5,5 %).

3) Programmons maintenant le corps de la méthode *éditerStock* de la classe *Hypermarché* :

```
class Hypermarché
{
        private Tableau<Article> stock ;

        // constructeur
        public Hypermarché (Int n)
        {
                stock = new Tableau<Article> (n) ;
                …
        }

        // affiche le contenu du stock
        public void éditerStock ()
        {
                for (Int i = 0 ; i < stock.taille() ; i++)
                {
                        out.println(stock.getIème(i).getDésignation()) ;
                }
        }

        // autres méthodes
        …
}
```

La notation *stock.getIème(i).getDésignation()* s'interprète de gauche à droite : la méthode *getIème* des tableaux est d'abord appliquée à l'objet *stock* ; elle fournit un objet de type *Article* sur lequel la méthode *getDésignation* est invoquée.

Remarques

1) Comme c'est le cas dans la plupart des langages à objets, il a été considéré en outre que l'entrée standard (le clavier) et la sortie standard (l'écran) sont représentées respectivement par les objets prédéfinis *in* et *out*. Effectuer une opération d'entrée-sortie consiste à appliquer l'opération adéquate à l'un de ce deux objets, ici *println*, choisie dans l'interface de la classe *EntréeSortie* de la bibliothèque de base du langage[9]. Cet objet est souvent implicitement créé lors de sa première utilisation.

2) Si le sélecteur de la méthode appartient à la classe de l'objet receveur, le corps de la méthode correspondant est exécuté. Sinon, la recherche se poursuit dans la superclasse en considérant les méthodes qu'elle possède. On remonte ainsi dans le graphe d'héritage, de superclasse en superclasse, jusqu'à celle qui définit la méthode recherchée. Ce processus de recherche par remontée dans le graphe d'héritage prend donc directement en compte la redéfinition de méthode.

9 En pratique, ces flux d'entrée *in* et de sortie *out* ne sont pas spécifiques à une instance de la classe *EntréeSortie*, mais à la classe elle-même. On dit alors que ce sont des variables de classe, par opposition à des variables d'instance (objets). De façon similaire, il existe des méthodes de classe. En Java, on les introduit par le mot-clé ***static*** et on préfixe l'accès par le nom de la classe.

1.3. Désignation de l'objet receveur (*self*)

Dans les exemples de la section précédente, l'identité du receveur du message était connue : l'objet *laLacoste*, l'objet *a*, l'objet *out*… Le plus souvent, on ignore ce receveur au moment de l'écriture d'une méthode. Dans ce cours, nous le désignerons, comme en Java, par le mot-clé ***this***. L'envoi de message à cet objet cible (ou objet ***self***) s'écrit alors :

this.sélecteur(paramètre$_1$, ..., paramètre$_n$)

Exemple

Considérons par exemple une méthode calculant le *prixAvecLivraison* au client d'un article, par addition du *prixDeVente* et du *coûtLivraison*. Cette méthode est définie dans la classe *Article*, mais doit appliquer le calcul du taux de TVA adéquat quel que soit l'objet pour lequel elle est activée. En particulier, le prix de vente d'une chemise doit être calculé avec la méthode *prixDeVente* de la classe *Chemise* (20 % de TVA), tandis que celui d'une eau gazeuse doit l'être avec la méthode de la classe *EauGazeuse* (5,5 % de TVA).

Dans le corps de la méthode *prixAvecLivraison*, les messages permettant de calculer le *prixDeVente* et le *coûtLivraison* doivent donc être adressés à l'objet ayant reçu le message *prixAvecLivraison*, c'est-à-dire la variable ***this*** dénotant l'instance courante :

```
class Article
{
        …

        // calcule le prix de vente avec livraison d'un article
        public Float prixAvecLivraison ()
        {
                return (this.prixDeVente() + this.coûtLivraison()) ;
        }

        // autres méthodes
        …
}
```

Remarque

En fait, cette notion d'instance courante était implicite dans notre toute première version de la classe *Article* du chapitre 1, puisque le code d'une classe est celui de l'instance courante. Désormais, ***this*** préfixera systématiquement l'accès à un attribut

ou l'invocation d'une méthode lorsqu'il s'agira de l'objet courant[10], comme dans le fragment de code :

```
class Article
{
    private String désignation ;
    private Float prixHT ;

    // calcule le prix de vente d'un article
    public Float prixDeVente ()
    {
        return (this.prixHT * 1.2) ;            // TVA = 20 %
    }

    // calcule le coût de livraison d'un article
    public Float coûtLivraison ()
    {
        return (this.prixHT * 0.05) ;           // taxe = 5 %
    }

    // fournit la désignation d'un article
    public String getDésignation ()
    {
        return (this.désignation) ;
    }

    // fournit le prix hors taxes d'un article
    public Float getPrixHT ()
    {
        return (this.prixHT) ;
    }

    // modifie le prix hors taxes d'un article
    public void setPrixHT (Float p)
    {
        this.prixHT = p ;
    }
}
```

1.4. Accès à la superméthode (*super*)

L'héritage par substitution masque la méthode héritée. Ne pas accéder au sein d'une sous-classe au code d'une méthode masquée par héritage peut s'avérer contraignant, notamment lorsque la méthode redéfinie se traduit simplement par une spécialisation de la méthode masquée. Il arrive en effet assez fréquemment que la méthode redéfinie se contente d'ajouter, en amont ou en aval, un traitement spécifique au traitement hérité.

[10] De nombreux langages dispensent le programmeur de mentionner cet objet receveur ***this*** ; il s'agit alors d'une simplification syntaxique que nous ne préconisons pas pour l'écriture des codes.

Par analogie avec les notions de classe et de superclasse, une méthode masquée est appelée superméthode de la méthode qui la spécialise. Cette superméthode est invoquée en envoyant un message à une variable spéciale appelée ***super*** (qui n'est pas un objet !), comme c'est l'usage dans de nombreux langages. L'invocation de la superméthode s'écrit :

super.sélecteur(paramètre$_1$, ..., paramètre$_n$)

En héritage simple, ce principe ne pose aucun problème. La superméthode d'une méthode appartenant à une classe donnée est la méthode homonyme recherchée dans la hiérarchie de cette classe.

Dans le cas d'héritage multiple, il peut y avoir conflit de nom entre deux méthodes appartenant à des classes héritées. Le choix de la superméthode est soit lié au parcours du graphe d'héritage, soit établi par le programmeur qui explicite par exemple un chemin. On retrouve ainsi les deux possibilités exposées dans le cas d'un conflit d'héritage.

Exemple

Si l'on considère que le prix de vente d'une chaîne hi-fi inclut le coût de la garantie, la méthode *prixDeVente* de la classe *Article* devient une superméthode de son homonyme de la classe *ChaîneHiFi.*

```
class ChaîneHiFi extends Article
{

        private Float coûtGarantie ;

        public void enregistrerGarantie ()
        {
                …
        }

        // calcule le prix de vente d'une chaîne haute-fidélité
        public Float prixDeVente ()
        {
                return (super.prixDeVente() + this.coûtGarantie) ;
        }

        // autres méthodes
        …
}
```

Comme ***this***, ***super*** désigne l'objet receveur, mais le processus de recherche du sélecteur qui lui est associé fonctionne différemment. La méthode activée est la première méthode de sélecteur *prixDeVente* rencontrée dans la hiérarchie de la classe

ChaîneHiFi (conformément au parcours du graphe d'héritage), celle-ci exclue. Il s'agit donc de la méthode appartenant à la classe *Article*.

Remarque

Une classe a toujours accès aux attributs et méthodes publiques de ses classes ancêtres par l'objet courant ***this***. Par rapport au calcul de la vente d'une chaîne haute-fidélité, ***super***.*prixDeVente()* référence *prixDeVente* de la classe *Article*, alors que ***this***.*coûtGarantie()* désigne l'attribut *coûtGarantie* de la classe *ChaîneHiFi*. Il convient donc de réserver l'usage du mot-clé ***super*** à un appel à la superméthode.

2. LE POLYMORPHISME

2.1. Définition

Etymologiquement, polymorphisme signifie plusieurs (*poly*) formes (*morphe*). C'est donc la capacité pour une entité à prendre plusieurs formes.

En programmation, cette notion n'est pas nouvelle. On parle ainsi de polymorphisme *ad hoc* dans le cas de conversions implicites. Présentes dans de nombreux langages, au moins pour des types de base, ces conversions interviennent chaque fois que l'appel d'une opération est validé malgré des types formels et effectifs distincts, par exemple pour un réel attendu alors qu'un entier est transmis.

Un autre polymorphisme *ad hoc* consiste à spécifier une même opération pour des types différents. C'est la notion de surcharge qui permet au compilateur de choisir l'opération à utiliser en fonction du type des paramètres. Par exemple, l'addition sur des réels, sur des entiers, ou pour deux matrices….

Lorsqu'un même code s'applique à n'importe quel type, le polymorphisme est dit paramétrique. Le plus souvent, il s'agit de paramètres supplémentaires à transmettre à l'opération. Cette technique, appelée généricité, permet de construire des familles de types. Les types abstraits génériques en sont une bonne illustration, par exemple une pile dont les éléments sont d'un type quelconque.

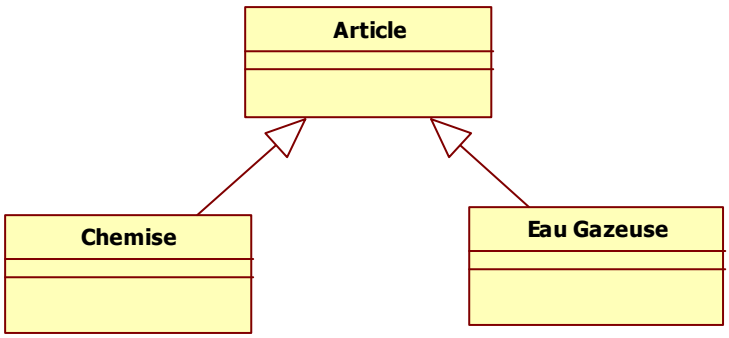
En programmation par objets, d'autres formes de polymorphismes s'ajoutent, notamment le polymorphisme d'inclusion et le polymorphisme de redéfinition (ou polymorphisme d'héritage). Nous examinons tout d'abord la notion d'attachement polymorphe.

2.2. L'attachement polymorphe

L'attachement polymorphe résulte de l'inclusion des extensions : les instances d'une classe sont aussi des instances de ses superclasses. Ainsi, tout objet d'une classe *A* peut désigner en cours d'exécution un objet d'une sous-classe de *A*.

Exemple

Reprenons le graphe d'héritage suivant :



Grâce à l'héritage, les chemises et les eaux gazeuses sont définies comme des sortes d'articles. Ce sont donc aussi des articles. Il semble alors normal de pouvoir utiliser un objet de la classe *Chemise* ou de la classe *EauGazeuse* chaque fois qu'un objet de la classe *Article* peut l'être.

Soient deux objets *a* et *c* dénotant respectivement un article et une chemise :

```
Article a ;
Chemise c ;
c = new Chemise ("Chemise", 80.0, 1, rouge) ;
```

Les écritures suivantes sont valides :

```
a = c ;
a = new EauGazeuse ("EauGazeuse", 1.0) ;
```

Ces instructions affectent à une entité désignant un article la valeur d'une entité désignant, dans le premier cas, une chemise et dans le second, une eau gazeuse.

Ces affectations où le type de la source (la partie droite) diffère du type de la cible (la partie gauche) sont appelées attachements polymorphes. Outre l'affectation, on retrouve cette notion dans le cas des transmissions de paramètres où le paramètre formel d'une méthode peut être remplacé par un paramètre effectif dès lors qu'il est sous-type (sous-classe) du paramètre formel.

En pratique, au niveau interne, l'identité d'un objet est souvent une référence vers l'objet[11]. Ainsi, après exécution de l'instruction *c = **new** Chemise (…)*, la variable *c* pointe vers la chemise créée. La référence de cette chemise peut être mémorisée dans un objet de type *Article* en écrivant simplement *a = c*. De même, *a*, de type *Article*, désigne une eau gazeuse lorsqu'on écrit *a = **new** EauGazeuse (…)*.

Les affectations polymorphes données en exemple sont légitimes : la structure d'héritage nous permet de considérer une instance de *Chemise* ou d'*EauGazeuse*

[11] Contrairement à C++, le programmeur Java ne manipule pas explicitement de pointeur, ni de référence visible… Mais tout est référence ! A tel point qu'il faut utiliser la méthode *clone* de la classe *Object* pour réaliser la copie d'un objet. Les affectations et les comparaisons d'objets ont en Java une sémantique de référence.

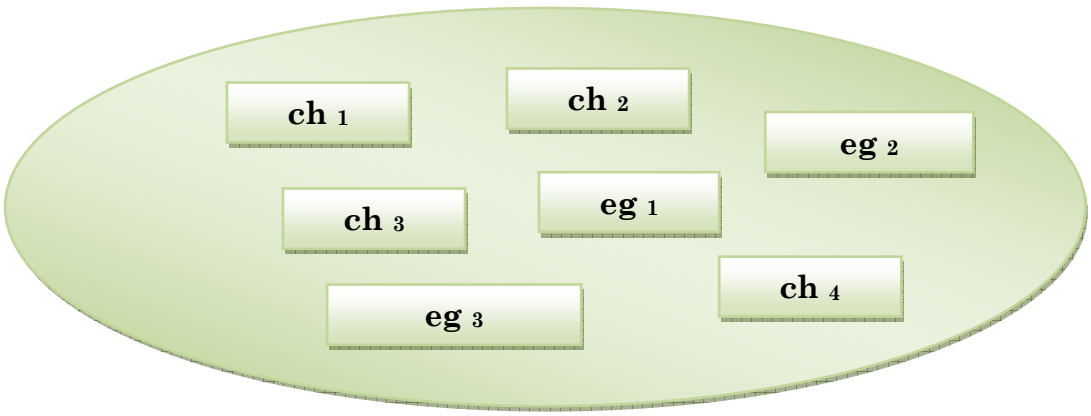
comme une instance d'*Article*. Nous disons que le type de la source est conforme au type de la cible. Dans la direction opposée, comme $c = a$, l'affectation (appelée affectation inverse) n'est en général pas valide, sauf si nous avons l'assurance que a référence bien à l'exécution une chemise. De plus, cette opération nécessite un transtypage (***cast***) de l'objet a pour toute invocation d'une méthode spécifique à c.

2.3. Le polymorphisme d'inclusion

L'héritage fait de chaque objet une entité polymorphe pouvant désigner des objets de types différents au cours d'une même exécution. Cette propriété est appelée polymorphisme d'inclusion.

Ainsi, d'après l'attachement polymorphe, tout objet d'une classe *A* peut désigner en cours d'exécution un objet d'une sous-classe de *A*. Cette propriété d'inclusion résulte de l'équation classe = type, et donc de l'équation sous-classe = sous-type[12]. L'objet ***self*** peut référencer tout sous-objet d'une hiérarchie d'héritage ; il en est de même lorsque l'objet est argument ou résultat d'une méthode.

On peut donc, avec le même code, traiter une famille d'objets en ignorant la classe d'appartenance de chacun des objets du problème, à condition de mettre en place un héritage d'interface. En imaginant un code traitant de façon générique d'articles, une chemise ou une eau gazeuse pourra être substituée à une variable d'instance de type *Article*. L'espace du problème en termes d'objets peut ainsi se caractériser par l'ensemble des données *ch* (Chemise) et *eg* (Eau Gazeuse) ci-dessous :



Exemple

Comme indiqué dans le code de la classe *Hypermarché* et considérant la hiérarchie d'héritage des articles du paragraphe précédent, on peut définir un *stock* de type *Tableau<Article>*. Chaque élément du tableau peut ainsi désigner un objet dont la classe est *Article* ou un de ses descendants c'est-à-dire une *Chemise*, une *EauGazeuse*, ou tout type d'article. On dit alors que le tableau *stock* est un tableau polymorphe.

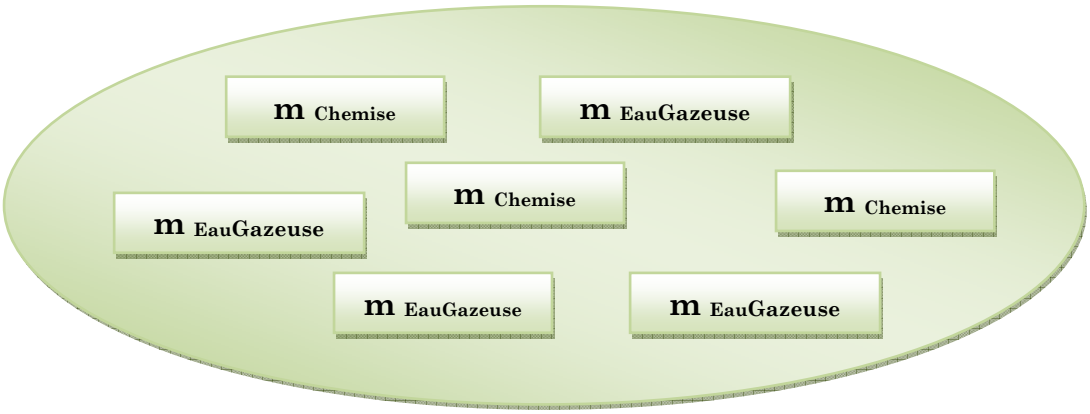
[12] Ces équations ne sont qu'approximatives…

2.4. Le polymorphisme de redéfinition (ou polymorphisme d'héritage)

Il concerne des méthodes appartenant à des classes différentes ayant le même sélecteur. L'envoi de message avec ce sélecteur à des instances de ces classes invoquera la méthode idoine en fonction du type de l'objet receveur.

Considérons un objet *o*, instance de la classe *A* et une méthode *m* spécifiée dans *A*. Lorsque la méthode *m* est redéfinie dans les différentes sous-classes de *A*, l'invocation *o.m* sélectionne (automatiquement) la méthode *m* en fonction du type de l'objet *o*. On dit ainsi que la méthode sélectionnée dépend du type de l'objet à l'exécution. Le mécanisme permettant le choix de la méthode en fonction du type de l'objet est appelé mécanisme de liaison dynamique.

En imaginant un code traitant de façon générique d'articles, les appels à une méthode *m* spécifiée dans la classe *Article* et redéfinie dans l'une des deux classes *Chemise* ou *EauGazeuse* donnera l'impression de n'avoir qu'une seule méthode *m* s'appliquant à des chemises ou à des eaux gazeuses. L'espace du problème en termes de méthodes peut ainsi se caractériser par l'ensemble des traitements *m* ci-dessous :



Exemples

1) Pour établir un prix de vente, nous avons respectivement défini dans les classes *Article* et *EauGazeuse* :

```
// calcule le prix de vente d'un article
public Float prixDeVente ()
{
      return (this.prixHT * 1.2) ;                    // TVA = 20 %
}
```

```
// redéfinition du calcul du prix de vente de la classe Article
// pour une Eau Gazeuse
public Float prixDeVente ()
{
      return (this.getPrixHT () * 1.055) ;            // TVA = 5,5 %
}
```

Dès lors, l'affectation

```
prix = this.stock.getIème(i).prixDeVente()
```

invoque la méthode *prixDeVente* d'*Article* ou d'*EauGazeuse* selon que l'objet ***this****.stock.getIème(i)* référence à l'exécution un article, une chemise ou une eau gazeuse. S'il s'agit d'une chemise, la méthode invoquée sera celle d'*Article* ; dans le cas d'une eau gazeuse, la redéfinition de la méthode *prixDeVente* sera prise en compte avec un taux de TVA de 5,5 %. En conclusion, le *prixDeVente* dépend de la forme de l'objet à l'exécution.

2) Soit à calculer la masse salariale mensuelle de l'hypermarché. En supposant que le personnel soit représenté par une liste d'employés, on définit comme suit la méthode *masseSalariale* qui cumule dans *s* le salaire de chacun des employés sans en connaître le type (*directeur commercial, comptable, caissière, …)* :

```
// détermine la masse salariale mensuelle
public Float masseSalariale ()
{
        Float s = 0.0;
        for (Int i = 0 ; i < this.personnel.taille() ; i++)
                s = s + this.personnel.getIème(i).salaire() ;
        return (s) ;
}
```

Dans cette écriture, *taille* est un accesseur permettant de connaître le nombre d'éléments de la liste et *getIème* accède à l'élément de rang *i*.

On notera que l'ajout d'une catégorie d'employé n'affecte pas le code de la méthode *masseSalariale* : pour un chef de rayon par exemple, il conviendra de définir la classe définissant ce type d'employé, *ChefDeRayon*, sous-classe d'*Employé*, et sa méthode *salaire*.

Glossaire

accesseur : méthode permettant l'accès en consultation ou en modification à un attribut.

attribut : donnée d'une classe ; à cette donnée sera associé un champ de l'objet.

classe : description d'une famille d'objets similaires. Une classe est décrite par des données appelées attributs et par des sous-programmes appelés méthodes.

constructeur : méthode d'initialisation des instances d'une classe.

délégation : mécanisme de communication qui permet à un objet fournisseur de confier une tâche à un objet sous-traitant, sans le signaler à son objet client.

encapsulation : synonyme de *masquage d'information*.

extension : ensemble des instances d'une classe.

généricité : dans le cadre de la programmation par objets, possibilité pour une classe de définir conceptuellement un ensemble de types. On parle aussi de polymorphisme paramétrique.

héritage : mécanisme permettant le partage et la réutilisation de propriétés entre les objets.

héritage de développement : héritage qui conduit à partager la représentation et le code de la superclasse.

héritage d'interface (ou de sous-typage) : héritage qui conduit à pouvoir remplacer dynamiquement tout objet par un objet d'une sous-classe, sans erreur de type à l'exécution.

héritage par enrichissement : héritage qui consiste à compléter la classe ancêtre par ajout d'attributs et/ou de méthodes.

héritage par substitution : héritage qui consiste à redéfinir une méthode héritée. On parle alors de masquage.

intention : ensemble des propriétés d'une classe.

interface : ensemble des méthodes applicables à un objet connues à l'extérieur de la classe.

liaison (dynamique) : mécanisme (souvent dynamique) permettant de rechercher le sélecteur d'une méthode à appliquer lors d'un envoi de message (conformément au parcours du graphe d'héritage).

masquage : phénomène associé à la redéfinition qui implique que la propriété redéfinie masque ses définitions dans les superclasses.

masquage d'information : principe selon lequel un objet est entièrement défini par l'ensemble des opérations qui lui sont applicables ; la réalisation de ces opérations et la représentation physique de l'objet restent cachées et inaccessibles au monde extérieur.

message : requête adressée à un objet demandant l'exécution d'une opération (accès à un champ, exécution d'une méthode).

méthode : sous-programme d'une classe désigné par un sélecteur.

objet : entité possédant des valeurs spécifiques pour chaque attribut de sa classe d'appartenance (état de l'objet) et réagissant conformément à ses méthodes (comportement). On parle aussi d'instance ou de représentant de classe.

objet composite : objet défini par l'assemblage d'autres objets, appelés ses parties ou ses composants. On parle alors de relation client.

objet receveur : objet ayant en charge l'interprétation d'un message.

polymorphisme : capacité pour une entité à prendre plusieurs formes à l'exécution.

polymorphisme d'héritage : polymorphisme associé à la redéfinition de méthode.

polymorphisme d'inclusion : polymorphisme associé au sous-typage.

polymorphisme paramétrique : synonyme de *généricité*.

propriétés (d'une classe) : attributs (données) et méthodes (procédures ou fonctions) de ses instances.

relation d'instanciation : opération qui permet de créer un objet à partir de sa classe d'appartenance.

self : désignation de l'objet receveur dans un envoi de message.

super : désignation de l'objet receveur d'un envoi de message (comme *self*), mais recherche de l'opération spécifique à l'objet dans la (ou les) classe(s) ancêtre(s) (conformément au parcours du graphe d'héritage). Autorise l'accès à la superméthode.

superclasse : classe ancêtre d'une classe.

superméthode : méthode d'une classe ancêtre masquée par une substitution.

type : ensemble d'opérations et de valeurs. Pour les objets, on distingue le type statique, annotation dans les programmes, du type dynamique, type des valeurs à l'exécution.

Bibliographie

[1] Mokrane Bouzeghoub, Georges Gardarin, Patrick Valduriez
Les Objets
Eyrolles, 1997

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns – Catalogue de modèles de conception réutilisables
Vuibert, 1999

[3] Martine Gautier, Gérald Masini, Karol Proch
Cours de programmation par objets, Principes et applications avec Eiffel et C++
Editions Masson, 1996

[4] Gérald Masini, Amadeo Napoli, Dominique Colnet, Daniel Léonard, Karl Tombre
Les langages à objets
Langages de classes, langages de frames, langages d'acteurs
InterEditions, 1989

[5] Bertrand Meyer
Conception et programmation orientées objet
Editions Eyrolles, 2000

[6] Pierre-Alain Muller, Nathalie Gaertner
Modélisation objet avec UML
Eyrolles, deuxième édition, 2000

[7] Jean-François Perrot
Langages à objets
Traité Informatique, Techniques de l'Ingénieur