Université de Montpellier / Faculté Des Sciences

Modélisation et Programmation par Objets (2e partie)

2019-2020 - HLIN 505

Cours 2
Héritage, Spécialisation/généralisation
Redéfinition et surcharge statique
Spécialisation naturelle vs. substituabilité

Enseignants

Marianne Huchard, Stéphane Bessy, Marie-Laure Mugnier,

Clémentine Nebut, Abdelhak-Djamel Seriai

Ce document est un recueil de notes de cours. Il peut développer des aspects vus en cours plus succinctement ou inversement vous aurez en cours plus de détails ou d'autres exemples sur certains points. Si vous y relevez des erreurs, n'hésitez pas à nous les signaler afin de l'améliorer.

1 L'héritage, motivation par un cas d'étude

L'héritage est une caractéristique très spécifique des langages à objets. Il est apparenté à la spécialisation/généralisation que l'on trouve dans les langages de modélisation à objets comme UML. Il se base sur un mode de pensée très courant qui est le raisonnement par classification ou classement. Ses principaux atouts sont :

- De permettre d'exprimer des hiérarchies de classification des concepts rencontrés dans un programme, allant des concepts les plus techniques (menu d'une interface graphique, structure de données comme une pile) aux concepts les plus ancrés dans le métier ciblé par le logiciel (compte bancaire, client, produit). Ces classifications, si elles sont bien réalisées, clarifient la structure du programme et lui permettent de s'étendre facilement par de nouveaux concepts.
- Une définition par genre et différence qui allège les définitions et rend l'écriture de code plus efficace.
- La possibilité de réutiliser des représentations et des parties de code, de manière orthogonale à d'autres caractéristiques que nous verrons plus loin dans ce cours, comme la généricité.

Ce cours[1] introduit l'héritage pour les étudiants de licence 2 math-info et consiste en des rappels

1. D'après des diapositives de M.-L. Mugnier revisitées et avec l'aide LaTeX de V. Boudet

pour les étudiants de licence 2 informatique sur les principes présentés dans le cours de Modélisation et Programmation par Objets (1) du module HLIN406.

1.1 Solutions alternatives

Nous introduisons tout d'abord un cas d'étude très simple, sur lequel nous expliquerons l'intérêt d'utiliser l'héritage face à d'autres solutions de conception et de programmation. Nous étudions la représentation simplifiée d'un ensemble de produits limité aux livres, aliments et articles d'électroménager. Les produits sont décrits par différents éléments communs comme une référence, une désignation et un prix hors taxe (prix HT). Certaines informations complémentaires sont spécifiques à certains produits :

— pour un livre : l'éditeur et l'année de parution.
— pour un aliment : la date limite de validité.
— pour un article d'électroménager : une durée de garantie.

Certains calculs peuvent être effectués sur les livres, nous citerons ici seulement le calcul du prix Toutes Taxes Comprises (prix TTC). Ce calcul du prix TTC s'effectuera de différentes manières suivant les produits :

— Pour les livres et les aliments, on appliquera une TVA de 5,5%.
— Pour les aliments, on appliquera de plus une réduction lorsque la date de péremption approche.
— Pour les articles d'électroménager, on appliquera une TVA de 19,6%.

1.2 Solution 1 : une unique classe

La figure 1 présente une première solution consistant à n'utiliser qu'une classe pour représenter les trois sortes de produits.

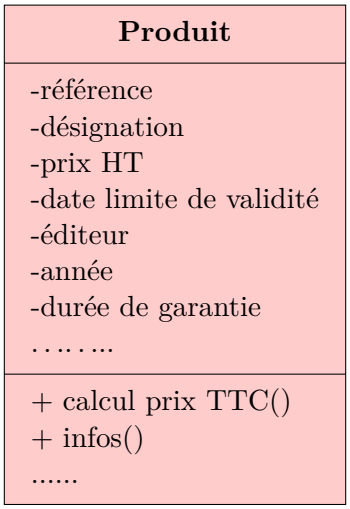

FIGURE 1 – Solution 1 : utilisation d'une unique classe pour représenter les trois sortes de produits

Cette solution présente des inconvénients aussi bien du point de vue de la structure que du point de vue de l'expression des comportements.

Du point de vue de la structure, quel que soit l'objet que l'on va créer, cela complexifie la représentation et certains des attributs seront inutiles (mais la place sera cependant occupée en mémoire) :

— éditeur, année et durée de garantie pour les aliments.
— date limite de validité et durée de garantie pour les livres.

— date limite de validité, éditeur et année pour les articles d'électroménager.

Du point de vue du comportement, le fait de ne disposer que d'une classe pour tout représenter va provoquer l'apparition dans le code de ce que l'on appelle des "tests de types". Ils seront parfois difficiles à repérer dans un code, mais le principe est toujours le même. Il consiste à savoir, d'une manière ou d'une autre, quel est le type de l'objet (livre, aliment ou article d'électroménager) pour savoir quel calcul lui appliquer. Le type peut être connu : soit parce qu'il a été explicitement décrit par un attribut de la classe, appelé `type` et dont les valeurs peuvent être par exemple celles d'une énumération ou de simples chaînes de caractères ("livre", "aliment", "article électroménager") ; soit en testant la présence d'une valeur d'un ou plusieurs attributs (par exemple, en testant la valeur de la date limite de validité : si cet attribut contient une date réelle et non pas une valeur fictive destinée à représenter l'absence de valeur, on sait que l'objet est un aliment) ; soit en utilisant l'opérateur `instanceof` qui sera présenté au prochain cours. Dans le cas de nos produits, voici deux exemples de méthodes sujettes à être écrites avec des tests de types.
Pour la méthode double prixTTC(), voici une structure schématique de code avec test de type :

— si (livre ou aliment) { ··· } // appliquer la TVA à 5.5%
— si (aliment){ ··· } // appliquer la réduction due à la date limite de validité
— si (électro.){ ··· } // appliquer la TVA à 19.6%

Pour la méthode String infos(), supposée retourner une chaîne de caractères contenant des informations sur les objets, la structure et les tests seraient légèrement différents :

.... Réf + désignation // pour tous les produits
si (livre) { ... éditeur/an}
si (aliment) { ... date limite}
si (électro.) { ... garantie}

Pour synthétiser, cette solution présente plusieurs inconvénients : de la place perdue avec des espaces réservés inutilement dans les objets, un code complexe à lire (surtout si les tests de type ne sont pas très explicites) et enfin une difficulté à étendre l'ensemble des produits par de nouveaux produits, qui demanderaient, outre d'ajouter de nouveaux attributs, de mettre également à jour des méthodes déjà complexes pour ajouter les nouvelles conditions et le nouveau comportement.

1.3 Solution 2 : une classe par sorte de produit

Dans la deuxième solution, on fait inversement le choix d'avoir une classe par sorte de produit à représenter, cette solution est présentée dans la figure 2. Trois classes sont introduites : `Aliment`, `Livre`, `Electro`. Chacune a les attributs nécessaires à la représentation de ses instances et une version de méthode adaptée. On peut observer en rouge des répétitions qui apparaissent alors : les trois attributs `référence`, `désignation` et `prixHT` et les signatures (sur la figure, ces signatures sont restreintes au nom pour simplifier) `prixTTC` et `infos`.

Ces répétitions peuvent avoir des conséquences à différentes étapes du cycle d'écriture du programme. Au moment de l'écriture initiale du code, des incohérences peuvent être introduites comme : donner un type différent à deux attributs de même nom (chaîne de caractères pour `référence` de `Aliment` et entier pour `référence` de `Livre`) ; donner des noms légèrement différents comme `référence`, `reference`, ou encore `référence_Aliment`, `référence_Livre` pour des données qui dans cet exemple devraient être d'un même type et avec le même nom, s'il s'agit par exemple du système de références précis d'un magasin donné. On peut avoir le même phénomène sur les méthodes, ce qui va de plus réduire les opportunités de polymorphisme. Nous y reviendrons plus loin dans le cours, mais vous pouvez déjà retenir que l'un

des ressorts de la programmation par objets, qui permettra notamment l'écriture de code générique et extensible, est une dénomination adéquate et uniforme des méthodes, par exemple toutes les méthodes d'impression s'appelant d'un même nom, tel que `print`.

L'autre moment du développement où cela posera des problèmes est lors des opérations d'évolution et de correction car il deviendra nécessaire d'identifier tous les points du code représentant la même entité à modifier. Par exemple, si on désire changer le type de `prixHT` pour `double`, alors que le type initial était `float`, il faudra retrouver tous les attributs `prixHT`. Dans cette sorte de modification, le programmeur risque d'oublier des sites de déclaration de `prixHT`, ajoutant des incohérences au programme.

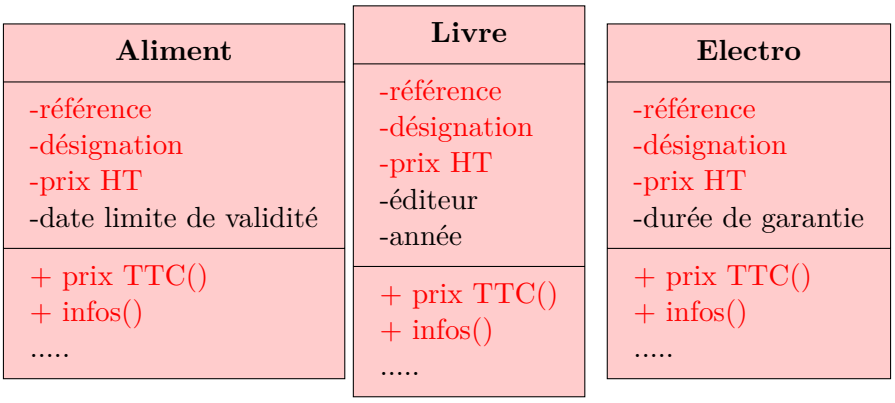

FIGURE 2 – Solution 2 : une classe par sorte de produit

1.4 Solution 3 : une hiérarchie d'héritage

La troisième solution (figure 3) consiste à organiser les trois classes dans une hiérarchie de classification. Pour cela, on identifie plusieurs concepts importants qui seront codés sous forme de classes. Il peut y avoir des variantes d'organisation, bien entendu, l'essentiel étant de produire une classification qui a du sens pour le domaine que l'on représente. Ici nous proposons de travailler avec plusieurs concepts supplémentaires.

- Le concept de `Produit` sera le concept le plus général de notre classification, il traduit la sémantique commune aux trois classes initiales. Il introduira ainsi tout ce qui est commun : attributs communs, signatures de méthodes, parties de méthodes repérées comme étant nécessaires dans tous les cas.
- Le concept `ProduitTNormal` représentera tous les produits pour lesquels le calcul du prix TTC se fera avec un taux normal (19.6). Parmi ses sous-concepts on peut ainsi observer la classe `Electromenager`.
- Le concept `ProduitTReduit` représentera tous les produits pour lesquels le calcul du prix TTC se fera avec un taux réduit de 5.5. Parmi ses sous-concepts on peut ainsi observer les classes `Livre` et `Aliment`.

On voit apparaître quelques premières raisons de spécialiser un concept :

- Les objets de son extension peuvent se diviser en sous-groupes partageant des attributs, comme les objets `Produit` qui se divisent suivant s'ils ont une date limite de validité (sous-groupe des aliments), un éditeur et une année de parution (sous-groupe des livres), une durée de garantie (sous-groupe des articles d'électroménager). Dans d'autres contextes de spécialisation, vous verrez apparaître des divisions sur des valeurs d'attributs, comme les véhicules, qui ont tous l'attribut `milieu de déplacement`. Les véhicules se subdivisent alors en véhicules terrestres, ou aquatiques, ou aériens, ou amphibies suivant la valeur de cet attribut.

— Les objets de son extension peuvent se diviser suivant des comportements différents, comme les objets `Produit` qui se divisent en deux sous-groupes (produit à taux normal versus produit à taux réduit) suivant le calcul du prix TTC.

Dans les sections qui viennent nous allons développer cette solution en en montrant les principales caractéristiques.

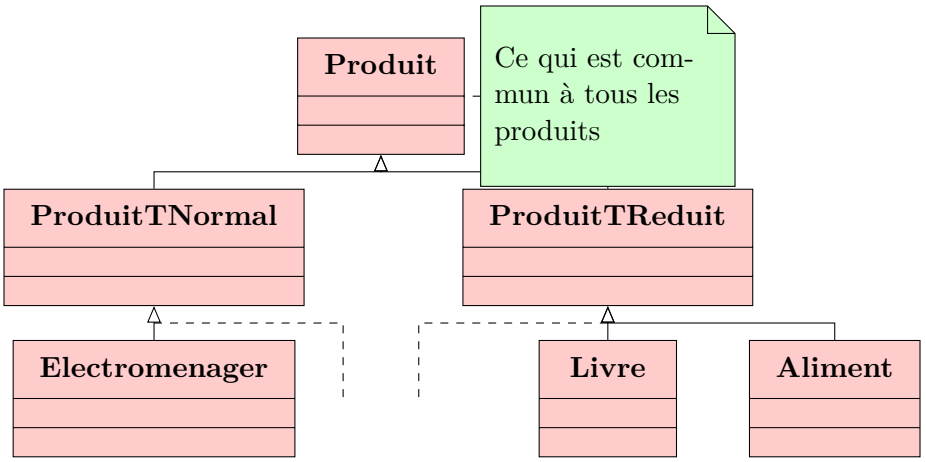

FIGURE 3 – Solution 3 : une hiérarchie de classes

Nota : En UML, il est bienvenu de compléter ce type de diagramme de spécialisation par des critères de classification (*power-types*) et des contraintes (`complet`, `incomplet`, `disjoint`, `chevauchement`). Reportez-vous au cours de HLIN406 pour ces éléments.

2 Principaux éléments sur l'héritage

2.1 Une définition simplifiée

L'héritage (en programmation Java) consiste à construire une classe `B` (sous-classe) à partir d'une classe `A` (super-classe) en :

— ajoutant des attributs,
— ajoutant des méthodes,
— redéfinissant des méthodes.

La figure 4 montre schématiquement en notation UML ce qui sera transcrit en Java. Nous verrons les éléments de code Java nécessaires à cela plus loin. La classe `A` possède l'attribut `m` et les méthodes `f()` et `g()`. La classe `B` possède par héritage l'attribut `m` et déclare en plus l'attribut `n`. Elle hérite de la méthode `g()`, elle redéfinit la méthode `f()` et elle déclare une nouvelle méthode, `h()`.

Une instance de `B` sera composée des attributs `m` et `n` et la redéfinition de `f()` dans B masquera la définition de `f()` qui a été effectuée dans `A`.

Pour ce qui est des appels de méthodes, voici quelques cas représentatifs :

1. `B b = new B();`
2. `b.f(); // appelle f de B`
3. `b.g(); // appelle g de A`
4. `b.h(); // appelle h de B`

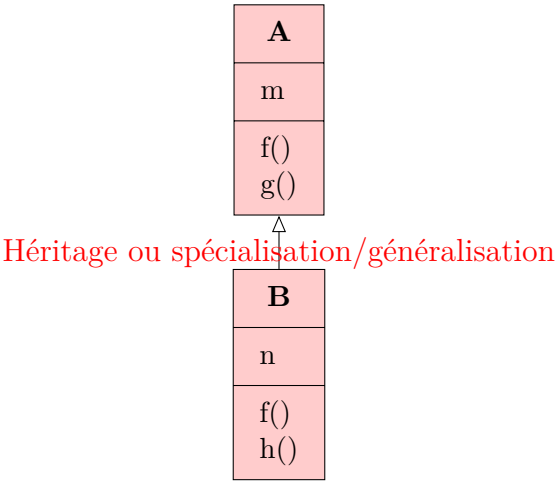

FIGURE 4 – Définition d'une classe `B` à partir d'une classe `A`

Nous reviendrons plus tard sur le mode de vérification des appels et leur exécution. Ici nous observons une version simple qui montre que lors de l'appel de la méthode `f` sur un objet de la classe `B`, la version écrite dans `B`, qui masque celle écrite dans `A` est appelée ; lors de l'appel de la méthode `g`, la version héritée de `A`, qui est l'unique version existante, est appelée ; lors de l'appel de la méthode `h`, la version écrite dans `B`, qui est l'unique version existante, est appelée.

2.2 Première approche pour les produits

Nous revenons sur les classes représentant les produits. La classe la plus haute, qui représente le concept général de produit est écrite ci-dessous partiellement dans une première version. Les attributs sont les attributs communs à tous les produits (on dit qu'ils sont factorisés dans cette classe). Un constructeur permet d'initialiser ces attributs avec des valeurs passées en paramètre.

```
public class Produit{
    // attributs
    private String reference;
    private String designation;
    private double prixHT;

    // constructeur
    public Produit(String r, String d, double p)
    {reference=r; designation=d; prixHT=p;}

    // Methode d'acces a l'attribut reference
    public String getReference()
    {return reference;}
    public void setReference(String r)
    {reference=r;}

    // autres methodes d'acces laissees en exercice

    // quelques methodes
    public double leprixTTC(){}    // vide
    public String infos(){...}
```

```
}
```

La méthode `double leprixTTC()` peut être laissée vide car son code est différent pour les trois classes (une partie est cependant commune à deux des classes, nous verrons plus loin quelle forme cela prend).

La méthode `String infos()` sera décrite plus loin.

2.3 Classe et méthode abstraite

En réalité, dans la classe `Produit`, la méthode leprixTTC() a un comportement inconnu. Une manière de le déclarer est de ne pas lui donner de corps (la signature se termine par un `;`) et de la précéder du mot-clef `abstract`. Cela empêche de créer des objets produits directement, ainsi la classe `Produit` devient elle-même également abstraite, ce que l'on voit syntaxiquement dans son entête qui commence par le même mot-clef `abstract`.

```
abstract public class Produit{
    // attributs
    private String reference;
    ...

    // quelques methodes
    abstract public double leprixTTC();
    public String infos(){...}
}
```

En UML, cela se traduit soit en ajoutant explicitement le même mot-clef `abstract` sur le modèle, soit en mettant les noms de la classe et de la méthode concernées en italiques (voir la figure 5 où la deuxième solution est présentée).

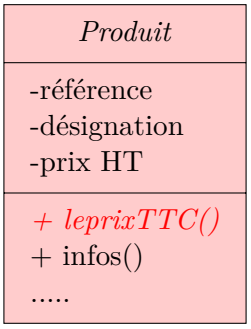

FIGURE 5 – Classe et méthode abstraites en notation UML

RÈGLE 1 (CLASSE ET MÉTHODE ABSTRAITES)

- *Toute classe possédant une méthode abstraite est abstraite*
- *Une classe peut être abstraite même si elle n'a pas de méthode abstraite*

Remarquons que "posséder" peut signifier :
- définir
- hériter de

Une classe peut donc être abstraite si elle hérite une méthode abstraite d'une classe abstraite. On peut ainsi observer dans la figure 6 une classe `A` qui est abstraite car elle déclare la méthode `f()` abstraite, une classe `B` qui est abstraite car elle hérite de la méthode `f()` abstraite et enfin une classe `C` qui est concrète car elle redéfinit `f()` en lui donnant un corps. Elle est dite concrète et n'est plus en italiques sur le schéma.

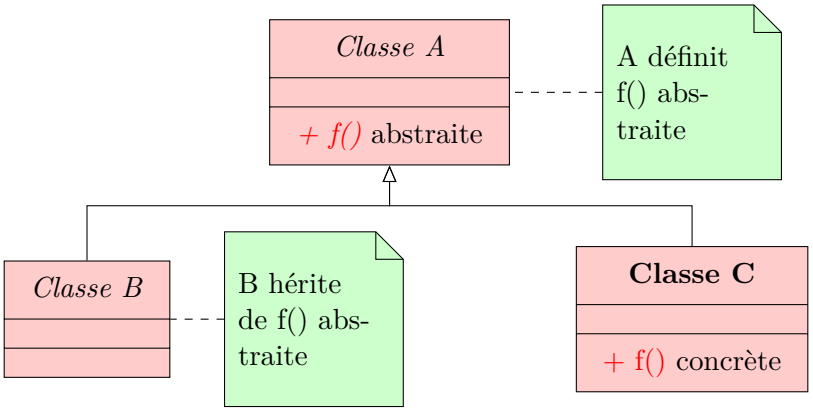

FIGURE 6 – Classe et méthode abstraites et concrètes en notation UML

2.4 Factorisation de code

Dans l'écriture de la méthode `lePrixTTC`, nous n'avons pas trouvé de partie de calcul commune, ce qui nous a conduit à la laisser vide ou à la rendre abstraite. Si nous cherchons maintenant à écrire la méthode `String infos()`, nous pouvons cette fois déterminer pour tous les produits un ensemble d'instructions commun. Ces instructions sont dites "factorisées" dans la méthode `String infos()` de `Produit`. Elle est présentée ci-dessous et consiste à concaténer dans une chaîne de caractères les attributs référence, désignation, prix hors taxe et la valeur du prix TTC obtenue en appelant la méthode `leprixTTC()`. Cette dernière méthode est abstraite pour la classe `Produit`, ce qui peut sembler troublant. Cependant, lorsqu'elle sera appelée, ce sera pour un objet d'une classe concrète, dans notre cas `Aliment`, `Electromenager` ou `Livre`, qui aura défini un calcul dans `leprixTTC()`. Ce sera mis en œuvre par le procédé de liaison dynamique, que nous préciserons un peu plus loin dans ce cours.

```
public String infos(){ // dans la classe Produit
        String s = reference + ' ' + designation;
        s += '\n' + "prix HT: " + prixHT;
        s += '\n' + "prix TTC: " + leprixTTC();
        return s;
}
```

2.5 Déclarer la relation d'héritage

La relation d'héritage entre classes, que nous avons représentée jusqu'ici par une flèche à la pointe fermée (notation UML), s'écrit `extends` dans l'entête de la sous-classe (code Java). Par exemple :

```
public class ProduitTNormal extends Produit{
        ...
}
```

Cette déclaration `extends` est représentée en vert sur la figure 7.

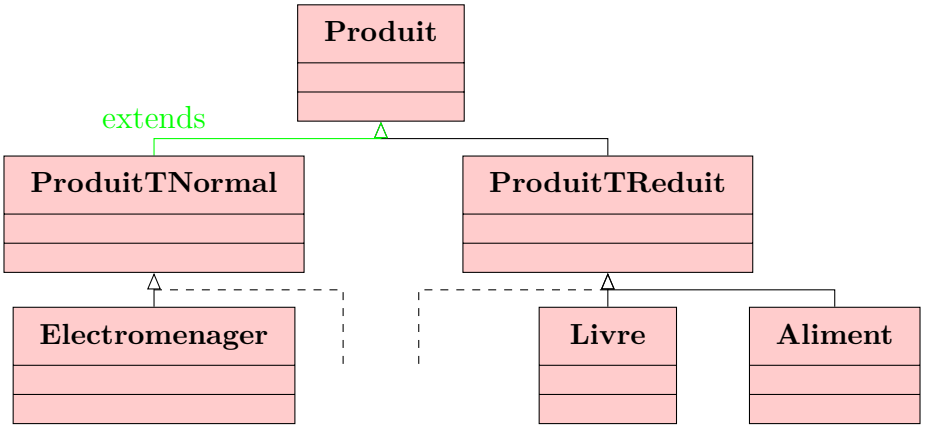

FIGURE 7 – Hiérarchie d'héritage avec mise en évidence de la relation d'héritage traduite en Java par l'expression `public class ProduitTNormal extends Produit`

2.6 Une première sous-classe

La première sous-classe que nous écrivons partiellement est la classe `ProduitTNormal`. Son rôle est de concrétiser la méthode leprixTTC dans une version simplifiée. Une version recommandée, mais un peu plus complexe, utiliserait une valeur partagée TauxNormalTVA et vous est laissée à titre d'exercice [2]. La formule de calcul du prix TTC pour les produits à taux normal est :

$$prixTTC = prixHT + (prixHT \times 19,6\%)$$

La classe `ProduitTNormal` s'écrit ainsi partiellement :

```
public class ProduitTNormal extends Produit{
        ...
        public double leprixTTC(){
                return this.getPrixHT() * 1.196;
        }
}
```

On peut noter dans ce code l'appel de la méthode d'accès `getPrixHT()`, héritée de la superclasse `Produit`.

Il n'y a pas d'attribut dans la classe `ProduitTNormal`, donc on n'y trouvera pas de nouvelles méthodes d'accès. Elle devra contenir un ou plusieurs constructeurs, sujet qui sera abordé un peu plus loin.

2.7 Responsabilité des classes sur leur structure

Nous descendons maintenant dans l'autre branche de la hiérarchie d'héritage pour étudier l'écriture de la classe `Livre`. Une instance de Livre a 5 attributs :

— 3 hérités de Produit (on ne les répète pas !)

— 2 attributs qui lui sont propres : éditeur et année

Cela donne un début de code suivant pour la classe `Livre` :

2. Une solution utilisant un attribut `static double TauxNormalTVA` muni de ses accesseurs est recommandée dans un souci de maintenance du code. En effet, elle évite d'enfouir de l'information métier (un taux de TVA) dans le code d'une méthode. La faire figurer sous forme d'attribut la positionne dans la partie déclaration donc cela permet de l'identifier facilement.

```
public class Livre extends ProduitTReduit{
        // attributs
        private String editeur;
        private int annee;

        // accesseur pour editeur
        public String getEditeur()
                {return editeur;}
        public void setEditeur(String e)
                {editeur = e;}
        // idem pour annee (exercice)
}
```

RÈGLE 2 *Responsabilité des classes sur leur structure*

- *chaque classe s'occupe des attributs qu'elle définit*
- *pour les attributs hérités, elle délègue à ses super-classes*

Cette règle s'applique à différentes méthodes, ici ce sera plus particulièrement :

- le(s) constructeur(s)
- la méthode `String infos()`

Nous l'illustrons sur la méthode `String infos()`, qui, pour un objet livre, retourne :

- le résultat de la méthode infos() héritée
- la présentation de éditeur et année

Donnant le code suivant :

```
public String infos(){
        return super.infos() + '\n' + editeur + '␣' + annee;
}
```

Dans ce code, vous voyez apparaître un nouveau mot-clef, `super` qui sert à appeler la méthode héritée et vous est expliqué plus en détails dans la section qui suit. Cet appel de la méthode héritée permet de lui déléguer la partie de la construction de la chaîne d'informations concernant les attributs hérités. Un code alternatif, non recommandé, n'utilisant pas la délégation serait le suivant. Il recopie le code de la méthode de la super-classe en remplaçant les accès aux attributs (privés) par des appels des accesseurs. Si ce code change dans la super-classe, par exemple parce que l'ordre de concaténation change, ou parce que les noms ou les types des attributs changent, il y a tout lieu de penser qu'il devrait être changé ici aussi, demandant un effort supplémentaire et générant un risque d'erreur (comme toute modification sur de l'information dupliquée).

```
//version non recommandee du code de la methode infos
public String infos(){
      String s = reference + '␣' + getDesignation();
        s += '\n' + "prix␣HT:␣" + getPrixHT();
        s += '\n' + "prix␣TTC:␣" + leprixTTC();
       s += '\n' + editeur + '␣' + annee;
       return s;
}
```

2.8 Spécialisation des méthodes et liaison dynamique

La figure 8 présente une hiérarchie d'héritage simple, composée de deux classes `A` et `B` possédant différentes méthodes. Les codes de ces méthodes sont indiqués dans les boîtes de commentaires UML

(partie droite de la figure).

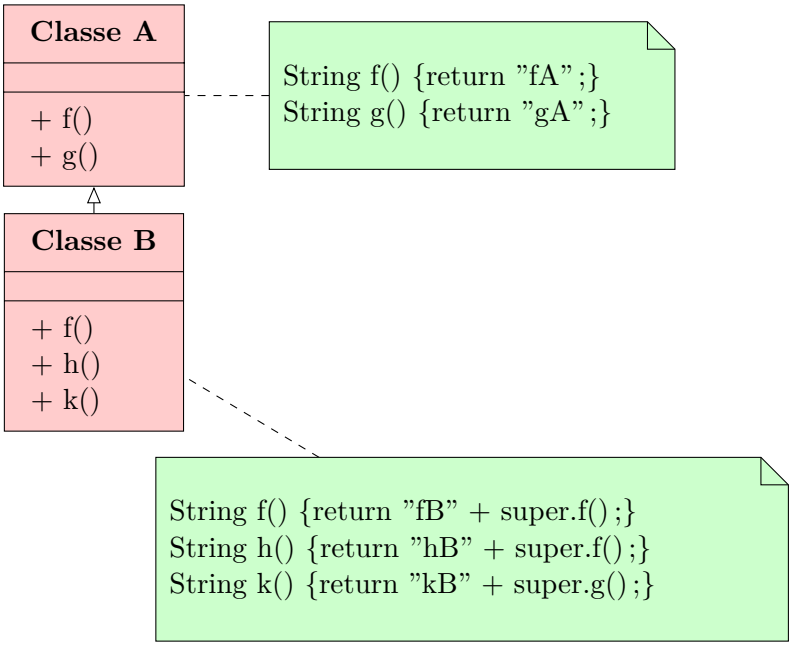

FIGURE 8 – Illustration de l'appel de méthode avec le mot-clef `super`

Arrêtons-nous un moment sur la définition des méthodes de `B` et utilisant `super`

- Dans `f()` : l'expression super.f() consiste à appeler la méthode `f()` définie dans la classe `A` pour compléter son comportement avec des éléments spécifiques aux objets de `B`. C'est une bonne manière de procéder (schéma de programmation par spécialisation de comportement).
- Dans `h()` : l'expression super.f(), qui appelle la méthode `f()` définie dans la classe `A`, pose un problème de conception, car une méthode `f()` existe dans la classe `B`, sûrement plus adaptée pour les objets de type `B` et cette méthode `f()` spécialisée ne sera pas appelée. Il faut exclure ce type d'écriture sauf cas très particulier (voir 3.8).
- Dans `k()` : l'expression super.g() consiste à appeler la méthode `g()` définie dans la classe `A`. Cependant cette méthode est héritée (et non spécialisée dans `B`), il faut donc simplement écrire g().

Ces remarques nous amènent à la règle suivante.

RÈGLE 3 (UTILISATION DE `SUPER`)
On n'utilise généralement ***`super.f()`*** *que dans une nouvelle définition de* ***`f()`****.*

Etudions maintenant la manière dont les méthodes seront appelées. La programmation par objets a introduit un mode de liaison appelé ”liaison dynamique”. Nous l'illustrons sur les deux instructions suivantes, la première déclare et crée un objet, la seconde imprime le résultat de l'appel de la méthode `f()` sur cet objet.

```
A ab = new B();
System.out.println(ab.f());
```

Deux règles sont importantes à retenir dans le cas des langages à objets statiquement typés de la famille de Java (pour les langages typés dynamiquement, le mécanisme est un peu différent car le compilateur ne dispose pas de marque de type).

Règle 4 *Compilation (étape statique)*

- *Le compilateur vérifie que l'objet* `ab` *a une classe proposant l'opération* `f()` *; la liste de types de paramètres compte pour cette recherche.*
- *Il utilise le type de la variable, encore appelé type statique (ici* `A`*), pour cette vérification.*

Règle 5 *Interprétation (exécution, étape dynamique)*

- *L'interprète recherche la méthode* `f()` *la plus spécialisée pour l'objet* `ab`*.*
- *Il utilise le type de l'objet, donné lors de la création (*`new`*), encore appelé type dynamique (ici* `B`*), pour cette recherche.*

Dans notre exemple, le programme affichera `fB fA`.

Observons à présent la hiérarchie de la figure 9.

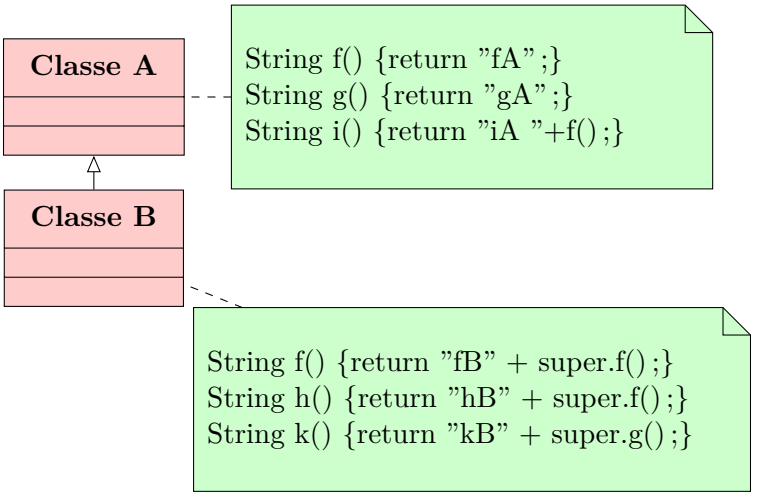

Figure 9 – Compilation et liaison dynamique

Pour cette figure 9, prenons une première séquence d'instructions.

1. B b1 = new B() ;
2. System.out.println(b1.f()) ;
3. System.out.println(b1.g()) ;
4. System.out.println(b1.h()) ;
5. System.out.println(b1.k()) ;
6. System.out.println(b1.i()) ;

Analysons leur résultat.

1. `b1` a pour type statique `B` et pour type dynamique `B`. Cela signifie que `this`, dans les méthodes appelées, aura toujours pour type dynamique `B` (mais `this` a pour type statique celui de la classe dans laquelle la méthode est déclarée).
2. `System.out.println(b1.f());` appelle f() de B, puis f() de A (par super) et affiche fB fA
3. System.out.println(b1.g()) ; appelle g() de A (car il n'y a pas de g() dans B), et affiche gA
4. System.out.println(b1.h()) ; appelle h() de B, puis f() de A (par super) et affiche hB fA

5. System.out.println(b1.k()) ; appelle k() de B, puis g() de B (par super) et affiche kB gA
6. System.out.println(b1.i()) ; appelle i() de A (car il n'y a pas de i() dans B) ; puis l'appel de f() est traité sur `this`. `this` a pour type dynamique B (même si `this` a pour type statique A dans `i()`), ainsi f() est cherchée tout d'abord dans B, ce qui retourne "fB" et appelle à nouveau f() (par super) cette fois sur la classe A, qui retourne fA. L'instruction affiche iA fB fA

Pour la même figure 9, changeons la déclaration de type de la variable (à présent `A`), prenons la même séquence d'instructions et analysons-les.

1. A b2 = new B() ;
2. System.out.println(b2.f()) ; // même résultat que b1.f()
3. System.out.println(b2.g()) ; // même résultat que b1.g()
4. System.out.println(b2.h()) ; // erreur de compilation, car h() n'existe pas dans la classe A (type statique de b2)
5. System.out.println(b2.k()) ; // erreur de compilation, car k() n'existe pas dans la classe A (type statique de b2)
6. System.out.println(b2.i()) ; // même résultat que b1.i()

On observe que le changement de type statique limite les méthodes que l'on peut appeler à celles que la classe correspondant au type statique possède (déclare ou hérite). Par contre, sur une méthode que l'on a pu appeler, cela ne change pas le résultat des appels.

Enfin, dernier point sur l'appel `super`, il faut noter que c'est un mécanisme "anonyme" d'appel de méthode héritée, au sens où on n'indique pas explicitement à partir de quelle classe la méthode héritée doit être cherchée. On indique seulement que la méthode doit être recherchée à partir de la super-classe directe. Sur l'exemple de la figure 10, on n'a aucun moyen d'appeler directement f() de A dans la méthode f() de B. On peut dire également que `super.f()` désigne la prochaine occurrence de f() dans la hiérarchie.

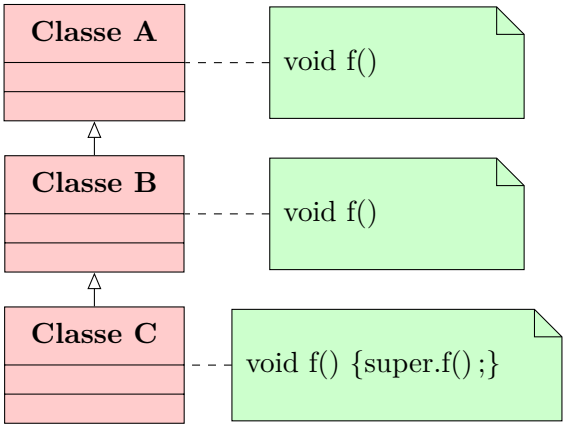

FIGURE 10 – L'appel super.f() fait implicitement référence à la super-classe directe, donc ici à f() de B

2.9 Constructeurs

Dans cette dernière section, nous revenons sur la définition des constructeurs dans le cas de l'héritage. Nous rappelons la règle selon laquelle chaque classe s'occupe des attributs qu'elle définit. Pour les attributs

hérités, elle délègue à ses super-classes. Appliquée à un constructeur de la classe `Livre`, cette règle implique qu'il va :

— déléguer à ses super-classes l'initialisation des attributs hérités.

— initialiser editeur et an.

Mais dans un constructeur, on ne peut appeler qu'un constructeur de la super-classe directe, donc il sera nécessaire d'avoir dans la classe intermédiaire ProduitTReduit (superclasse de `Livre` et sous-classe de `Produit`) un constructeur qui servira de "passeur" de paramètres. La figure 11 schématise cette structuration qui sera très fréquente. Ici nous prenons le cas de constructeurs qui initialisent tous les attributs.

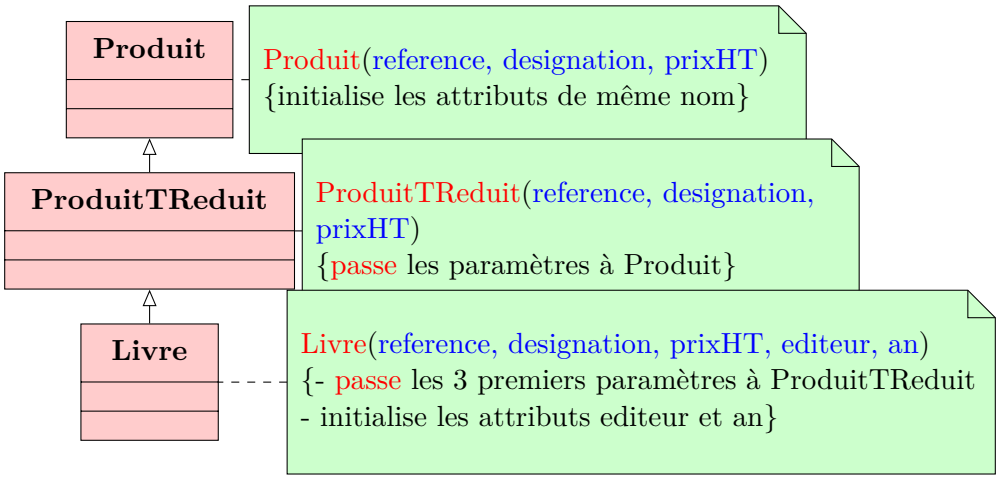

FIGURE 11 – Schéma des appels de constructeurs

Le code des constructeurs présenté ci-après fait un nouvel usage du mot-clef `super`, qui est cette fois suivi uniquement d'une liste de valeurs (paramètres réels transmis à un constructeur de la super-classe dont les paramètres formels correspondent).

```
public Livre(String reference, String designation, float prixHT, String editeur, int an)
{
        super(reference, designation, prixHT);
        this.editeur = editeur;
        this.annee = an;
}

public ProduitTReduit(String reference, String designation, float prixHT)
{
        super(reference, designation, prixHT);
}
```

Créons un livre avec l'instruction qui suit :

```
Livre L = new Livre(r, d, p, e, a);
```

Son initialisation va suivre les étapes suivantes :

Appel de Livre(r,d,p,e,a)
↪ Appel de ProduitTReduit(r,d,p)
↪ Appel de Produit(r,d,p)
Init de reference, designation, prixHT

```
    ↪ Init de editeur, annee
```

Deux principes sont à connaître :

— L'appel super(...) est toujours la première instruction du constructeur.

— L'exécution d'un constructeur commence toujours par un appel à un constructeur de la super-classe directe, sauf pour Object qui n'a pas de super-classe. Ce peut être implicite : au besoin le compilateur insère l'instruction super() ; qui appelle le constructeur sans paramètre de la super-classe directe. Mais que cette instruction apparaisse ou pas ne change rien à l'exécution.

Lorsque vous programmez avec l'IDE eclipse, celui-ci peut, suivant le paramétrage choisi, insérer automatiquement `super();` en début de chaque constructeur dans lequel vous n'avez pas vous-même placé d'appel à super. On peut le voir dans le code suivant :

```
public Produit(String reference, String designation, float prixHT)
{
        super();

        this.reference = reference;
        this.designation = designation;
        this.prixHT = prixHT;
}
```

A quel constructeur fait référence cette instruction `super();` ? Ici il s'agit de celui de la super-classe directe de `Produit` c'est-à-dire `Object`. Cette dernière n'en définit pas, mais comme elle ne définit par ailleurs aucun constructeur, il en existe un, sans paramètres et vide, qui est généré automatiquement. Rappelons une recommandation du cours précédent qui consiste à toujours prévoir (sauf rare exception) un constructeur par défaut sans paramètre dans chaque classe, qui construit une instance sous une forme légale "par défaut", sur laquelle un appel de méthode ne devrait pas mettre le programme en échec.

2.10 Conclusion

Dans cette partie, nous avons introduit différents principes généraux qui seront approfondis prochainement :

— La hiérarchie d'héritage permet une meilleure compréhension du code par sa proximité avec une classification des objets d'un domaine (qui peut être plus ou moins technique).

— L'héritage (la spécialisation/généralisation en UML) a pour principe de définir des concepts par genre et différence, ce qui permet une économie dans les descriptions structurelles et comportementales.

— L'élimination des redondances et des tests de type, ainsi que la répartition des responsabilités entre les classes des différents niveaux d'abstraction, conduisent à une meilleure cohérence du code.

— Ces qualités facilitent à terme l'évolution et la correction des programmes.

— L'héritage est également une caractéristique unique des langages à objets qui facilite l'extension des programmes et permet de ne pas tout penser à l'avance (avec des limites toutefois).

Du point de vue technique, nous avons présenté les principaux éléments de Java qui supportent la définition de l'héritage :

— La relation `extends` qui traduit qu'une classe hérite (spécialise, étend) une autre classe et qui est régie par l'héritage simple (une classe ne peut pas étendre de manière directe deux autres classes).

— La construction `super` dans une méthode ou un constructeur (2 syntaxes) qui permet d'appeler un comportement défini plus haut dans la hiérarchie.

— Le mot-clef `abstract` qui permet d'indiquer qu'une méthode a un comportement inconnu, ou qu'une classe n'est pas instanciable directement.
— Le fonctionnement de la factorisation et de l'héritage des propriétés (attributs d'instance).
— Le principe de délégation et de redéfinition des comportements.
— La séparation entre le typage statique des variables et le typage dynamique des objets et son utilisation dans le mécanisme de la liaison dynamique.

3 Aspects avancés de l'héritage

Dans cette partie, nous étudions différents points de vue avancés sur l'héritage :

1. Eléments sur la structure (modèles d'héritage et d'instanciation)
2. Visibilité (contrôle d'accès statique aux propriétés et classes internes)
3. Redéfinition versus surcharge statique (comprendre la différence entre les deux et la manière dont les deux mécanismes se combinent)
4. Coercition et tests de type (définition et usage)
5. Aperçu sur l'extensibilité
6. Le combat "spécialisation naturelle" contre "substituabilité"

3.1 Eléments sur la structure

Dans cette partie nous précisons les deux modèles proposés par UML et Java pour l'héritage et l'instanciation.

Il existe deux grandes catégories de modèles d'héritage :

— L'héritage simple, pour lequel un concept (classe) n'a qu'un super-concept (super-classe) direct. Cela a pour conséquence que la hiérarchie est un arbre. C'est le modèle choisi par Java pour les classes.
— L'héritage multiple, pour lequel un concept (classe ou interface) peut avoir plusieurs super-concepts (super-classe ou super-interface) directs. Dans ce cas, la hiérarchie est un ordre partiel. C'est le modèle choisi par UML, C++ pour les classes et par Java pour les interfaces. Différents problèmes de conflits peuvent apparaître, principalement lorsqu'il s'agit de classes. L'héritage multiple est illustré figure 12. Un conflit peut apparaître par exemple dans la classe `Carré` si les classes `Rectangle` et `Losange` définissent chacune une méthode de calcul du périmètre.

UML et Java ont également deux philosophies distinctes en ce qui concerne les modèles d'instanciation :

— Le modèle choisi par Java est celui de la mono-instanciation, dans lequel une instance est rattachée directement à une seule classe. Dans une expression de création d'objet, derrière l'opérateur `new` on ne va trouver qu'une classe. `Carre c= new Carre(...);`

— Le modèle d'UML est celui de la multi-instanciation, dans lequel une instance est rattachée à plusieurs classes. La figure 13 présente un exemple où un objet `c1` a deux classes de rattachement, Rectangle et Losange.

Ces deux modèles ont des conséquences sur (1) la modélisation et (2) la traduction UML vers Java lors du développement. Par exemple, en UML, on pourra sans difficulté introduire deux classes `VehiculeTerrestre`

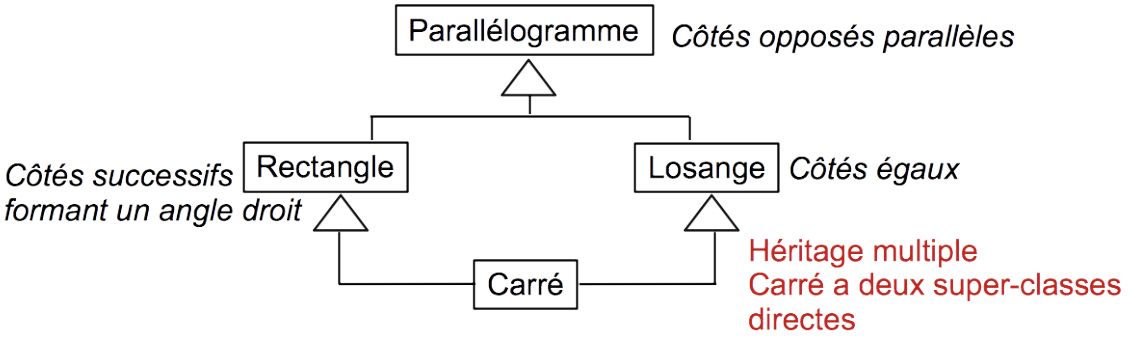

FIGURE 12 – La classe Carré a deux super-classes directes, Rectangle et Losange

et `VehiculeAquatique`, et l'instance représentant une sQuba [3] se rattachera à ces deux classes sans qu'il soit nécessaire d'introduire la notion de `VehiculeAmphibie`. En Java on devra ajouter une classe ou une interface "relai" pour représenter la combinaison des deux concepts (en Java un concept sera une classe ou une interface) : imaginons que l'on dispose d'un concept `VehiculeTerrestre` et d'un concept `VehiculeAquatique`, on devra introduire une classe `VehiculeAmphibie`, spécialisant les deux autres pour instancier la sQuba. Une mise en œuvre satisfaisante en Java sera d'autant plus compliquée à trouver que l'on ne dispose pas d'héritage multiple entre les classes.

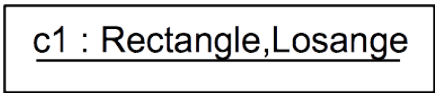

FIGURE 13 – `c1` a deux classes : Rectangle et Losange

3.2 Visibilité (contrôle d'accès statique)

La visibilité est une manière de contrôler l'appel d'une méthode ou l'accès à un attribut ou à une classe interne qui est en quelque sorte inverse de celle que nous avons étudiée et qui concerne les liaisons statique et dynamique. Dans un appel `o.m()` effectué dans une méthode d'une classe `C` :

- Les liaisons statique et dynamique vérifient qu'une méthode `m()` existe dans la classe `Cm` de l'objet `o` et se préoccupent de choisir la plus adaptée.
- La visibilité (aussi appelée contrôle d'accès statique) consiste à vérifier que la classe `C` a le droit d'appeler `m()` sur `o`.

En Java et UML, quatre directives de visibilité peuvent être placées sur les membres d'une classe `Cm` (incluant les attributs, méthodes et classes internes) :

- privé : se note `private` en Java et par le symbole `-` en UML. Le membre privé est accessible seulement par les méthodes de `Cm`.
- paquetage : pour l'indiquer on ne met pas de directive devant le membre en Java, et on utilise le symbole ∼ en UML. L'accès de niveau paquetage rend le membre accessible seulement par les méthodes des classes qui sont dans le même package que `Cm`.

3. https ://www.largus.fr/actualite-automobile/top-15-des-voitures-amphibies-646142-minidiapo.html

— protégé : se note `protected` en Java et par le symbole # en UML. En Java, le membre protégé est accessible seulement par les méthodes des classes qui sont dans le même package que `Cm` et les méthodes des classes dérivées de `Cm` (directement ou indirectement). La règle de Java est assez complexe, nous l'illustrerons par la suite. Elle stipule que :
Hors du package, les propriétés protégées de `Cm` sont accessibles par une sous-classe Cs de `Cm` soit sur ses propres instances, créées avec new Cs(...), soit sur une instance d'une sous-classe Css de Cs, créée avec new Css(...), et jamais en remontant.
UML n'est pas aussi précis sur la visibilité `protected` et indique seulement que le membre protégé est accessible par les spécialisations de `Cm` (UML version 2.5).
— public : se note `public` en Java et par le symbole + en UML. Le membre public est accessible sans restriction à condition que la portée et les imports le permette.

RÈGLE 6 *Recommandation. De manière générale, les attributs sont privés (sauf cas particuliers, comme certaines constantes). Cela permet en particulier de contrôler les modifications et les affectations de valeurs de manière à ce que les attributs aient toujours une valeur conforme à leur sémantique. Par exemple, une note pour un étudiant sera de type* `double` *(contrainte assurée par le typage de l'attribut) et comprise dans l'intervalle* `[0,1]` *(contrainte assurée par des accesseurs appropriés et par leur appel systématique pour les modifications).*

Une question que l'on peut se poser est de savoir si on peut redéfinir une méthode privée (comme en C++ par exemple). En réalité, en Java, on ne peut pas. La figure 14 illustre une situation que l'on doit analyser comme le fait que la classe `B` définit une nouvelle méthode.

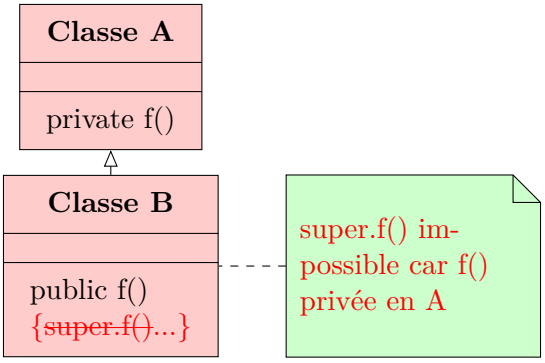

FIGURE 14 – `B` définit une nouvelle méthode

Revenons sur la directive `protected` de Java et la règle indiquant que *Hors du package, les propriétés protégées de* `Cm` *sont accessibles par une sous-classe Cs de* `Cm` *soit sur ses propres instances, créées avec new Cs(...), soit sur une instance d'une sous-classe Css de Cs, créée avec new Css(...), et jamais en remontant.*

La figure 15 illustre cette règle de manière détaillée.

Dans cette situation :

— Dans le paquetage `arbrebinaire`, une classe `ArbreBinaire` (AB en notation raccourcie) introduit un attribut protégé `estVide`.
— Dans le paquetage `arbrebinaireborné`, une classe `ArbreBinaireProfondeurBornee` (ABPB en notation raccourcie) est sous-classe de AB et a trois méthodes de copie.

— Dans le paquetage `abpc`, une classe `ArbreBinaireProfBorneeColore` (ABPBC en notation raccourcie) est sous-classe de ABPB.

Analysons les accès dans les méthodes de copie de la classe ABPB :

— Dans `copie(AB a1)`, l'accès au membre protégé `estVide` sur `a1` n'est pas autorisé car il est effectué sur un objet d'une super-classe de ABPB (en remontant).

— `copie(ABPB a2)`, l'accès au membre protégé `estVide` sur `a2` est autorisé car il est effectué sur un objet de ABPB.

— `copie(ABPBC a3)`, l'accès au membre protégé `estVide` sur `a3` est autorisé car il est effectué sur un objet d'une sous-classe de ABPB.

Le code qui suit présente cette même situation. Les commentaires évoquent également la notion de redéfinition que nous verrons plus loin dans ce document.

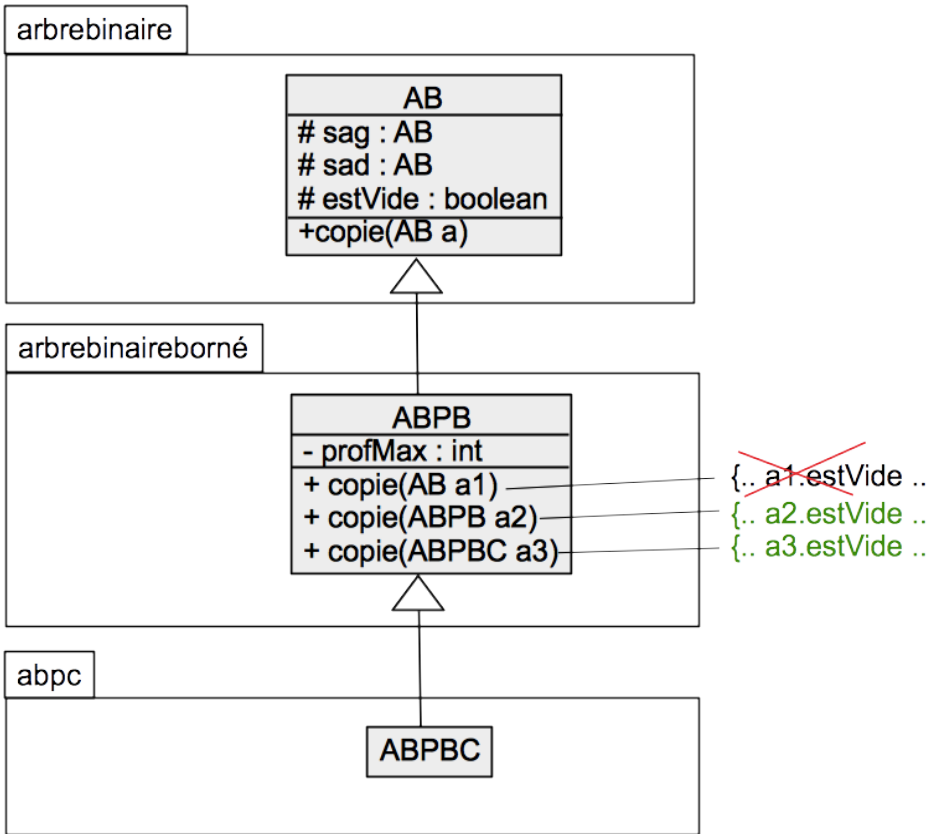

FIGURE 15 – Illustration de la règle régissant la directive `protected`

```
package visibilite.arbrebinaire;
public class ArbreBinaire {
        protected ArbreBinaire sag,  sad;
        protected boolean estVide; // dans ce cas, sag et sad inutiles

        public void copie(ArbreBinaire a)
        {
                this.estVide = a.estVide;
                if (! a.estVide)
                // code qui copie a dans this sans regarder la profondeur
                // ...
```

```
                {

                } ...} ...}
```

```
package visibilite.arbrebinaireborne;
import visibilite.abpc.ArbreBinaireProfBorneeColore;
import visibilite.arbrebinaire.ArbreBinaire;
public class ArbreBinaireProfondeurBornee extends ArbreBinaire{
        // la structure est deja definie, on va seulement verifier la profondeur de l'arbre
        private int profondeurMax;

        // on ne peut pas acceder au champ protected sur un arbre binaire qui est la superclasse et cette methode est bien une redefinition de copie de la superclasse
        public void copie(ArbreBinaire a){
                this.estVide = a.estVide; // PB
                if (! a.estVide){ // PB
                // code qui copie a dans this jusqu'a profondeurMax
                ....}
        }

        // on  peut acceder au champ protected sur un arbre de cette classe mais cette methode n'est pas une redefinition de copie de la superclasse
        public void copie(ArbreBinaireProfondeurBornee a){
                this.estVide = a.estVide;
                if (! a.estVide){
                // code qui copie a dans this jusqu'a profondeurMax
                ....}
        }


package visibilite.abpc;
import visibilite.arbrebinaire.ArbreBinaire;
import visibilite.arbrebinaireborne.ArbreBinaireProfondeurBornee;

public class ArbreBinaireProfBorneeColore extends ArbreBinaireProfondeurBornee {


}

package visibilite.arbrebinaireborne;
import visibilite.abpc.ArbreBinaireProfBorneeColore;
import visibilite.arbrebinaire.ArbreBinaire;

public class ArbreBinaireProfondeurBornee extends ArbreBinaire{
// suite
        // on  peut acceder au champ protected sur un arbre d'une sous-classe de cette classe
        // mais cette methode n'est pas une redefinition de copie de la superclasse
        public void copie(ArbreBinaireProfBorneeColore a)
        {
                this.estVide = a.estVide;
                if (! a.estVide){
                // code qui copie a dans this jusqu'a profondeurMax
                // ...
```

```
                }
        }
}
```

3.3 Redéfinition versus surcharge statique en Java

Dans cette partie, nous expliquons plus en détails la différence entre redéfinition et surcharge statique en Java et nous l'illustrons sur un exemple. Nous commençons par donner les conditions dans lesquelles une méthode en redéfinit une autre. Notons que cette section ne concerne que Java. Les règles de redéfinition de méthodes sont différentes en UML et sont précisées dans la section 3.6.

RÈGLE 7 *Redéfinition de méthode*
En Java, une méthode m_r redéfinit une méthode m_i (la liaison dynamique sera appliquée) lorsque :

- *m_r et m_i portent le même nom.*
- *m_r a la même liste de types de paramètres que m_i (invariance de la liste des types de paramètres).*
- *le type de retour de m_r est le même ou est une spécialisation de celui de m_i (covariance du type de retour).*

De plus lors d'une redéfinition :

- On peut changer la visibilité pour l'augmenter (de `protected` à `public` par exemple).
- Des règles seront ajoutées pour les déclarations d'exceptions (voir dans un prochain cours).

Dans les autres cas, on parle de surcharge statique. La figure 16 illustre un cas simple de surcharge statique. La classe `A` définit une méthode de signature `f(int)` et sa sous-classe `B` définit une méthode de signature `f(String)`. `f(String)` ne redéfinit pas `f(int)` car le type du paramètre est différent. Une instance de `B` possède 2 méthodes `f`. Le code suivant montre le résultat lors d'appels des méthodes `f`.

```
1 B b = new B();
2 b.f(5);          // f de A
3 B.f("5");        // f de B
```

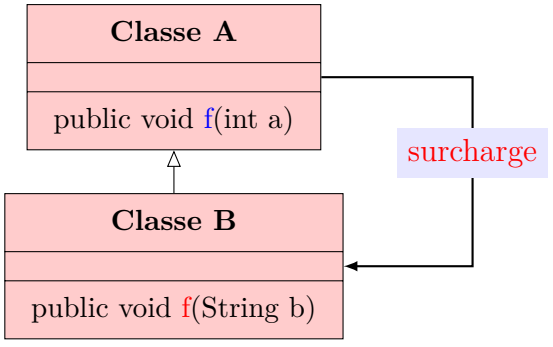

FIGURE 16 – Surcharge statique

Cette première situation était assez simple à comprendre. Nous présentons ci-dessous un exemple plus difficile à analyser. Nous introduisons tout d'abord une classe `Point`, qui déclare une redéfinition de la méthode `boolean equals(Object p)`.

```java
public class Point{
     private int x,y;
     public Point() {this.x = 0; this.y = 0;}
     public Point(int x, int y) {this.x = x; this.y = y;}

     public int getX() {return x;}
     public void setX(int x) {this.x = x;}
     public int getY() {return y;}
     public void setY(int y) {this.y = y;}

     public boolean equals(Object p)
     // redefinition de celle d'Object, avec coercition
     {return this.x==((Point)p).x && this.y==((Point)p).y;}
}
```

Puis nous introduisons une classe `Cercle` et une sous-classe `CercleColore` qui chacune déclare une méthode `equals`.

```java
public class Cercle {
    private Point centre; private double rayon;
    .....
    public boolean equals(Cercle c)
    {return this.getCentre().equals(c.getCentre()) &&
                this.getRayon() == c.getRayon();}
}

public class CercleColore extends Cercle {
    private String couleur;
    .........
    public boolean equals(CercleColore c) {
        return super.equals(c) && this.getCouleur().equals(c.getCouleur());}
}
```

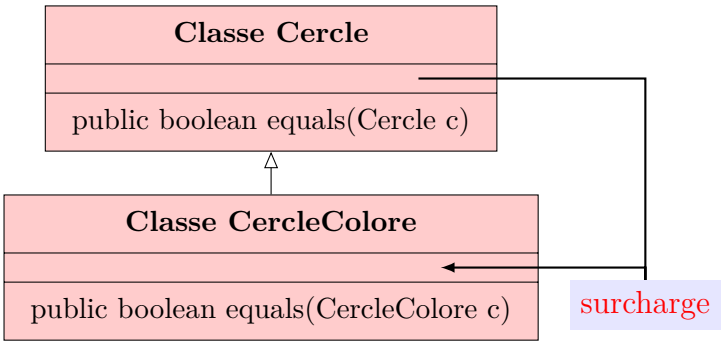

FIGURE 17 – Surcharge statique entre les deux méthodes `equals`

Pour comprendre ce qui va se dérouler, il faut tout d'abord considérer les éléments suivants, illustrés par la figure 17 :

- Le type de paramètre de la méthode de CercleColore spécialise celui de la méthode de Cercle, il est donc bien *différent*.
- Comme le type de paramètre diffère, il s'agit de surcharge statique.
- On a donc deux méthodes bien distinctes `boolean equals(Cercle c)` et `boolean equals(CercleColore c)`.

— Le choix de la signature de méthode à appeler est fait par le compilateur ; puis pour une signature choisie il peut y avoir liaison dynamique.

Observons les trois instructions suivantes, qui créent respectivement un cercle `c1` de centre (1,2), de rayon 12 et de couleur bleue et un cercle `c2` de centre (1,2), de rayon 12 et de couleur verte. L'affichage résultant est `true` (les deux cercles seraient considérés comme égaux pour ce programme alors qu'ils sont de couleur différente) :

```
Cercle c1 = new CercleColore(new Point(1,2), 12, "bleu");
Cercle c2 = new CercleColore(new Point(1,2), 12, "vert");
System.out.println("c1 egal c2 ?"+c1.equals(c2)); //--> TRUE
```

Voici comment cela s'explique :

— `c1` a pour type statique `Cercle` et pour type dynamique `CercleColore`.
— Pour le compilateur : une signature pour `equals` et admettant un `Cercle` (type statique de c2) en paramètre réel est cherchée dans `Cercle` : c'est `equals(Cercle c)`.
— Pour l'interprète : il regarde si `equals(Cercle c)` est redéfinie dans le type dynamique de `c1`. Ce n'est pas le cas, la méthode `equals(CercleColore c)` n'étant pas une redéfinition.
— `equals(Cercle c)` est exécutée et ne compare que le centre et le rayon, sur ce plan `c1` et `c2` sont identiques.

Regardons à présent les trois instructions suivantes pour lesquelles il y a une variation des types, et dont l'affichage résultant est `false` :

```
CercleColore c3 = new CercleColore(new Point(1,2), 12, "bleu");
CercleColore c4 = new CercleColore(new Point(1,2), 12, "vert");
System.out.println("c3 egal 4 ?"+c3.equals(c4));// --> FALSE
```

Cette fois il faut comprendre que :

— `c3` a pour type statique et pour type dynamique `CercleColore`.
— Pour le compilateur : une signature pour `equals` et admettant un `CercleColore` en paramètre réel est cherchée dans `CercleColore` : c'est `equals(CercleColore c)`.
— Pour l'interprète : il cherche `equals(CercleColore c)` à partir de `CercleColore` (et la trouve là).
— `equals(CercleColore c)` est exécutée et compare centre, rayon, et couleur ; sur la couleur, `c1` et `c2` sont différents.

Enfin, observons une troisième variation, où s'affiche `true` :

```
Cercle c1 = new CercleColore(new Point(1,2), 12, "bleu");
CercleColore c4 = new CercleColore(new Point(1,2), 12, "vert");
System.out.println("c4 egal c1 ?"+c4.equals(c1)); // --> TRUE
```

L'explication est maintenant :

— `c4` a pour type statique et dynamique `CercleColore`.
— `c1` a pour type statique `Cercle` et pour type dynamique `CercleColore`.
— Pour le compilateur : une signature pour `equals` et admettant un `Cercle` en paramètre réel est cherchée dans `CercleColore` : c'est `equals(Cercle c)` (héritée), car `equals(CercleColore c)` ne peut accepter un cercle.
— Pour l'interprète : Il regarde si `equals(Cercle c)` est redéfini dans le type dynamique de `c4`. Ce n'est pas le cas, la méthode `equals(CercleColore c)` n'étant pas une redéfinition.
— `equals(Cercle c)` est exécutée et ne compare que centre et rayon, sur ce plan `c1` et `c4` sont identiques.

3.4 Coercition et tests de type en Java

Nous revenons à présent sur un point du cours précédent dans lequel nous avions indiqué qu'il était préférable d'éviter de programmer en utilisant des tests de type, plus ou moins dissimulés, sur les objets. En réalité il y a des contextes où ces tests de type, souvent accompagnés de coercition (*typecast*) s'avèrent nécessaires. C'est notamment le cas où l'on redéfinit une méthode comme `equals` qui a un type de paramètre que l'on ne peut spécialiser dans la sous-classe (pour respecter la règle de redéfinition).

Cherchons à redéfinir correctement la méthode `boolean equals(Cercle c)` dans `CercleColore`. Une première approche consiste à écrire :

```
1 public boolean equals(Cercle c) {
2    return super.equals(c) &&
3             this.getCouleur().equals(c.getCouleur());
4 }
```

Ce code provoque une erreur de compilation car le type statique de `c` est `Cercle` mais la méthode `getCouleur()` n'existe pas dans `Cercle`. Une solution avec de la coercition (ou typecast) et un test de type sera bienvenue et vous est montrée dans le code suivant :

```
1 public boolean equals(Cercle c) {
2    if (c instanceof CercleColore)
3       return super.equals(c) &&
4          this.getCouleur().equals(((CercleColore)c).getCouleur());
5    else return false;
6 }
```

L'expression `(CercleColore)c)` est une *coercition* : on indique au compilateur de considérer `c` comme s'il avait pour type statique `CercleColore`.

La condition `c instanceof CercleColore` est un *test de type* : à l'exécution, on vérifiera le type dynamique de `c` et l'expression renverra true si c'est une instance de `CercleColore` ou de l'une de ses sous-classes.

L'exemple donné (redéfinition d'une "biméthode") est un des cas où l'on s'autorise la coercition et les tests de type mais retenons malgré tout la règle suivante pour la majorité des autres cas :

RÈGLE 8 *Usage de la coercition et des tests de type*
En général, on utilise le moins possible la coercition et les tests de type. Ils révèlent souvent une mauvaise conception.

Vous pourriez vous demander si la coercition permet d'appeler la méthode *masquée*, par exemple, pour la figure 18. Mais ce n'est pas le cas, dans ce contexte, il n'y a pas de moyen d'appeler directement `f()` de `A` dans la classe `C`, même en écrivant une expression comme `((A)super).f();`

Pour terminer ce paragraphe, notez que la méthode `boolean equals(Object o)` existe dans la classe `Object` qui est la superclasse implicite de toutes les classes. De ce fait, une bonne solution pour l'écrire, que ce soit dans `Cercle` ou dans `CercleColoré` consiste donc à garder cette signature. Nous vous laissons effectuer à titre d'exercice la réécriture idoine.

3.5 Aperçu sur l'extensibilité

Dans cette section, nous abordons la question de l'extensibilité des programmes qui est une caractéristique importante apportée par les mécanismes des approches à objets et notamment l'héritage.

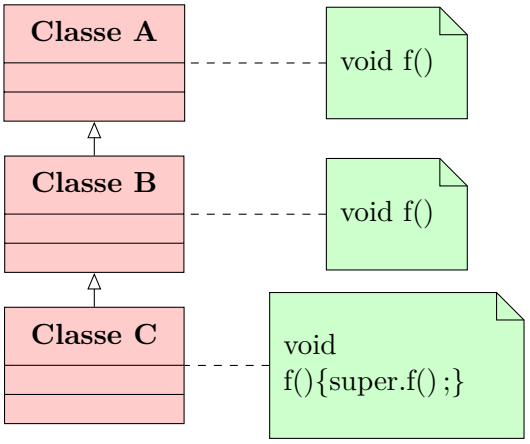

FIGURE 18 – On ne peut pas appeler la méthode masquée même avec une coercition

Nous l'illustrons en revenant sur l'exemple des produits du cours précédent avec cet extrait de code qui calcule le prix des produits stockés dans un caddie représenté comme un tableau.

```
1  Produit [] monCaddie = new Produit [50];
2  // remplissage du caddie
3  ....
4  monCaddie[i] = new Livre (...);
5  ....
6  // passage a la caisse
7  float prixTotal = 0;
8  for(int i = 0; i < nb ; i ++)
9  {
10         prixTotal += monCaddie[i].lePrixTCC();
11 }  // MonCaddie : reference de type produit
```

Dans ce code, le compilateur vérifie que Produit possède une méthode lePrixTTC(). L'interpréteur choisit ensuite la bonne version de la méthode lePrixTTC(). Ce code est générique au sens où `monCaddie[i].lePrixTCC();` a un sens pour toutes les instances de n'importe quelle sous-classe de produit (grâce au fait que l'on a déclaré la méthode abstraite dans la classe `Produit`). Il est donc important de prévoir toutes les méthodes applicables à tous les produits au niveau de la racine (classe Produit) comme l'évoque la figure 19.

Si nous ajoutons par exemple à notre hiérarchie d'héritage la notion d'aliment frais avec les caractéristiques :

- température de conservation
- prix réduit si la température a été dépassée
- réduction de la date limite de validité si la température a été dépassée

Les expressions précédemment écrites fonctionnent toujours !

3.6 Le combat entre la spécialisation naturelle et la substituabilité

Deux principes sont sous-jacents à la modélisation et à la programmation par objets dont nous allons voir qu'ils sont parfois antagonistes.

Règle 9 *Spécialisation naturelle*
Un concept S spécialise un concept T lorsque l'ensemble des instances de S est inclus dans l'ensemble des instances de T.

La spécialisation naturelle est le point de vue notamment de la modélisation en UML. Par exemple, les aliments frais sont des aliments ; les carrés sont des rectangles. En UML, lorsque l'on redéfinit une méthode, on pourra spécialiser les types des paramètres (au contraire de Java) et le type de retour (comme en Java).

Règle 10 *Principe de substitution de Liskov (sous-typage correct)*
Lorsqu'un type S est sous-type d'un type T, tout objet de type S peut remplacer une expression de type T sans que le programme n'ait de comportement imprévu ou ne perde de propriétés souhaitées.

Le principe de substitution de Liskov est le point de vue des langages à typage statique, il sert à anticiper au maximum les erreurs dans les programmes. Par exemple, même si cela peut sembler curieux, les aliments frais ne seraient pas des aliments dans ce point de vue car on ne peut pas les ranger dans tous les lieux où l'on range des aliments (il faut un lieu réfrigéré pour les aliments frais).

Le principe de substitution de Liskov, posé par Jeannette Wing et Barbara Liskov [4], est assez contraignant car il impose :

— des restrictions sur les signatures lors des redéfinitions,
— des conditions comportementales (apparentes notamment sur les contrats que nous décrirons dans un prochain cours).

C'est cependant lui qui permet d'appliquer la règle (avec un risque réduit d'erreur à l'exécution) :

— une variable de type A peut contenir la référence d'une instance de A ou la référence d'une instance de l'une des sous-classes de A.

Ce principe de substitution, utile notamment pour la vérification des programmes, est parfois en contradiction avec la vision de la spécialisation/généralisation comme la pense UML, dont l'objectif est de modéliser la sémantique naturelle d'un domaine et non pas de vérifier des expressions. Lors de la modélisation de la sémantique naturelle, on ne se préoccupe pas de prédire tout ce qui va se passer pour un sous-groupe particulier d'objets (par exemple les aliments frais) d'un groupe plus général (les aliments). Lorsque l'on vérifie des expressions, on veut au contraire être certains de ce qui va se produire. Si on exprime dans une première étape d'un programme que les aliments se rangent dans des placards (quelconques, y compris non réfrigérés), il faudrait que ce soit vrai pour tous les sous-groupes décrits plus tard lors d'extensions du programme.

Nous vous présentons ci-après sur des exemples les deux volets de la substituabilité, sur les signatures puis sur le comportement.

3.7 Substituabilité et restriction des signatures lors des redéfinitions

Par *redéfinition*, dans cette section, on entend le fait de déclarer une méthode d'un certain nom dans une classe et de déclarer une méthode du même nom dans la sous-classe.

La substituabilité revient à demander moins et donner plus au site d'appel d'une méthode lors de l'appel d'une redéfinition. Par site d'appel, on entend une expression du type `o.m();`. La figure 20 illustre les deux principales règles (contravariance des types des paramètres, covariance du type de retour).

— contravariance des types de paramètres

4. Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems, 16 :1811-1841, 1994.

  - les types des paramètres varient en sens inverse de la redéfinition
- covariance du type de retour
  - le type de retour varie dans le même sens que la redéfinition
- affaiblissement des pré-conditions
- renforcement des post-conditions

Imaginons une redéfinition de méthode avec covariance du paramètre, contraire au principe de substitution de Liskov pour comprendre quel problème se trouverait posé. Dans la figure 21, la méthode `stockerDans` est déclarée dans `Produit` avec un paramètre de type `Lieu` et redéfinie (au sens d'UML) avec un paramètre de type `LieuRefrigere` dans `AlimentFrais`.

Le code suivant compilerait si cette redéfinition était autorisée en Java, mais, à l'exécution, il poserait problème par exemple si stocker dans un lieu réfrigéré fait accès à la température, inconnue pour les lieux ordinaires et pour les placards.

```
Produit prod = new AlimentFrais(...);
Lieu monPlacard = new Placard(...);
prod.stockerDans(monPlacard);
```

La contravariance sur les types des paramètres (demandée par la substituabilité) n'est en réalité pas en accord avec la spécialisation naturelle (que préconise UML). L'invariance choisie par Java est donc un choix raisonnable.

Observons à présent une redéfinition légale pour le principe de substitution de Liskov (covariance du type de retour) qui est présentée dans la figure 22. La méthode `cloner` est déclarée dans `Produit` avec le type de retour `Produit` et dans `AlimentFrais` avec le type de retour `AlimentFrais`. La covariance du type de retour, conforme au principe de substitution de Liskov est de plus en accord avec la spécialisation naturelle. La covariance choisie par Java est en accord avec les deux.

3.8 Substituabilité comportementale

Dans cette partie, nous présentons un exemple de problème de substitution comportementale, qui est un classique de la littérature et est inspiré de J. Wing et B. Liskov, avec :

- S : Square
- T : Rectangle

Notons qu'un carré peut être considéré comme un rectangle du point de vue de la spécialisation naturelle.

Nous introduisons tout d'abord une classe pour représenter les rectangles.

```
public class Rectangle {
    private double width, height;

    public Rectangle() {}
    public Rectangle(double width, double height) {
        this.width = width; this.height = height;
        assert(this.getWidth()==width&&this.getHeight()==height);
    }

    public double getWidth()
        { return width; }
    public void setWidth(double width)
        { this.width = width; assert(this.getWidth() == width); }
    public double getHeight()
        { return height; }
```

```
16      public void setHeight(double height) {
17          this.height=height;
18          assert(this.getHeight() == height);
19      }
20      public String toString()
21          { return "Rectangle:"+this.getWidth()+" "+this.getHeight();}
22  }
```

Puis une méthode qui devrait fonctionner pour les rectangles (sans que l'assertion ne soit activée par une erreur d'exécution ; nous verrons les assertions en détails au prochain cours, mais comprenez qu'il s'agit de vérifier en un point du code la propriété placée derrière le mot-clef `assert`).

```
1  public static void mystery(Rectangle r) {
2          r.setWidth(5.0);
3          System.out.println(r);
4          r.setHeight(4.0);
5          System.out.println(r);
6          assert(r.getWidth() * r.getHeight() == 20.0);
7  }
```

Enfin, nous introduisons une classe pour représenter les carrés, qui hérite de la classe représentant les rectangles. Elle stocke la valeur du côté dans les deux attributs hérités `width` et `height` (cette implémentation est faite pour les besoins de l'exemple, elle n'est aucunement recommandée, mais elle est typique de ce qui peut parfois se produire lors d'une opération de spécialisation d'une classe par une autre).

```
1  public class Square extends Rectangle {
2          public Square() {}
3          public Square(double width) {
4                  super(width, width);
5                  assert(getWidth()==getHeight());
6          }
7          public void setWidth(double width) {
8                  super.setWidth(width);
9                  super.setHeight(width);
10                 assert(getWidth()==getHeight());
11                 assert(this.getWidth() == width);
12         }
13         public void setHeight(double height) {
14                 this.setWidth(height);
15         }
16         public String toString() {
17                 return super.toString()+"Square:"+this.getWidth();
18         }
19 }
```

Notez un cas d'appel `super.setHeight()` dans une méthode nommée `setWidth()` (exception à la règle vue au précédent cours, mais nécessaire ici).

Le code suivant va s'interrompre en exécution suite à une erreur :

```
1  public static void main(String[] args) {
2                  Rectangle r = new Rectangle(8.0, 7.0);
3                  System.out.println(r);
4                  mystery(r); // se passe bien...
5
6                   r = new Square(3.0);
7                  System.out.println(r);
```

```
8                   mystery(r);
9                   // la derniere instruction de mystery,
10                  // r.setHeight() attribue 4 a la hauteur
11                  // ET a la largeur donc
12                  // r.getWidth() * r.getHeight() == 16.0
13                  // Declenchement de l'assertionError
14  }
```

Il faut le comprendre de cette manière :

— Même si un carré peut remplacer un rectangle dans le programme Java sans erreur de compilation, et même si naturellement un carré est un rectangle, un carré ne se comporte pas tout à fait de la même manière qu'un rectangle.

— Le principe de substitution de Liskov n'est pas vérifié en ce qui concerne la partie comportementale.

— Le comportement du programme est sain pour un rectangle et est altéré pour le carré (déclenchement d'une erreur pour non respect de l'assertion).

3.9 Conclusion

Dans cette partie, nous avons passé en revue différents aspects avancés de l'héritage :

1. Eléments sur la structure (héritage simple pour les classes Java, multiple en UML ; mono-instanciation en Java, multi-instanciation en UML)
2. Visibilité (public < protected < package < private)
3. Redéfinition versus surcharge statique en Java (attention aux règles de redéfinition)
4. Coercition et tests de type ((T), instanceof)
5. Aperçu sur l'extensibilité (code générique utilisant des méthodes abstraites)
6. Spécialisation naturelle versus substituabilité
   — Des décisions à prendre lors de la modélisation et de la programmation.
   — La spécialisation naturelle est structurante pour la pensée, l'organisation et la compréhension du programme et ne demande pas de tout prévoir.
   — La substituabilité permet de limiter les erreurs à l'exécution, mais demande d'anticiper.

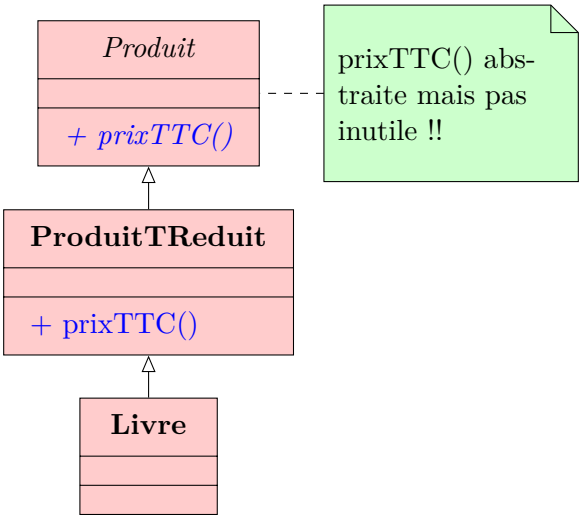

FIGURE 19 – Il faut prévoir des méthodes, même abstraites, dans les niveaux conceptuels les plus élevés où ils ont du sens

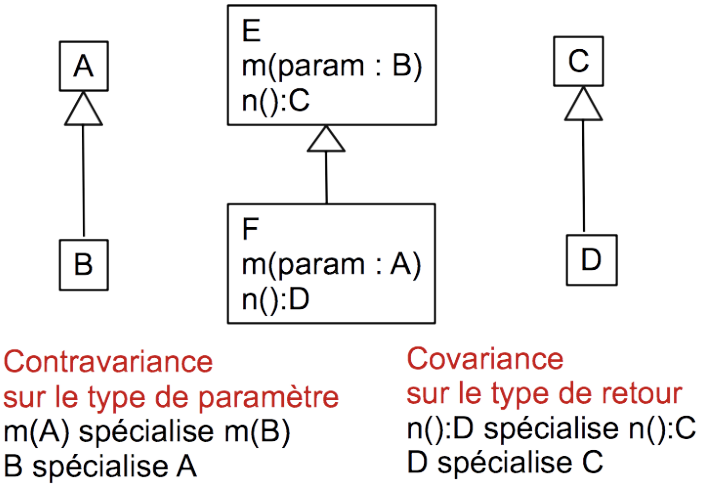

FIGURE 20 – Restrictions sur les signatures

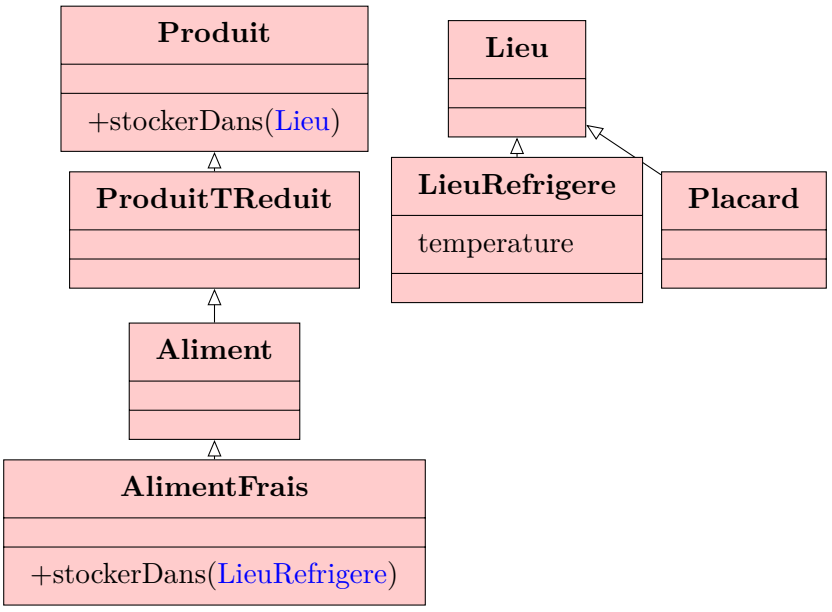

FIGURE 21 – Redéfinition (pour UML) de `stockerDans` avec covariance du type du paramètre

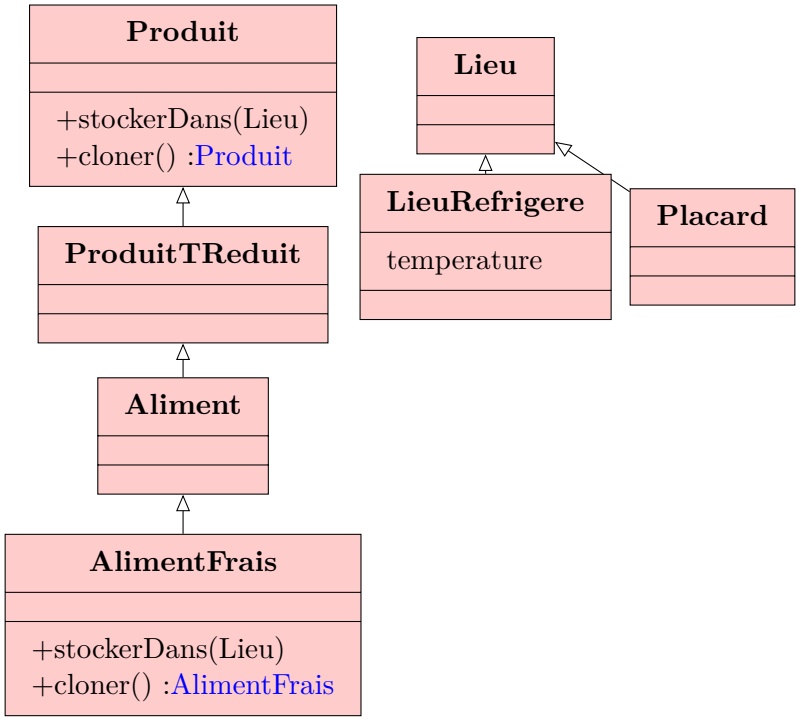

FIGURE 22 – Covariance du type de retour pour la méthode `cloner`