

Conception par Objets  
ULIN 407

Marianne Huchard, Clémentine Nebut

14 janvier 2009

Ces notes de cours sont en cours de réalisation. Elles sont donc susceptibles de contenir des erreurs ou des imprécisions, ou d'être incomplètes. Elles ne dispensent en aucun cas d'une présence en cours, TD ou TP.

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Pourquoi vous parler de conception par objets ? . . . . .	4
1.2	Modélisation des systèmes informatique . . . . .	4
1.2.1	Notion de modélisation . . . . .	4
1.2.2	UML, un langage de modélisation . . . . .	5
1.3	Concrétisation en Java . . . . .	6
<b>2</b>	<b>Le modèle d'utilisation en UML</b>	<b>7</b>
<b>3</b>	<b>Classes et paquetages : les éléments de base du modèle statique</b>	<b>10</b>
3.1	Les classes et instances en UML . . . . .	10
3.2	Les paquetages en UML . . . . .	13
3.3	Classes, instances et paquetages en Java . . . . .	13
3.3.1	Types de base en Java . . . . .	13
3.3.2	Écriture des classes . . . . .	14
3.3.3	Création des instances . . . . .	14
3.3.4	Accès aux attributs . . . . .	14
<b>4</b>	<b>Opérations et méthodes</b>	<b>15</b>
4.1	Classes, opérations et méthodes . . . . .	15
4.2	Opérations en UML . . . . .	15
4.3	Méthodes en Java . . . . .	18
4.3.1	Déclaration de méthodes . . . . .	18
4.3.2	Exécution d'un premier programme . . . . .	19
4.3.3	Les accesseurs . . . . .	19
4.3.4	Quelques instructions de base . . . . .	20
4.3.5	Structures de contrôle . . . . .	23
<b>5</b>	<b>Les associations UML, et leur implémentation en Java</b>	<b>26</b>
5.1	Les associations . . . . .	26
5.1.1	Associations et liens . . . . .	26
5.1.2	Associations et attributs . . . . .	27
5.1.3	Agrégation et composition . . . . .	27
5.1.4	Associations n-aires . . . . .	28
5.1.5	Associations qualifiées . . . . .	28
5.1.6	Les classes d'association . . . . .	28
5.2	Comment traduire les associations en Java ? . . . . .	30
5.2.1	Les tableaux . . . . .	30
5.2.2	Les collections Java . . . . .	31

5.2.3	Les Vecteurs . . . . .	31
5.2.4	Les tables de hachage . . . . .	32
5.2.5	Ce qu'on ne peut pas traduire directement . . . . .	32
<b>6</b>	<b>Les diagrammes dynamiques en UML</b>	<b>33</b>
6.1	Introduction . . . . .	33
6.2	Les diagrammes de séquence . . . . .	33
6.2.1	La ligne de vie . . . . .	34
6.2.2	Les messages . . . . .	34
6.2.3	Composition de fragments de diagrammes de séquence . . . . .	37
<b>7</b>	<b>Spécialisation/généralisation et héritage</b>	<b>42</b>
7.1	Généralisation - Spécialisation . . . . .	42
7.1.1	Classer les objets . . . . .	42
7.1.2	Discriminants et contraintes . . . . .	42
7.2	Hierarchie des classes et héritage dans Java . . . . .	45
7.3	Redéfinition de méthodes - Surcharge - Masquage . . . . .	47
7.4	Les constructeurs . . . . .	48
7.5	Protection/contrôle d'accès statique . . . . .	49
7.6	Classes et méthodes abstraites . . . . .	49
7.7	Le polymorphisme . . . . .	50
7.7.1	Coercition, Transformation de type ou Casting . . . . .	50
7.7.2	Le polymorphisme . . . . .	53
<b>8</b>	<b>Les interfaces de Java</b>	<b>56</b>
8.1	Définition et objectif . . . . .	56
8.2	Éléments syntaxiques . . . . .	56
8.2.1	Définition d'une interface . . . . .	56
8.2.2	Spécialisation d'une interface . . . . .	57
8.2.3	Lien classe/interface . . . . .	57
8.3	Description de comportements génériques . . . . .	58
8.4	Description plus abstraite des types . . . . .	58
8.5	Spécialisation multiple . . . . .	60
8.6	Quelques interfaces importantes de l'API Java . . . . .	61
8.6.1	Interfaces marqueurs . . . . .	61
8.6.2	Comparaison d'objets et tris . . . . .	62
8.6.3	Collections et itérateurs . . . . .	62

# Chapitre 1

## Introduction

### 1.1 Pourquoi vous parler de conception par objets ?

Les approches par objets sont un succès dans l'histoire de l'informatique (30 dernières années) :

- elles sont fondées sur quelques idées simples qui consistent à décrire un système avec des représentations informatiques proches des entités du problème et de sa solution : si on parle d'un système bancaire on décrira des objets *Comptes bancaires* dans le langage informatique ; cela facilite la communication entre les intervenants d'un projet ;
- elles ont des avantages reconnus en termes de :
  - facilité du codage initial,
  - stabilité du logiciel construit car les objets manipulés sont plus stables que les fonctionnalités attendues,
  - aisance à réutiliser les artefacts existants et ...
  - à maintenir le logiciel, le corriger, le faire évoluer ;
- elles ont connu un fort développement dans les langages de conception, de programmation, les bases de données, les interfaces graphiques, les systèmes d'exploitation, etc.

**Objectifs** Ce cours présente les concepts essentiels de l'approche objet en s'appuyant sur un langage de modélisation (UML) et un langage de programmation (Java). Le langage de programmation permettra de rendre plus concrets les concepts étudiés.

### 1.2 Modélisation des systèmes informatique

#### 1.2.1 Notion de modélisation

La modélisation est la première activité d'un informaticien face à un système à mettre en place.

Modéliser consiste à produire une représentation simplifiée du monde réel pour :

- accumuler et organiser des connaissances,
- décrire un problème,
- trouver et exprimer une solution,
- raisonner, calculer.

Il s'agit en particulier de résoudre le hiatus entre :

- d’un côté le monde réel, complexe, en constante évolution, décrit de manière informelle et souvent ambiguë par les experts d’un domaine
- de l’autre le monde informatique, où les langages sont codifiés de manière stricte et disposent d’une sémantique unique.

La modélisation est une tâche rendue difficile par différents aspects :

- les spécifications parfois imprécises, incomplètes, ou incohérentes,
- taille et complexité des systèmes importantes et croissantes,
- évolution des besoins des utilisateurs,
- évolution de l’environnement technique (matériel et logiciel),
- des équipes à gérer plus grandes, avec des spécialisations techniques, nécessaires du fait de la taille des logiciels à construire, mais le travail est plus délicat à structurer.

Pour faire face à ces difficultés, les méthodes d’analyse et de conception proposent des guides structurant :

- organisation du travail en différentes phases (analyse, conception, codage, etc.) ou en niveaux d’abstraction (conceptuel, logique, physique),
- concepts fondateurs : par exemple les concepts de fonction, signal, état, objet, classe, etc.,
- représentations semi-formelles, documents, diagrammes, etc.

Dans cette approche le langage de modélisation est un formalisme de représentation qui facilite la communication, l’organisation et la vérification.

### 1.2.2 UML, un langage de modélisation

UML (Unified Modeling Language) est un langage de modélisation graphique véhiculant en particulier

- les concepts des approches par objets : classe, instance, classification, etc.
- intégrant d’autres aspects : associations, fonctionnalités, événements, états, séquences, etc.

UML est né en 1995 de la fusion de plusieurs méthodes à objets incluant OOSE (Jacobson), OOD (Booch), OMT (Rumbaugh).

UML propose d’étudier et de décrire un système informatique selon quatre points de vue principaux qui correspondent à quatre modèles (voir figure 1.1).

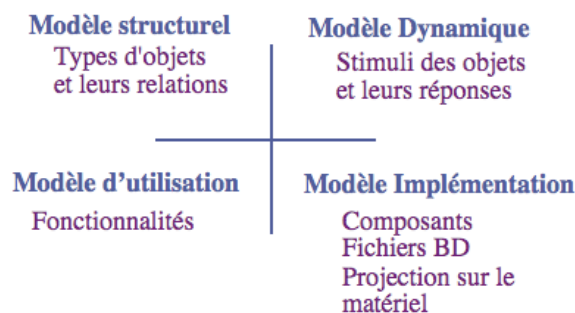


FIG. 1.1 – Vue générale des modèles UML

Chaque modèle est une représentation abstraite d’une réalité, il fournit une image simplifiée du monde réel selon un point de vue. Il permet :

- de comprendre et visualiser (en réduisant la complexité),

- de communiquer (à partir d’un langage commun à travers un nombre restreint de concepts),
- de mémoriser les choix effectués,
- de valider (contrôle de la cohérence, simuler, tester).

Dans chaque modèle, on écrit un certain nombre de diagrammes qui décrivent chacun certains aspects particuliers (voir Figure 1.2).

*Diagrammes (représentations graphiques de modèles)*

Diagrammes de classes d’instances	Diagrammes de collaboration de séquences d’états, d’activités
Diagrammes de cas d’utilisation	Diagrammes de déploiement de composants

FIG. 1.2 – Vue générale des diagrammes UML

L’un des atouts majeurs d’UML est que le langage sert dans la plupart des étapes de construction d’un logiciel, son rôle s’arrête juste pour la phase de codage (implémentation dans un langage de programmation).

### 1.3 Concrétisation en Java

Comme nous l’avons évoqué au début de l’introduction, nous utiliserons le langage Java pour projeter les constructions faites lors de la modélisation, donnant ainsi un aspect plus concret à ce cours.

Java est un langage de programmation à objets relativement récent (1991) et qui fait la synthèse de quelques-uns des langages existant à l’époque de sa création.

- Il emprunte une grande partie de sa syntaxe à C++ ;
- il recherchait à l’origine une plus grande simplicité que C++ ;
- il permet de s’abstraire des problèmes de gestion de la mémoire ;
- il n’est pas « tout objet » et n’a pas les capacités de réflexivité des langages à objets les plus avancés, mais en a cependant plus que C++ ;
- il fonctionne à l’aide de deux programmes, un compilateur et un interprète, et ce qui a fait en partie son succès est la possibilité d’avoir cet interprète dans tous les navigateurs internet.

## Chapitre 2

# Le modèle d'utilisation en UML

Le modèle d'utilisation délimite le système et décrit la manière de l'utiliser ; il oriente le développement entier du système. On l'appelle aussi le modèle fonctionnel.

Plus précisément il montre :

- Les fonctionnalités externes,
- le point de vue des utilisateurs,
- les interactions avec les acteurs extérieurs.

Les concepts les plus importants sont :

- la frontière qui délimite le système,
- les acteurs ; par acteur on entend toute entité extérieure au système et interagissant avec celui-ci. On trouve des acteurs humains et des acteurs machine (système extérieur communiquant avec le système étudié)
- Cas d'utilisation (toute manière d'utiliser le système).

La figure 2.1 montre la notation graphique utilisée pour les acteurs (petits personnages ou boîtes) et les cas d'utilisation (ovales). Entre acteurs et cas d'utilisation on trace des traits représentant leurs associations.

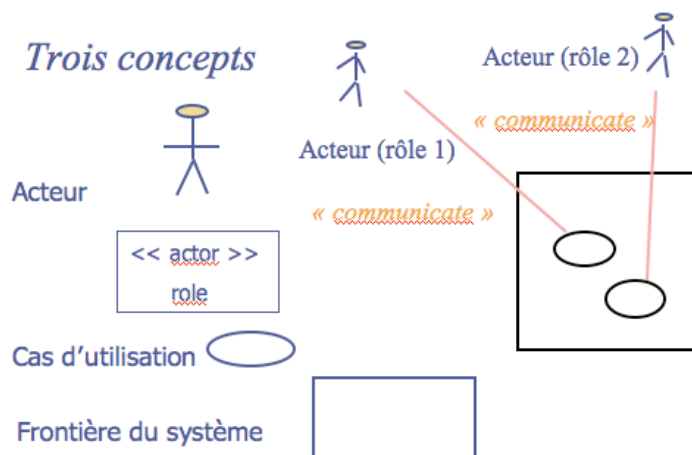


FIG. 2.1 – Éléments des diagrammes de cas d'utilisation

La figure 2.2 donne un exemple concret, où un client peut effectuer des commandes, tandis qu'un gestionnaire de stock effectue des livraisons. Notez le rectangle qui délimite le système : les cas d'utilisation sont à l'intérieur et les acteurs à l'extérieur. Elle montre également comment les cas d'utilisation peuvent se compléter les uns les autres.



Les cas d'utilisation peuvent être liés par des relations :

- d'utilisation **include** (le cas origine contient obligatoirement l'autre),
- de raffinement **extend** (le cas origine peut être ajouté optionnellement),
- de spécialisation (le cas origine est une forme particulière de l'autre).

Notez sur les figures la représentation graphique de ces relations.

Par exemple, le cas (la fonctionnalité) **commander** inclut (obligatoirement) le cas **décrire les produits** (Figure 2.2). Le cas **Paieement CB** est une spécialisation du cas **procéder au paieement**.

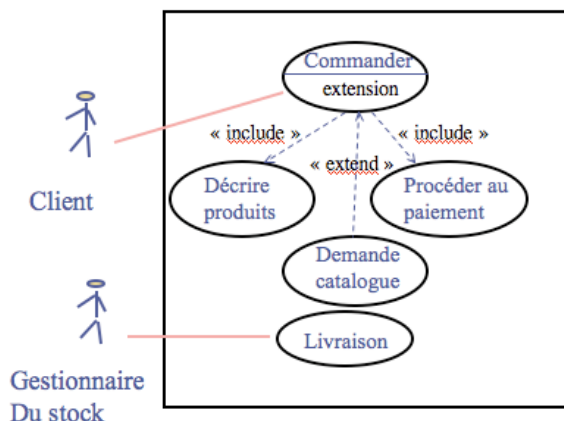


FIG. 2.2 – Relations include et extend

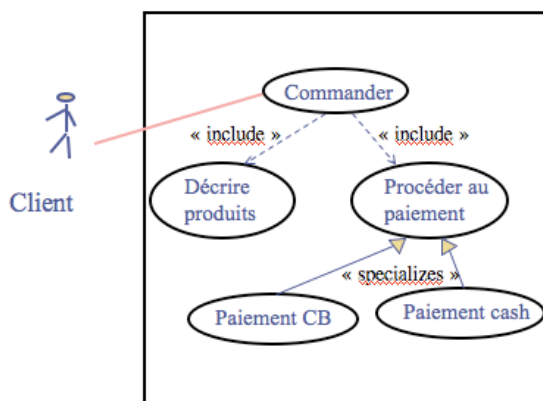


FIG. 2.3 – Relations de spécialisation entre cas d'utilisation

Les diagrammes de cas d'utilisation s'accompagnent le plus souvent de descriptions complémentaires, textes, diagramme de séquences ou d'activités, notamment pour décrire les aspects chronologiques dans l'utilisation du système.

Une proposition courante de description complémentaire comprend :

- Sommaire d'identification  
Titre, résumé, acteurs, dates de création et de mise à jour, version, auteurs ;
- Description des enchaînements  
Pré-conditions, scénario nominal, alternatives, exceptions, post-conditions ;

- Besoins IHM ;
- Contraintes « non fonctionnelles »  
Temps de réponse, concurrence, ressources machine, etc.

## Chapitre 3

# Classes et paquetages : les éléments de base du modèle statique

### 3.1 Les classes et instances en UML

Le modèle statique (ou structurel) se compose de deux types de diagrammes.

- Les diagrammes d’objets ou d’instances décrivent les objets du domaine modélisé et les éléments de la solution informatique (par exemple des personnes, des comptes bancaires), ainsi que des liens entre ces objets (par exemple le fait qu’une personne possède un compte bancaire) ;
- Les diagrammes de classes sont une abstraction des diagrammes d’objets : ils contiennent des classes qui regroupent des objets ayant des caractéristiques communes et des relations entre ces classes. De manière duale, les diagrammes d’instances doivent être conformes aux diagrammes de classes.

Voyons de plus près ces deux types de diagrammes.

Lors de l’analyse, notre esprit raisonne à la fois :

- par identification d’objets de base (Estelle, la voiture d’Estelle),
- par utilisation de ces objets comme des prototypes (la voiture d’Estelle vue comme une voiture caractéristique, à laquelle ressemblent les autres voitures, moyennant quelques modifications),
- par regroupement des objets partageant des propriétés structurelles et comportementales en classes.

Le deuxième mode de pensée a été exploré par une branche des langages à objets appelée les langages à prototypes qui sont moins connus que les langages dits à classes auxquels nous nous intéressons dans ce cours.

Dans le présent contexte des langages à classes, on dira souvent qu’une classe est un concept du domaine sur lequel porte le logiciel (voiture ou compte bancaire) ou du domaine du logiciel (par exemple un type de données abstrait tel que la pile). Une classe peut se voir selon trois points de vue :

- un aspect *extensionnel* : l’ensemble des objets (ou instances) représentés par la classe,
- un aspect *intensionnel* : la description commune à tous les objets de la classe, incluant les données (partie statique ou attributs) et les opérations (partie dynamique),
- un aspect *génération* : la classe sert à engendrer les objets.

À gauche de la figure 3.1, nous présentons une classe en notation graphique UML. Elle ne contient que des propriétés structurelles qui s'appellent des **attributs**. Ce sont des données décrivant l'objet (ici, le type, la marque et la couleur, toutes de type chaîne de caractères) et qui forment son **état**. En notation UML les diagrammes de classes montrent donc essentiellement l'aspect intensionnel des classes.

À droite de cette même figure 3.1, nous voyons un objet (ou instance) tel qu'en contiennent les diagrammes d'instances. Il s'agit ici d'une instance de la classe **Voiture**, qui se trouve décrite par une valuation des attributs. En l'absence d'ambiguïté, les noms des attributs peuvent être omis.

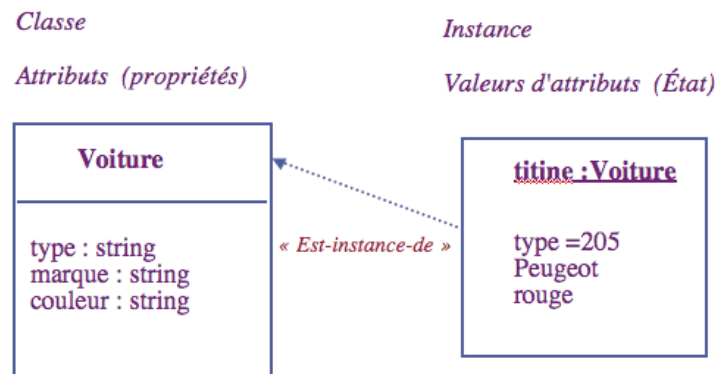


FIG. 3.1 – Classe (à gauche) et objet/instance (à droite)

Les attributs peuvent être décrits par de nombreux autres éléments que le type (voir exemple figure 3.2). La syntaxe est la suivante :

`[visibilité] [/]nom [ :type] [[multiplicité]] [= valeurParDéfaut]`

où *visibilité*  $\in \{+, -, \#, \sim\}$ , et *multiplicité* définit une valeur (1, 2, n, ...) ou une plage de valeurs (1..\*, 1..6, ...).

- la visibilité exprime la possibilité de référencer l'attribut suivant les contextes
  - Public. + est la marque d'un attribut accessible partout (public)
  - Privé. - est la marque d'un attribut accessible uniquement par sa propre classe (privé)
  - Package. ~ est la marque d'un attribut accessible par tout le paquetage
  - Protected. # est la marque d'un attribut accessible par les sous-classes de la classe
- le nom est la seule partie obligatoire de la description
- la multiplicité décrit le nombre de valeurs que peut prendre l'attribut (à un même moment)
- le type décrit le domaine de valeurs
- la valeur initiale décrit la valeur que possède l'attribut à l'origine
- des propriétés peuvent préciser si l'attribut est constant (`{constant}`), si on peut seulement ajouter des valeurs dans le cas où il est multi-valué (`{addOnly}`), etc.

Certains attributs peuvent être descriptifs de la classe elle-même plutôt que d'une instance, leur valeur est alors partagée par toutes les instances : ce sont les attributs *de classe* (voir figure 3.3). On les distingue des autres car ils sont soulignés.

Enfin certains attributs ont la particularité que leur valeur peut être déduite de la valeur d'autres attributs ou d'autres éléments décrivant la classe. Ce sont des attributs *dérivés* (voir figure 3.4).

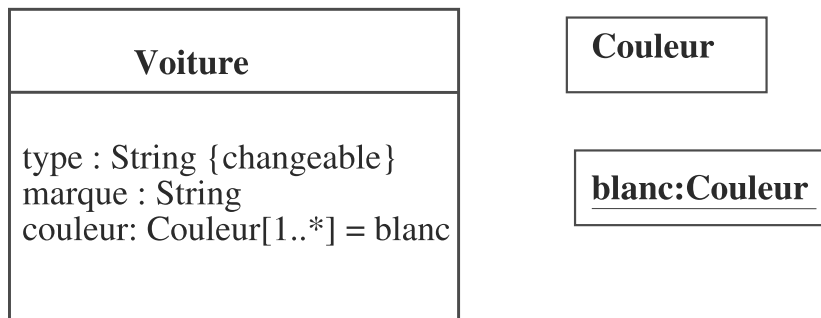


FIG. 3.2 – Détails sur la syntaxe de description des attributs

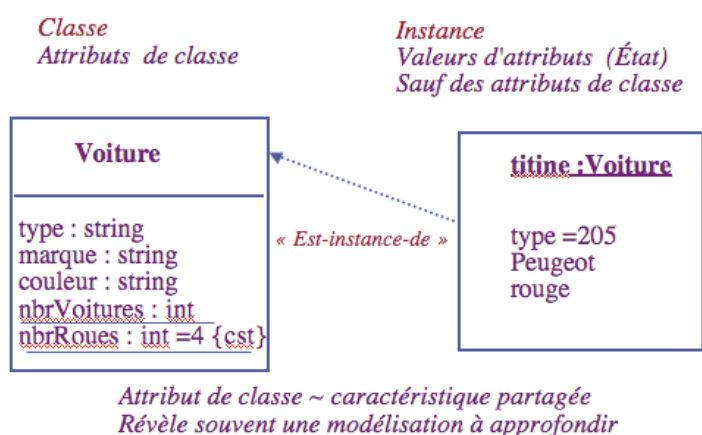


FIG. 3.3 – Attributs de classe

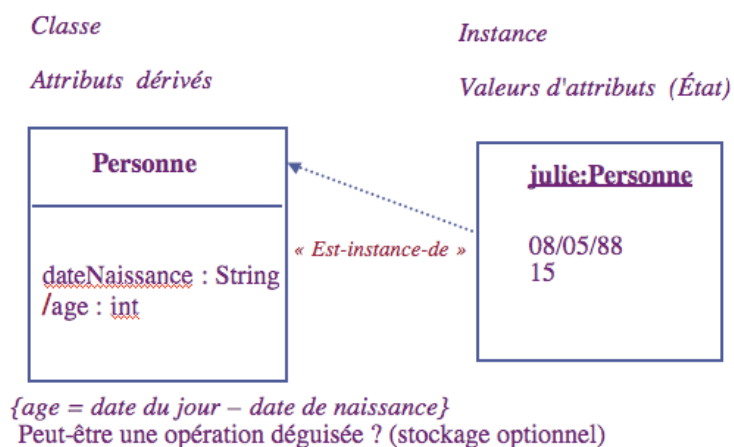


FIG. 3.4 – Attribut dérivé

## 3.2 Les paquets en UML

Un paquetage est un regroupement logique d'éléments UML, par exemple de classes. Les paquetages servent à structurer une application et sont utilisés dans certains langages, notamment Java, ce qui assure une bonne traçabilité de l'analyse à l'implémentation. Ils seront liés par des relations de dépendance dont nous reparlerons plus loin. Par exemple on regroupe dans le paquetage `VenteAutomobile` toutes les classes qui concernent ce domaine (Figure 3.5).

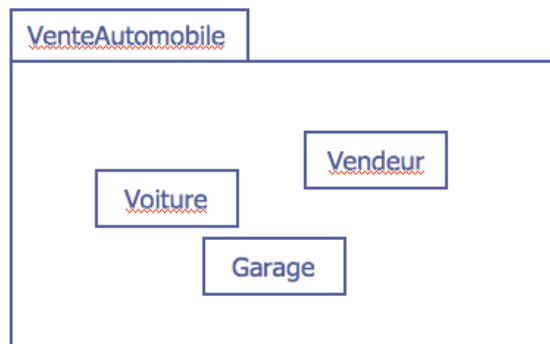


FIG. 3.5 – Paquetages en UML

## 3.3 Classes, instances et paquets en Java

À ce stade de notre cours, les classes et instances se traduisent assez directement dans le langage Java, procurant ce que l'on appelle des types construits, d'un plus haut niveau d'abstraction que les types de base (entiers, booléens, caractères), et plus proches des concepts manipulés en analyse.

### 3.3.1 Types de base en Java

Nous commençons cependant par évoquer ces types de base, car ils serviront en particulier à typer les attributs. Ils sont les suivants.

- **boolean**, constitué des deux valeurs **true** et **false**. Les opérateurs se notent :

Java	non	égal	différent	et alors / et	ou sinon / ou
	!	==	!=	&& &	 

- **int**, entiers entre  $-2^{31}$  et  $2^{31} - 1$
- **float**, **double**, ces derniers sont des réels entre  $-1.79E + 308$  et  $+1.79E + 308$  avec 15 chiffres significatifs.
- **char**, caractères représentés dans le système Unicode. Les constantes se notent entre apostrophes simples, par exemple `'A'`.
- **String**, qui n'est pas ... un type de base, c'est en réalité une classe mais nous rangeons ce type ici du fait de son usage très courant. Les chaînes de caractères constantes se notent entre guillemets, par exemple `"hello world"`. Les opérations seront déterminées par les méthodes de cette classe. Nous les verrons plus loin.

### 3.3.2 Écriture des classes

Nous donnons ici une traduction simplifiée à l'extrême de la classe `Voiture` qui serait incluse dans le paquetage `ExemplesCours1`. Notez les modificateurs **static** pour les attributs de classe, et **final** pour traduire le fait qu'un attribut est constant (plus précisément, on ne peut l'initialiser qu'une fois, mais l'initialisation peut être séparée de la déclaration : elle peut se faire par exemple dans un constructeur). Les attributs ont une valeur initiale implicite : 0 pour les nombres et null pour les références (variables désignant des objets).

```
package ExemplesCours1;
public class Voiture
{
    private String type; // null
    private String marque; // null
    private String couleur; // null
    private static int nbrVoitures; // 0
    private static final int nbrRoues = 4;
}
```

### 3.3.3 Création des instances

L'instruction suivante permet de déclarer une variable nommée `titine`.

```
Voiture titine;
```

Puis nous pouvons la créer.

```
titine = new Voiture();
```

`titine` doit être comprise comme une variable dont la valeur est une désignation de l'objet.

### 3.3.4 Accès aux attributs

Pour écrire des valeurs dans les attributs d'instance, nous utilisons des instructions d'affectation.

```
titine.type = "205";
titine.marque = "Peugeot";
titine.couleur = "rouge";
```

Grâce à elles, notre instance Java a à présent les mêmes valeurs que notre instance UML.

Pour écrire des valeurs dans les attributs de classe, on préfixe le nom de l'attribut par le nom de la classe.

```
Voiture.nbrVoitures = 3;
```

Toutes ces instructions ne peuvent bien entendu être écrites que dans les contextes où les attributs sont accessibles.

## Chapitre 4

# Opérations et méthodes

### 4.1 Classes, opérations et méthodes

Nous avons vu au chapitre précédent que l'on pouvait définir des classes, et leur associer des attributs. On peut ainsi définir ce qu'**est** un objet, mais pas ce qu'il fait, ou peut faire : c'est le rôle des opérations (terme UML) ou méthodes (terme Java).

Les méthodes / opérations définissent des comportements des instances de la classe. Par exemple, on a défini une classe voiture, on va maintenant voir comment exprimer ce que peut faire une voiture : klaxonner, fournir une assistance au parking, etc.

Les méthodes / opérations peuvent manipuler les attributs, ou faire appel à d'autres méthodes de la classe. Elles peuvent être paramétrées et retourner des résultats.

### 4.2 Opérations en UML

Les opérations sont les seuls éléments dynamiques du diagramme de classes. Elles se notent dans le compartiment inférieur des classes (voir figure 4.1).

#### Détail des opérations en UML (voir figure 4.2)

La syntaxe pour la déclaration des opérations est la suivante :  
[visibilité] nom ( liste-paramètres ) [ : typeRetour ] [liste-propriétés]

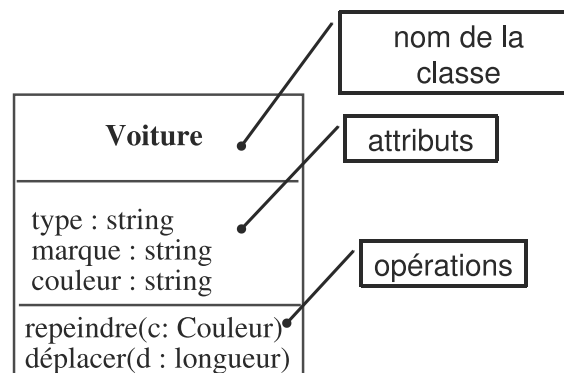


FIG. 4.1 – Les opérations dans les classes UML



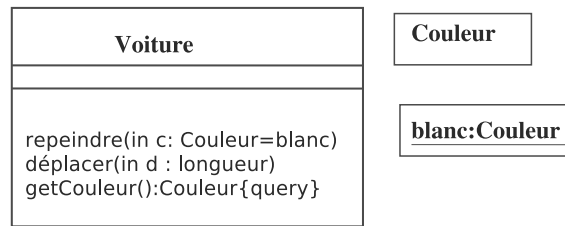


FIG. 4.2 – Exemple d’opérations en UML

où la syntaxe de chaque paramètre est :

[direction] nom : type[[multiplicité]] [= valeurParDéfaut] [liste-propriétés]

avec *direction*  $\in \{in, out, inout\}$ , et *multiplicité* définit une valeur (1, 2, n, ...) ou une plage de valeurs (1..\*, 1..6, ...). Une opération a un nom. On essaie en général de lui donner le nom portant le plus de sémantique possible : on évite d’appeler les opérations o1, o2, ou op, mais plutôt : klaxonner, déplacer, repeindre.

**Visibilité** Une opération a une visibilité.

- Publique. Dénuté +. Signifie que cette opération pourra être appelée par n’importe quel objet.
- Privée. Dénuté -. Signifie que cette opération ne pourra être appelée que par des objets instances de la même classe.
- Paquetage. Dénuté ~. Signifie que cette opération ne pourra être appelée que par des objets instances de classes du même paquetage.
- Protégée. Dénuté #. Signifie que cette opération ne pourra être appelée que par des objets instances de la même classe ou d’une de ses sous-classes (on verra plus tard ce que cela signifie exactement).

**Paramètres** Une opération peut avoir des paramètres. On peut spécifier le mode de passage d’un paramètre :

**in** le paramètre est une entrée de l’opération, et pas une sortie : il n’est pas modifié par l’opération. C’est le cas le plus courant. C’est aussi le cas par défaut en UML.

**out** le paramètre est une sortie de l’opération, et pas une entrée. C’est utile quand on souhaite retourner plusieurs résultats : comme il n’y a qu’un type de retour, on donne les autres résultats dans des paramètres out.

**inout** le paramètre est à la fois entrée et sortie.

**Propriétés** Une opération peut avoir des propriétés précisant le type d’opération, par exemple {query} spécifie que l’opération n’a pas d’effet de bord, ce n’est qu’une requête. Les propriétés sont placées entre accolades. Ces accolades signalent une valeur marquée (tagged value). Une valeur marquée a un nom, et peut contenir une valeur. Une valeur marquée peut être attachée à n’importe quel élément de modèle UML. Il existe des valeurs marquées pré-définies par UML, mais aussi définies par l’utilisateur, les valeurs marquées font donc partie des mécanismes d’extension d’UML.

## Opérations de classe

Une opération de classe est une opération qui ne s’applique pas à une instance de la classe : elle peut être appelée même sans avoir instancié la classe. Une opération de classe

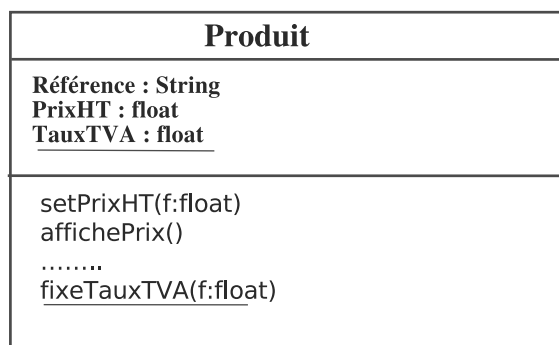


FIG. 4.3 – Opérations de classe

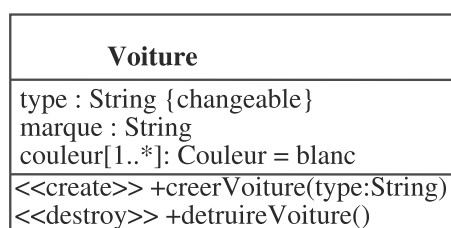


FIG. 4.4 – Constructeurs et destructeurs en UML

ne peut accéder qu'à des attributs et opérations de classe. En UML, les opérations de classe sont soulignées (voir Figure 4.3).

### Constructeurs et destructeurs

Il existe des opérations particulières qui sont en charge de la gestion de la durée de vie des objets : les constructeurs et les destructeurs. Un constructeur est une opération particulière d'une classe qui est l'opération qui permet de créer des instances de cette classe. Symétriquement, un destructeur est une opération particulière qui permet de détruire une instance de cette classe. En UML, pour préciser qu'une opération est un constructeur ou un destructeur, on place devant l'opération les stéréotypes <<create>> ou <<destroy>> (voir figure 4.4). Les stéréotypes se présentent comme des chaînes entre chevrons. Ce sont des étiquettes qui peuvent être attachées à n'importe quel élément de modèle UML, et qui donnent une sémantique particulière à l'élément de modèle.

### Le corps des opérations en UML

Nous avons vu jusqu'ici comment spécifier les signatures des opérations en UML, mais pas ce que font exactement les opérations, leur comportement. En UML, il n'y a pas à proprement parler de langage d'action permettant de spécifier le comportement des opérations. On peut par contre utiliser des diagrammes dynamiques pour les spécifier (nous verrons ces diagrammes plus tard). On peut aussi documenter l'opération avec du pseudo-code, dans une note de commentaire. On peut en effet attacher à tout élément de modèle UML une note contenant du texte (voir figure 4.5).

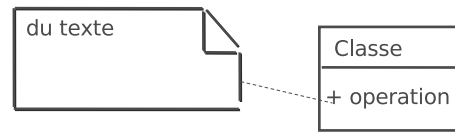


FIG. 4.5 – Note UML

## 4.3 Méthodes en Java

Nous allons voir comment écrire des méthodes en Java.

---

Listing 4.1 – Classe Voiture en java

---

```
1 package ExemplesCours2;
2 public class Voiture
3 {
4     private String type;
5     private String marque;
6     private String couleur;
7     private static int nbrVoitures;
8     private static final int nbrRoues = 4;
9
10    public Voiture(String leType, String laMarque, String couleur){
11        type=leType;
12        marque=laMarque;
13        this.couleur=couleur;
14    }
15
16    public static int getNbrRoues(){
17        return nbrRoues;
18    }
19
20    public String getMarque(){
21        return marque;
22    }
23
24    private void setMarque(String m){
25        marque=m;
26    }
27
28    public void repeindre(String c){
29        couleur=c;
30    }
31 }
```

---

### 4.3.1 Déclaration de méthodes

Le listing 4.1 illustre quelques déclarations de méthodes. On notera :

- la déclaration de constructeur : en Java, le constructeur d’une classe doit avoir le même nom que la classe, et il n’y a pas de type de retour.
- il n’y a pas vraiment de destructeur en Java. Il existe une méthode particulière nommée `finalize` qui est appelée quand le ramasse-miettes détruit l’objet car il n’est plus référencé.

- la déclaration de méthode de classe, avec le mot clef **static**. Pour appeler une méthode de classe, on préfixe le nom de l’opération par le nom de la classe ou par une instance de la classe si on en a une (on préférera le premier procédé).
- l’utilisation des mots clefs **private** et **public** pour définir la visibilité des méthodes
- il n’y a pas en Java la distinction entre paramètre in, out, ou inout.

On peut définir dans une même classe plusieurs méthodes portant le même nom, à condition que leur signature soient différentes. On peut en effet écrire une méthode `int add(int a, int b)` et une méthode `float add(float a, float b)` dans une même classe. Cette possibilité s’appelle la surcharge.

### 4.3.2 Exécution d’un premier programme

Le listing 4.2 donne un exemple de programme utilisant la classe `Voiture`.

Listing 4.2 – Utilisation de la classe `Voiture` en java

---

```
1 package ExemplesCours2;
2 public class essaiVoiture {
3     public static void main(String[] arg) {
4         Voiture v=new Voiture("C3", "Citroen", "rouge");
5         int nb=v.getNbrRoues();
6         System.out.println("Ma_"+v.getMarque()+"_a_"+nb+"_roues");
7     }
8 }
```

---

On notera :

- la méthode bizarre appelée `main` : c’est le point d’entrée de notre programme, c’est-à-dire que c’est elle qui est appelée quand on fait :  
    `> java ExemplesCours2.essaiVoiture`  
Cette méthode est statique : on n’a pas besoin de créer d’instance de la classe `essaiVoiture` pour utiliser la méthode `main`. Le paramètre correspond à ce qui est donné comme arguments en ligne de commande, ils sont stockés sous forme d’un tableau de chaînes. Nous verrons les tableaux ultérieurement.
- la concaténation de chaînes pour l’affichage, et la traduction automatique d’entiers en chaînes.

### 4.3.3 Les accesseurs

Les accesseurs sont des méthodes qui permettent d’accéder aux attributs, en lecture et en écriture. En Java, par convention ils sont notés `getAtt` et `setAtt` pour un attribut `att`. Leur signature est la suivante pour un attribut `att` de type `T` :

---

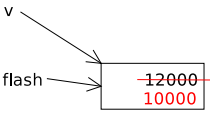
```
1 T getAtt()
2 void setAtt(T valeur)
```

---

Un exemple est donné au listing 4.4. Le `get` peut permettre de faire des statistiques sur les accès à l’attribut. Le `set` peut permettre d’effectuer des vérifications sur les valeurs, ou bien encore de prendre en charge des attributs dérivés. Un attribut dérivé peut être implémenté par une méthode, ou un attribut mis à jour quand cela est nécessaire. Ainsi le `set` d’un attribut peut permettre d’aller mettre à jour un attribut dérivé qui en dépend.

```
public class Voiture{
    public int prix;
    public Voiture(int p){
        prix=p;
    }
}

public class Expertise{
    public void expertiser(v:Voiture){
        v.prix=10000;
    }
    ...
    Voiture flash=new Voiture(12000);
    ...
    expertiser(flash);
}
```



Pendant l'exécution de la méthode `expertiser`, `v` et `flash` désignent le même objet.

FIG. 4.6 – Passage de paramètres par référence

#### 4.3.4 Quelques instructions de base

##### Affectation

L'affectation se note `=` (voir par exemple ligne 11 du listing 4.1).

##### Déclaration de variables locales

Dans le corps d'une méthode, on peut déclarer des variables locales. Par exemple :

```
int i;
int j=0;
Voiture v;
```

Pour les variables locales, on ne précise pas de visibilité : la portée de ces variables s'arrête à la fin de la méthode. On peut tout de suite initialiser les variables locales déclarées. Les variables locales n'ont pas de valeur initiale implicite.

##### Création de nouvelles instances

Pour créer de nouvelles instances d'une classe, on utilise le constructeur de la classe. On doit aussi utiliser le mot clef `new`.

```
Voiture v;
v=new Voiture("C4", "Citroen", "bleu");
```

##### Le passage de paramètres

Dans le corps d'une méthode, les paramètres sont comme des variables locales. Tout se passe comme si on avait des variables locales déclarées au début de la méthode, et qu'au début de méthode on affectait les valeurs des paramètres effectifs à ces variables locales. Les paramètres de type simple (types de base en Java comme `int` et `boolean`, dont le nom commence par une minuscule) sont passés par valeur. Tous les autres paramètres sont passés par référence (voir Figure 4.6)<sup>1</sup>. Nous allons illustrer le passage de paramètres sur un petit exemple, donné listings 4.3 et 4.4.

---

<sup>1</sup>On dit parfois que la référence est passée par valeur

Listing 4.3 – Echange.java

---

```
1 package ExemplesCours2;
2 public class Echange{
3     public void fauxEchange(int a, int b){
4         System.out.println("a="+a+" _b="+b);
5         int vi=a;
6         a=b;
7         b=vi;
8         System.out.println("a="+a+" _b="+b);
9     }
10
11     public void pseudoEchange(MonInt a, MonInt b){
12         System.out.println("a.getEntier="+a.getEntier()+" _b.getEntier="+b
13             .getEntier());
14         int vi=a.getEntier();
15         a.setEntier(b.getEntier());
16         b.setEntier(vi);
17         System.out.println("a.getEntier="+a.getEntier()+" _b.getEntier="+b
18             .getEntier());
19     }
20
21     public static void main(String[] args){
22         int x=2, y=3;
23         System.out.println("x="+x+" _y="+y);
24         Echange echange=new Echange();
25         echange.fauxEchange(x,y);
26         System.out.println("x="+x+" _y="+y);
27
28         MonInt xx=new MonInt(2);
29         MonInt yy=new MonInt(3);
30         System.out.println("xx.getEntier="+xx.getEntier()+" _yy.getEntier="+
31             yy.getEntier());
32         echange.pseudoEchange(xx,yy);
33         System.out.println("xx.getEntier="+xx.getEntier()+" _yy.getEntier="+
34             yy.getEntier());
35     }
36 }
```

---

Listing 4.4 – MonInt.java

---

```
1 package ExemplesCours2;
2 public class MonInt{
3
4     private int entier;
5
6     public MonInt(int e){
7         entier=e;
8     }
9
10    public int getEntier(){
11        return entier;
12    }
13
14    public void setEntier(int e){
```

```
15     entier=e;
16   }
17 }
```

---

### Désignation de l'instance courante

En Java, on désigne l'instance courante par le mot-clef **this**. On a besoin de cette désignation par exemple quand il y a conflit de noms (comme par exemple au listing 4.1) ou quand on veut passer l'instance courante en paramètre d'une méthode.

### L'instruction return

L'instruction **return** permet de retourner un résultat (voir l'exemple du listing 4.1). Un **return** provoque une sortie immédiate de la méthode : on ne doit donc jamais mettre de code juste sous un **return**, il ne serait pas exécuté. On ne peut utiliser un **return** que dans une méthode pour laquelle on a déclaré un type de retour, et bien sûr le type de l'objet retourné doit être cohérent avec le type de retour déclaré.

### Les commentaires

Il existe plusieurs formats pour les commentaires :

```
// ceci est un commentaire (s'arrête à la fin de la ligne)
/* ceci est un autre commentaire
   qui s'arrête quand on rencontre le marqueur de fin que voilà */
/** ceci est un commentaire particulier, utilisé par l'utilitaire javadoc **/
```

### Affichage

On peut afficher des données sur la console grâce à une bibliothèque java.

```
System.out.println("affichage puis passage à la ligne");
System.out.print("affichage sans ");
System.out.print("passer à la ligne");
```

### La méthode toString()

Toutes les classes disposent implicitement d'une méthode **String toString()** qui retourne une chaîne de caractères dont le rôle est de représenter une instance ou son état sous une forme lisible et affichable. Si on ne définit pas de méthode **toString** dans une classe, la méthode par défaut est appelée, elle retourne une désignation de l'instance. Il est conseillé de définir une méthode **toString** pour chaque classe. Nous verrons plus tard quel mécanisme se cache derrière cette méthode par défaut ... La méthode **toString** est illustrée au listing 4.5.

---

Listing 4.5 – *Personne.java*

---

```
1 package ExemplesCours2;
2 public class Personne{
3     private String nom;
4     private int numSecu;
5 }
```

```

6  public String toString(){
7      String result=nom+" "+age;
8      return result;
9  }
10 }
```

---

### 4.3.5 Structures de contrôle

#### Conditionnelles

**Conditionnelle simple** Syntaxe générale :

Listing 4.6 – Conditionnelle en Java

```

1      if (expression booléenne) {
2          bloc1
3      }
4      else {
5          bloc2
6      }
```

---

- La condition doit être évaluable en true ou false et elle est obligatoirement entourée de parenthèses.
- Les points-virgules sont obligatoires après chaque instruction et interdits après }.
- Si un bloc ne comporte qu'une seule instruction, on peut omettre les accolades qui l'entourent.
- Les conditionnelles peuvent s'imbriquer.

Listing 4.7 – Conditionnelle en Java

```

1  int a =3;
2  int b =4;
3  System.out.print("Le_plus_petit_entre_"+a+"_et_"+b+"_est_:");
4  if (b < a ) {
5      System.out.println(b);
6  }
7  else { System.out.println(a);
8  }
```

---

**L'opérateur conditionnel ( ) ? ... : ...** Le : se lit *sinon*.

```

1  System.out.println( (b < a) ? b : a );
2  int c = (b < a) ? a-b : b-a ;
```

---

**L'instruction de choix multiples** Syntaxe générale :

```

1  switch (expr entiere ou caractere) {
2      case i:
3      case j:
4          [bloc d'instructions]
5      .....break;
```



```
6 case _k:
7 ...
8 default:
9 .....
10 }
```

---

- L’instruction **default** est facultative ; elle est à placer à la fin. Elle permet de traiter toutes les autres valeurs de l’expression n’apparaissant pas dans les cas précédents.
  - Le **break** est obligatoire pour ne pas traiter les autres cas.
- 

```
1 int mois, nbJours;
2 switch (mois) {
3 case 1:
4 case 3:
5 case 5:
6 case 7:
7 case 8:
8 case 10:
9 case 12:
10  nbJours = 31;
11  break;
12 case 4:
13 case 6:
14 case 9:
15 case 11:
16  nbJours = 30;
17  break;
18 case 2:
19  if ( ((annee % 4 == 0) && !(annee % 100 == 0)) || (annee % 400 == 0)
20      )
21      nbJours = 29;
22  else
23      nbJours = 28;
24  break;
25 default nbJours=0;
26 }
```

---

### Boucles

**while** Syntaxe :

---

```
1 while (expression) {
2   bloc
3 }
```

---

```
1 int max = 100, i = 0, somme = 0;
2 while (i <= max) {
3   somme += i;      // somme = somme + i
4   i++;
5 }
```

---

**do while** Syntaxe :

---

```
1      do
2          { bloc }
3      while (expression)
```

---

---

```
1 int max = 100, i = 0, somme = 0 ;
2 do {
3     somme += i ;
4     i++;
5 }
6 while ( i <= max );
```

---

**for** Syntaxe :

---

```
1 for ( expression1; expression2 ; expression3 ){
2     bloc
3 }
```

---

- utilisée pour répéter N fois un même bloc d'instructions
- **expression1** : initialisation. Précise en général la valeur initiale de la variable de contrôle (ou compteur)
- **expression2** : la condition à satisfaire pour rester dans la boucle
- **expression3** : une action à réaliser à la fin de chaque boucle. En général, on actualise le compteur.

---

```
1 int somme = 0, max = 100;
2 for (int i =0 ; i <= max ; i++ ) {
3     somme += i;
4 }
```

---

### Instructions de rupture

- Pas de **goto** en Java;
- instruction **break** : on quitte le bloc courant et on passe à la suite;
- instruction **continue** : on saute les instructions du bloc situé à la suite et on passe à l'itération suivante.

## Chapitre 5

# Les associations UML, et leur implémentation en Java

### 5.1 Les associations

#### 5.1.1 Associations et liens

##### Définition

Une association est une relation entre 2 ou plusieurs classes qui décrit les connexions structurelles entre leurs instances. Par exemple, dans la figure 5.1, la classe Pays et la classe Ville entretiennent la relation *a pour capitale* (ou dans l'autre sens : *est la capitale de*). Au niveau instance, cela se traduit par un lien. Une association peut être définie entre 2 classes (le plus classique), entre  $n > 2$  classes (on parle alors d'association n-aire), sur une même classe (on parle alors d'association réflexive, voir Figure 5.4).

##### Nom, rôles, multiplicité, navigabilité

Une association binaire est représentée par un trait entre 2 classes, sur laquelle on apporte plusieurs informations (voir Figure 5.2) :

- le nom de l'association,
- le nom des rôles aux extrémités de l'association,
- la multiplicité des extrémités,
- la navigabilité.

Le nom de l'association est souvent augmenté d'une pointe de flèche ou d'un triangle qui précise dans quel sens l'association doit être lue (dans la Figure 5.1, on lit : “France a pour

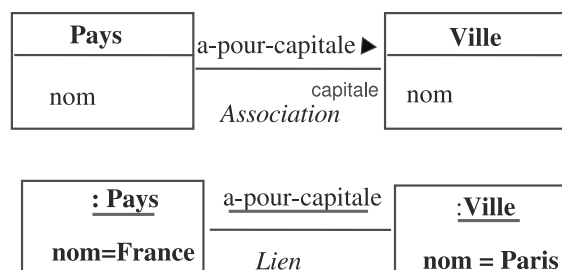


FIG. 5.1 – Associations et liens



FIG. 5.2 – Association binaire

capitale Paris”, et non pas “Paris a pour capitale France”.

Les noms des rôles sont placés aux extrémités de l’association. Un rôle placé à l’extrémité A d’une association entre A et B désigne le rôle que jouent les instances de la classe A dans l’association. Paris joue le rôle de capitale dans l’association `a_pour_capitale` entre Pays et Ville. Pour la figure 5.1, une personne joue le rôle de passager, le bus joue le rôle de véhicule.

La multiplicité (ou cardinalité) dénote le nombre d’instances impliquées de part et d’autre. Un bus transporte plusieurs passagers et un passager est transporté par un seul bus (à un instant *t*). La multiplicité peut être *i..j* (de *i* à *j*), *\** (plusieurs), *1..\** (au moins un), *i* (exactement *i*).

La navigabilité indique s’il est possible de traverser une association. On représente graphiquement la navigabilité par une flèche du côté de la terminaison navigable et on empêche la navigabilité par une croix du côté de la terminaison non navigable (mais ce n’est pas obligatoire). Dans la figure 5.2, le bus peut traverser l’association pour avoir connaissance de ses passagers, mais les passagers n’ont pas connaissance du bus qui les transporte. On n’a pas forcément le sens de lecture du nom de l’association dans le même sens que la navigabilité de l’association. Une association peut être navigable dans les 2 sens.

### 5.1.2 Associations et attributs

Il y a évidemment un lien entre associations et attributs. On pourrait se demander pourquoi introduire la notion d’association, puisqu’on pourrait toujours utiliser celle d’attribut. Il y a plusieurs raisons à cela :

- tout d’abord, pour une question de lisibilité des diagrammes : on voit bien mieux les liens entre classe avec une association qu’avec des attributs ;
- ensuite, avec une association, on définit 2 liens dépendants : les 2 bouts de l’association. Deux attributs de part et d’autre ne sont pas équivalents à une association ;
- on peut définir des associations complexes, comme on le verra par la suite.

On utilisera la convention suivante : les attributs sont uniquement de type simple (entiers, flottants, booléens, ...). Conséquence logique : on ne mettra jamais un attribut de type complexe, on préférera alors une association.

### 5.1.3 Agrégation et composition

On peut “renforcer” la notion d’association avec celle d’agrégation et de composition. Quand une association représente une relation *ensemble-élément*, on dira que l’association est une agrégation, dénotée par un losange non rempli du côté de l’ensemble. Quand l’association représente une relation de composition, on la note avec un losange plein du côté du composite. Derrière la notion de composition, il y a une notion d’exclusivité : un composant ne peut pas être partagé par plusieurs composites. Par exemple, à la figure 5.3 un département a une agrégation vers ses enseignants. Une université est composée de départements d’enseignement. Un département ne peut appartenir qu’à une seule univer-



FIG. 5.3 – Agrégation et composition

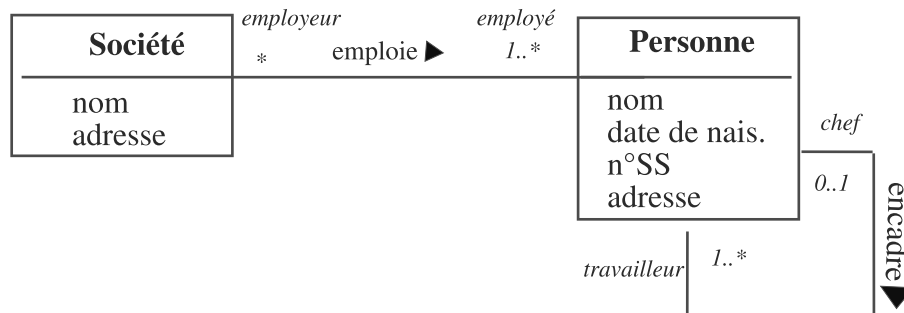


FIG. 5.4 – Association réflexive

sité. Un enseignant peut enseigner dans plusieurs départements. La sémantique exacte de l'agrégation et de la composition ne sont pas bien définies dans la norme UML, aussi de multiples interprétations existent. Nous en donnons ici la nôtre.

### Association Réflexive

Dans une association réflexive, le choix du nom des rôles et des cardinalités est particulièrement important. Voir Figure 5.4.

### Associations dérivées

Une association peut parfois être dérivée d'autres associations. On parle alors d'association dérivée, et on la marque par : / (voir Figure 5.5).

#### 5.1.4 Associations n-aires

La figure 5.6 montre une association ternaire.

#### 5.1.5 Associations qualifiées

Un qualifieur sur une association permet de sélectionner un sous-ensemble dans l'association (cela n'a de sens que pour une cardinalité \*). Par exemple, à la figure 5.7, à partir d'une banque et d'un numéro de compte, on sélectionne un unique compte. Ou bien, à partir d'une banque et d'un numéro de compte, on sélectionne au plus 2 personnes.

#### 5.1.6 Les classes d'association

Une classe d'association permet de rajouter des informations sur une association complexe (voir Figure 5.8). C'est une classe à part entière, elle peut donc être liée à d'autres

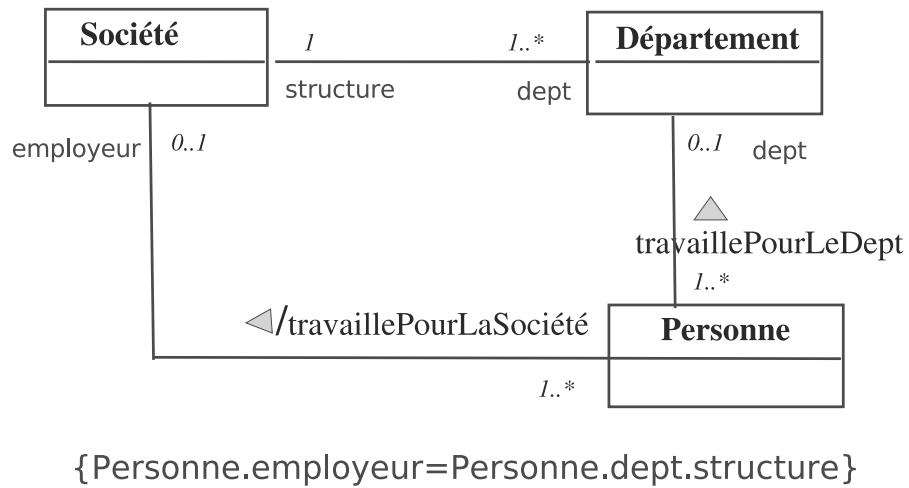


FIG. 5.5 – Association dérivée

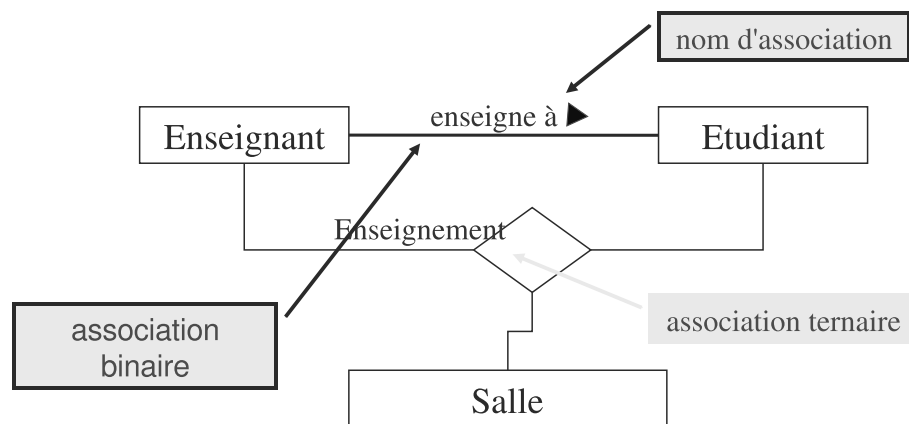


FIG. 5.6 – Association ternaire

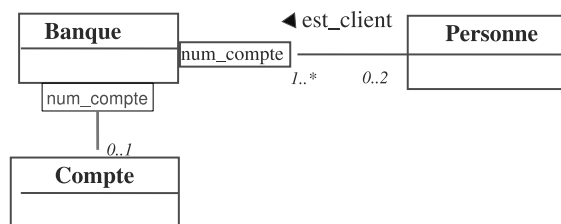


FIG. 5.7 – Associations qualifiées

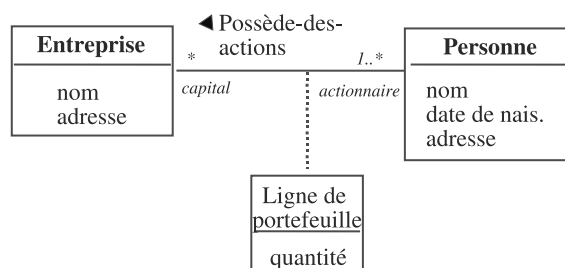


FIG. 5.8 – Classe d’association

classes.

## 5.2 Comment traduire les associations en Java ?

- On ne traduit que les extrémités navigables.
- Quand une extrémité a une cardinalité inférieure ou égale à 1, on traduit l’extrémité par un attribut.
- Quand on a une extrémité de cardinalité strictement supérieure à 1, on utilise les types de bibliothèque représentant des collections : listes, tableaux, ensembles, ...
- Quand on a une association bidirectionnelle, on veille bien à faire les mises à jour des 2 côtés. Ex : voiture et propriétaire.
- Quand on a une composition, on veille à bien respecter la contrainte de non partage de composants.

### 5.2.1 Les tableaux

En Java on sépare la déclaration d’une variable de type tableau, la construction effective d’un tableau et l’initialisation du tableau. La déclaration d’une variable de type tableau de nom `tab` dont les éléments sont de type `typ`, s’effectue par :

```
typ[] tab;
```

Lorsque l’on a déclaré un tableau en Java, il faut le créer avant de pouvoir le remplir. L’opération de construction s’effectue en utilisant un `new`, ce qui donne :

```
tab = new typ[taille];
```

Dans cette instruction, `taille` est une constante entière ou une variable de type entier dont l’évaluation doit pouvoir être effectuée à l’exécution. Une fois qu’un tableau est créé avec une certaine taille, celle-ci ne peut plus être modifiée. Cependant, la variable `tab` peut héberger par la suite un tableau de taille différente. Exemple :

```
int[] tab = new int[10];
for(int i = 0; i < 10; i++)
    tab[i] = i;
// autre solution
int[] tab2={0,1,2,3,4,5,6,7,8,9};
```

## tableaux à plusieurs dimensions

```
typ[] [] tab;
tab = new typ[N][M]; // N lignes, M colonnes
int[] [] tab = {{1,2,3},{4,5,6},{7,8,9}};
```

### 5.2.2 Les collections Java

En Java, il existe beaucoup de types complexes définis dans la bibliothèque Java permettant de représenter des collections : ensembles, listes, ... Ces types sont dits “génériques” : ils définissent un ensemble d’éléments de type E (où E est de type non primitif). Au moment de déclarer une collection, on précise si on veut une collection d’entiers (en utilisant le type `Integer` et non `int`), de voitures, de personnes, etc. On remplace le type générique E par le type souhaité. Il existe un certain nombre d’opérations définies pour toutes les collections : l’ajout et la suppression d’éléments, l’obtention de la taille de la collection, etc.

### 5.2.3 Les Vecteurs

Les vecteurs (`Vector`) sont des collections génériques qui ressemblent aux tableaux, mais dont la taille n’est pas prédéfinie. Quand on déclare un vecteur, on précise le type des objets qu’il va contenir : on déclare un vecteur de voitures, un vecteur de chaînes, un vecteur de personnes, etc. De la même façon, quand on crée le vecteur avec un `new`, le nom du type d’objets doit être précisé. La syntaxe pour déclarer et créer un vecteur `v` d’objets de type `MonType` est la suivante :

```
Vector<MonType> v; // déclaration
v=new Vector<MonType>(); // création
```

On peut bien sûr déclarer et créer un vecteur en une seule ligne :

```
Vector<MonType> v=new Vector<MonType>();
```

Pour manipuler un vecteur, on dispose de beaucoup d’opérations définies dans la classe `Vector`. Pour un vecteur d’objets de type E, on dispose entre autres des méthodes :

- `void addElement(E obj)` ou plus court `void add(E obj)`. Ajoute l’objet `obj` à la fin du vecteur, et augmente sa taille de 1.
- `E get(int index)`. Retourne l’objet placé en position `index` dans le vecteur.
- `boolean isEmpty()`. Teste si le vecteur n’a aucun élément.
- `int size()`. Retourne la taille du vecteur.
- `E remove(int index)`. Supprime l’objet en position `index` et le retourne.
- `boolean remove(Object o)`. Supprime la première occurrence de `o` rencontrée (laisse le vecteur inchangé si aucun tel objet n’est rencontré, et retourne alors faux).

Voilà un petit exemple d’utilisation des vecteurs :

```
// Création d’un ensemble capable de ne stocker que des objets String
Vector<String> prenom = new Vector<String> ();
prenom.addElement("Thomas");
prenom.addElement("Sophie");
// La méthode add interdit d’ajouter un autre type d’objet
// prenom.addElement(new Voiture()); -> interdit
```



```
//affichage des prénoms
for (int i = 0; i < prenom.size(); i++)
{
    System.out.println(prenoms.get(i));
}
// autre affichage des prénoms avec la boucle for existant depuis Java 1.5
for (String prenomCourant:prenom){
    System.out.println(prenomCourant);
}
```

### 5.2.4 Les tables de hachage

On traduit en général une association qualifiée par une table de hachage. Une table de hachage est une table indexée par une clef, c'est à dire un identifiant permettant d'accéder directement à l'élément correspondant à cet identifiant dans la table. Par exemple, on peut envisager une table de hachage de comptes bancaires, indexés par leur numéro de compte, que l'on sait unique. Ici, la clef est le numéro de compte, et est de type entier. La table contient des comptes, auxquels on accède directement grâce à la clef. Le terme de hachage est dû au mécanisme qui est mis en œuvre pour permettre cet accès direct et rapide à partir d'une clef, mais que nous ne verrons pas ici. Pour déclarer et créer une table de hachage contenant des objets de type `MonType` indexés par des objets de type `Clef`, on écrit :

```
Hashtable<Clef, MonType> table=new Hashtable<Clef, MonType>();
```

Pour une table d'objets de type `V` et de clefs de type `K`, on a par exemple les méthodes suivantes :

- `V get(Object key)`. Retourne l'objet de clef `key` ou `null` s'il n'y en a pas.
- `V put(K key, V value)`. Ajoute l'élément `value` avec comme clef `key`. S'il existait déjà un élément de même clef, cet élément est retourné (et écrasé dans la table par `value`).
- `V remove(Object key)`. Retire l'élément de clef `key` de la table.

Voilà un petit exemple d'utilisation des tables de hachage :

```
Hashtable<String, Integer> numbers = new Hashtable<String, Integer>();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));

Integer n = numbers.get("two");
if (n != null) {
    System.out.println("two = " + n);
}
```

### 5.2.5 Ce qu'on ne peut pas traduire directement

On ne peut pas traduire directement les classes d'association : on les transforme en général en classes "normales" fortement associées aux autres classes. On ne peut pas traduire directement une association ternaire ou n-aire  $n > 2$  : on réifie en général l'association.

## Chapitre 6

# Les diagrammes dynamiques en UML

### 6.1 Introduction

Les diagrammes UML que nous avons vus jusqu'ici (diagrammes de cas d'utilisation, d'instances et de classe) permettent de modéliser les fonctions requises et la structure d'un système, mais pas sa dynamique. Nous avons pu spécifier des classes mais pas comment les instances de ces classes interagissent entre elles. De même, nous avons défini des signatures de méthodes, mais pas le comportement de ces méthodes. UML permet de modéliser cette dynamique du système, avec plusieurs types de diagrammes. Les diagrammes dynamiques ne sont pas déconnectés des diagrammes statiques : ce sont d'autres facettes du même système.

- Les diagrammes de séquence permettent de représenter des échanges de messages entre des instances, en mettant l'accent sur l'ordre dans lequel les échanges ont lieu.
- Les diagrammes de communication (anciennement appelés de collaboration) permettent eux aussi de représenter des échanges de messages entre des instances, mais en mettant l'accent sur les interactions existant entre les instances.
- Les diagrammes d'états-transitions, ou machines à états, permettent de représenter les différents états d'un élément de modèle (souvent d'une classe), et la façon de passer d'un état à un autre. C'est une sorte d'automate.
- Les diagrammes d'activités permettent de représenter des flots de contrôle et/ou de données entre des actions.

Nous ne développerons ici que les diagrammes de séquence, pour des raisons pédagogiques.

### 6.2 Les diagrammes de séquence

Les diagrammes de séquence permettent de représenter les interactions entre des instances particulières. Un diagramme met en jeu :

- des instances, et éventuellement des acteurs,
- des messages échangés par ces instances. Un message définit une communication entre instances. Ce peut être par exemple l'émission d'un signal, ou l'appel d'une opération.

Un diagramme de séquence se présente sous la forme d'une grande boîte avec noté dans le coin supérieur droit (dans un pentagone non régulier) : `sd : nomDiagramme`. Dans ce grand rectangle, apparaissent un ensemble d'instances (représentées comme dans

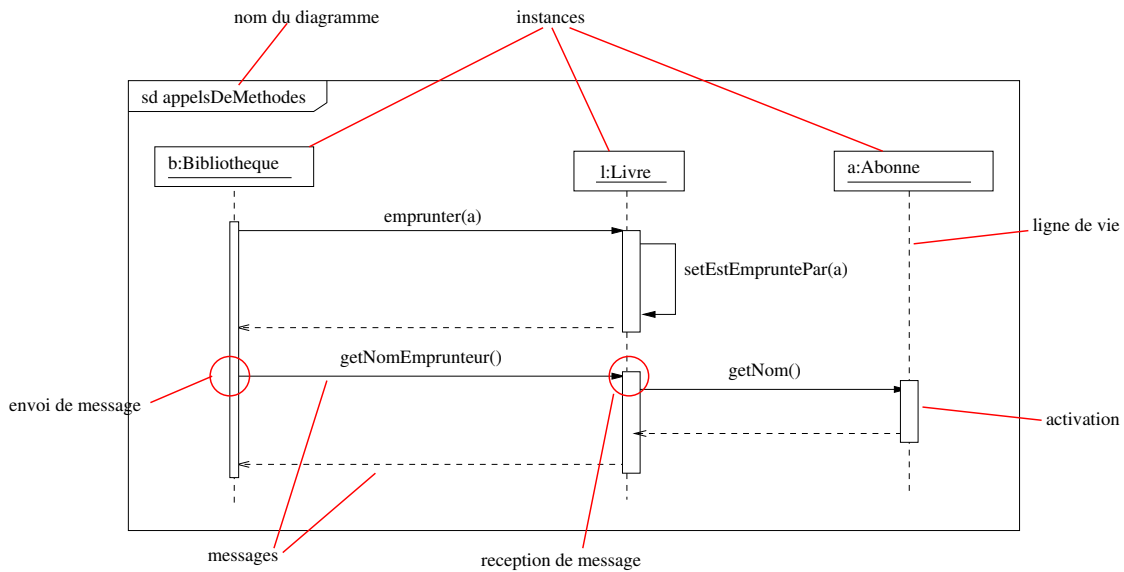


FIG. 6.1 – Premier exemple de diagramme de séquence

les diagrammes d'instance par des rectangles où apparaît : `nomInstance :Type`), dotées d'une ligne de vie (trait pointillé vertical, placé sous la boîte de l'instance). Les instances échangent des messages, qui apparaissent sous forme de flèches. Le diagramme de séquence permet d'insister sur la chronologie des interactions : le temps s'écoule grosso modo du haut vers le bas. Les diagrammes de séquence ont été profondément modifiés lors du passage d'UML1.x à UML2.0, et à l'heure actuelle, peu de gens utilisent la nouvelle notation.

La figure 6.1 illustre les principaux concepts mis en jeu dans un diagramme de séquence.

### 6.2.1 La ligne de vie

À chaque instance est associée une ligne de vie, qui représente la vie de l'objet. Les événements survenant sur une ligne de vie (réception de message ou envoi de message) sont ordonnés chronologiquement, de haut en bas (sur une même ligne de vie, un événement plus bas qu'un autre signifie qu'il a lieu après). La ligne de vie est représentée par une ligne pointillée quand l'instance est inactive, et par une boîte blanche ou grisée quand l'instance est active. Quand une instance est détruite, on stoppe la ligne de vie par une croix (voir Figure 6.2).

### 6.2.2 Les messages

Les messages sont représentés par des lignes fléchées. À chaque extrémité de la ligne fléchée correspond un événement (réception ou envoi). Le sens de la flèche permet de déterminer dans quel sens va le message. Une ligne continue avec flèche simple représente un message asynchrone, une ligne continue avec flèche pleine représente un appel synchrone, une ligne pointillée avec flèche simple représente un retour synchrone (voir Figure 6.3). Quand une instance envoie un message, elle peut rester bloquée en attendant la réponse à ce message. On parle alors de message synchrone (ou plutôt d'une paire de messages synchrones : l'envoi et la réception). C'est classiquement le cas quand on appelle une méthode dans un langage de programmation comme Java. Imaginons qu'une instance A

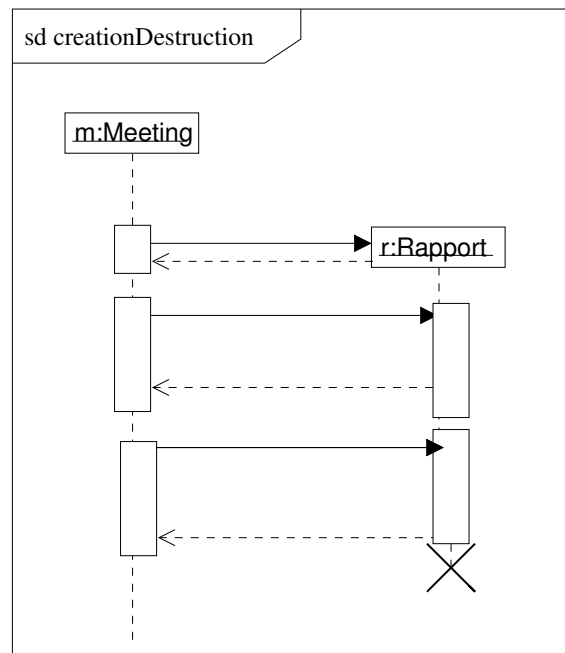


FIG. 6.2 – Ligne de vie

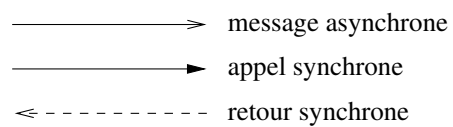


FIG. 6.3 – Messages

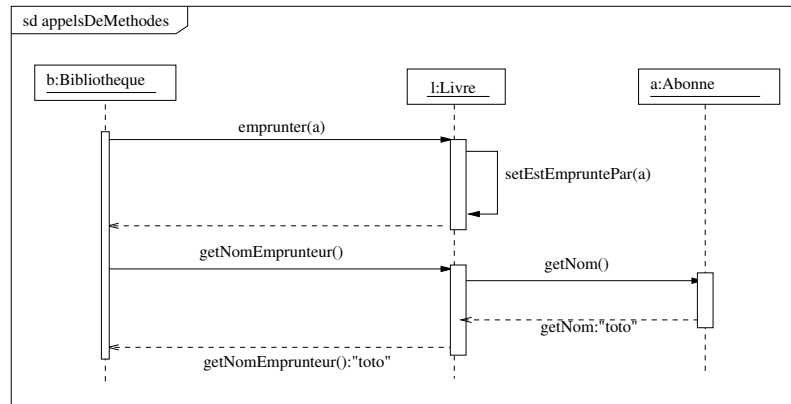


FIG. 6.4 – Appels de méthodes

appelle une méthode très longue sur une autre instance B. L'instance A se met alors en attente de la réponse de B, et ne fait rien pendant cette attente (notamment, elle n'exécute pas la ligne de code du dessous). Quand une instance envoie un message mais n'attend pas de réponse à ce message et peut continuer tout de suite à faire d'autres traitements, alors on parle de message asynchrone.

Quand on veut spécifier qu'un message crée une nouvelle instance, on fait pointer le message sur la boîte représentant l'instance (voir Figure 6.2).

La syntaxe pour le nom d'un message est :

```

([attribut =] signal-ou-NomOperation [( [liste-arguments]]) [ : valeur-retour] )
| *

```

où la syntaxe pour un argument est :

```

([nomParam =] valeur-argument) | (attribut = nomParamOut [ : valeurArgument] )
| -

```

\* signifie : n'importe quel type de message

- signifie : paramètre indéfini

Par exemple, on peut avoir les noms de message suivants :

- getAge()
- getAge() :12
- age=getAge() :12
- setAge(age=15)
- setAge(-)

La figure 6.4 illustre un appel de méthode.

Les diagrammes de séquence ne sont pas à concevoir indépendamment des autres diagrammes, comme par exemple le diagramme de classes. Ainsi, quand un message d'appel d'une méthode *m* arrive sur une ligne de vie d'une instance *a* : *A*, alors il doit y avoir une méthode *m* définie dans la classe *A*.

### Messages perdus et trouvés

Dans un diagramme de séquence, on peut représenter la réception de messages dont on ne connaît pas l'émetteur, et réciproquement l'envoi de messages dont on ne connaît pas le destinataire. C'est ce qu'on appelle des messages respectivement trouvés et perdus. On les représente par des messages sortant ou entrant dans un petit disque noir (voir Figure 6.5).

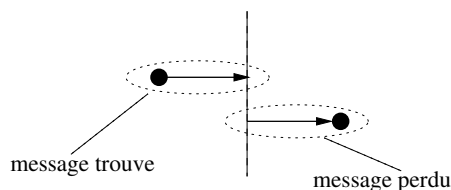


FIG. 6.5 – Messages trouvé et perdu

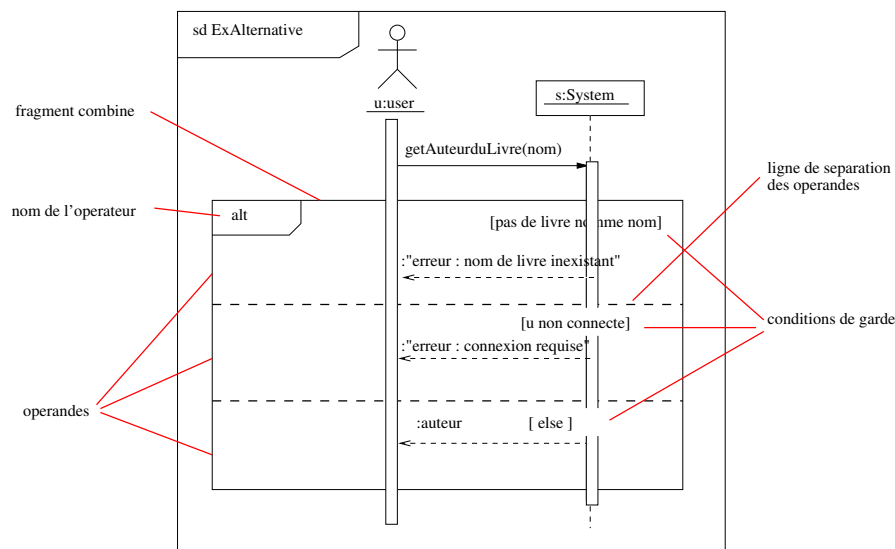


FIG. 6.6 – Opérateurs de composition et fragments combinés

On représente un message trouvé quand l'envoi du message est en dehors des objectifs du diagramme de séquence. Par exemple, si un diagramme sert à montrer la réaction d'une instance quand on appelle une de ses méthodes avec certains paramètres, il n'est pas utile de représenter quelle autre instance appelle cette méthode (cela pourrait nuire à la compréhension du diagramme de séquence).

On représente un message perdu quand le message n'arrive pas à destination.

### 6.2.3 Composition de fragments de diagrammes de séquence

Depuis la version 2.0 d'UML, on peut désormais composer des fragments de diagramme de séquence entre eux. Pour cela, un certain nombre d'opérateurs de composition ont été définis. De manière générale (voir Figure 6.6) :

- On représente les opérateurs par des grandes boîtes avec un compartiment dans le coin supérieur gauche, en forme de pentagone irrégulier, et dans lequel on place le nom de l'opérateur ainsi que (parfois) d'autres informations.
- Les opérateurs ont des opérandes (1, 2, plusieurs), qui sont représentées les unes en dessous des autres et séparées par une ligne pointillée.

#### Alternative et optionnalité

L'opérateur alternative (noté **alt**) permet de représenter le choix (exclusif) entre plusieurs comportements alternatifs (voir Figure 6.7). Chacun des comportements alternatifs

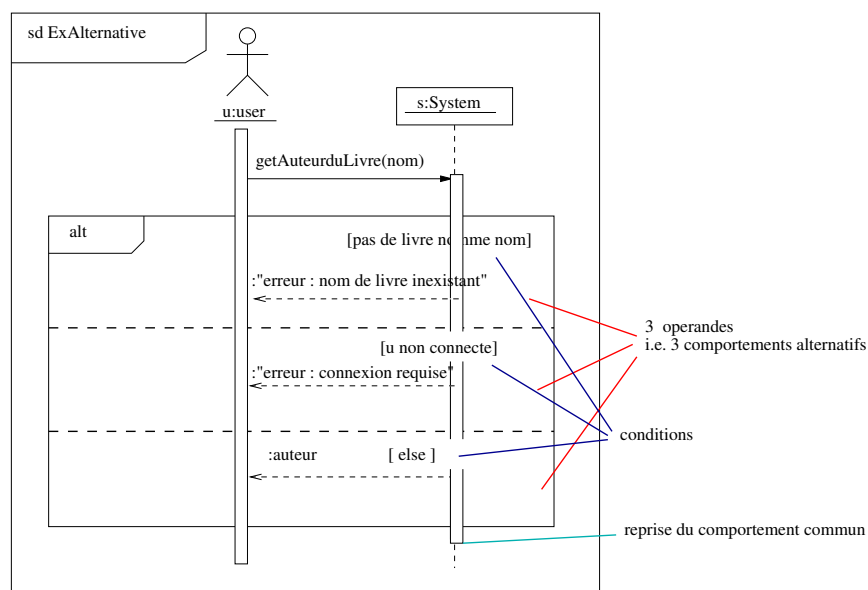


FIG. 6.7 – Comportements alternatifs

détient une condition de garde (l'absence de condition de garde implique une condition vraie). La condition **else** est vraie si aucune autre condition n'est vraie. Si plusieurs conditions de garde sont vraies, le choix est non déterministe.

L'opérateur d'optionnalité (noté **opt**) permet de représenter un comportement qui n'a lieu que si une condition de garde est vraie. Cet opérateur peut être construit à partir d'une alternative : un comportement C optionnel est équivalent à une alternative entre le comportement C ou sinon rien.

### Composition parallèle

L'opérateur de composition parallèle (noté **par**) permet de spécifier des comportements qui peuvent avoir lieu en parallèle les uns des autres. Cet opérateur est n-aire. Quand un comportement A est en parallèle avec un comportement B, l'ordre partiel des événements de A et de B est conservé.

Il existe un raccourci syntaxique appelé *corégion* permettant de représenter le cas où sur une même ligne de vie, les événements peuvent survenir dans n'importe quel ordre. Ceci est noté par des crochets verticaux délimitant la corégion sur la ligne de vie.

### Composition séquentielle faible et forte

Les opérateurs de composition séquentielle (notés **seq** pour la composition séquentielle faible et **strict** pour la composition séquentielle forte) permettent de spécifier que des comportements se déroulent les uns à la suite des autres. Il existe 2 types de composition séquentielle : forte et faible. Pour la composition séquentielle faible :

- L'ordre partiel des événements de chaque opérande est maintenu.
- Deux événements intervenant sur des lignes de vie différentes d'opérandes différentes peuvent avoir lieu dans n'importe quel ordre.
- Les événements qui sont sur la même ligne de vie d'opérandes différentes sont ordonnés de telle sorte que les événements de la première opérande surviennent avant

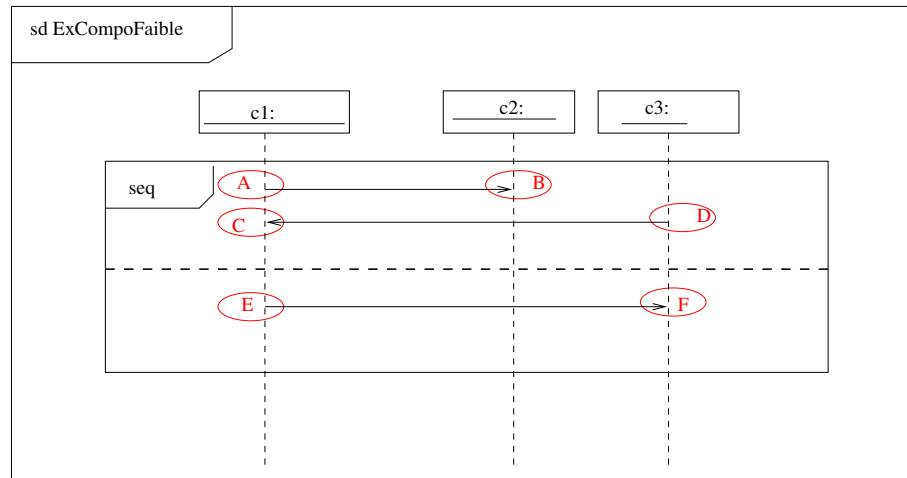


FIG. 6.8 – Composition séquentielle faible (seq)

ceux de la deuxième opérande.

La composition séquentielle faible est illustrée à la figure 6.8, où deux comportements sont composés par l'opérateur **seq**. Les événements intervenant dans ce diagramme de séquence ont été nommés de A à F. L'ordre partiel des événements respecte les contraintes suivantes (où la précédence temporelle est notée  $\prec$ ) :

- $A \prec B$  (car A et B sont les 2 événements associés à un envoi de message)
- $D \prec C$  (pour les mêmes raisons)
- $E \prec F$  (pour les mêmes raisons)
- $C \prec E$  (C et E interviennent sur la même ligne de vie d'opérandes différentes. La composition séquentielle faible impose que sur une même ligne de vie, les événements de la première opérande interviennent avant ceux de la seconde).
- $D \prec F$  (pour les mêmes raisons).

Dans ce diagramme de séquence, on peut très bien avoir  $B \succ E$  ! Ceci est illustré à la figure 6.9. Le message de c1 vers c2 est très long à arriver à c2, pendant ce temps, c3 envoie un message à c1, puis c1 envoie un message à c3.

Pour la composition séquentielle forte, on impose une synchronisation globale à toutes les lignes de vie des opérandes : tous les événements de la première opérande ont lieu avant tous les événements de la seconde opérande. La composition séquentielle forte est illustrée à la figure 6.10, où deux comportements sont composés par l'opérateur **strict**. Les événements intervenant dans ce diagramme de séquence ont été nommés de A à F. L'ordre partiel des événements respecte les contraintes suivantes (où la précédence temporelle est notée  $\prec$ ) :

- $A \prec B$  (car A et B sont les 2 événements associés à un envoi de message)
- $D \prec C$  (pour les mêmes raisons)
- $E \prec F$  (pour les mêmes raisons)
- $C \prec E$  (C et E interviennent sur la même ligne de vie d'opérandes différentes. La composition séquentielle forte impose que les événements de la première opérande interviennent avant ceux de la seconde).
- $D \prec F$  (pour les mêmes raisons).
- $B \prec E$  (B est un événement de la première opérande, E est un événement de la deuxième opérande, la composition séquentielle forte impose une synchronisation globale).



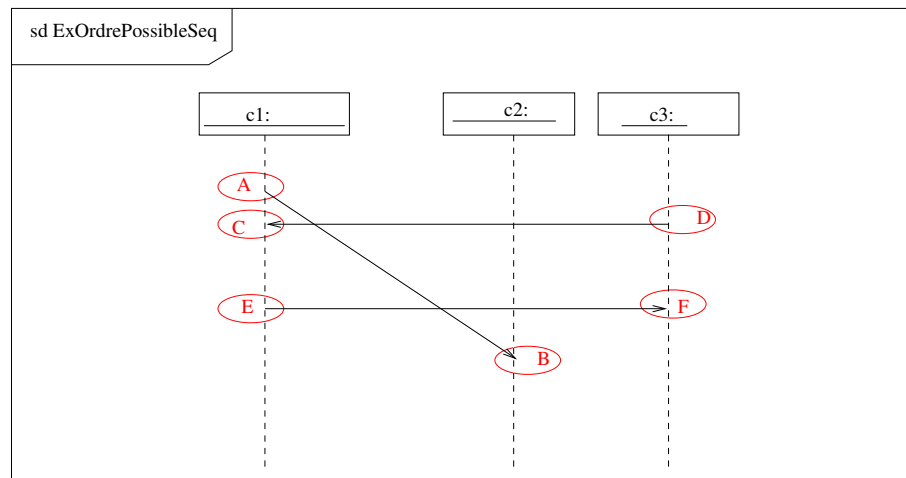


FIG. 6.9 – Un diagramme de séquence pouvant résulter de la composition séquentielle faible de la figure 6.8

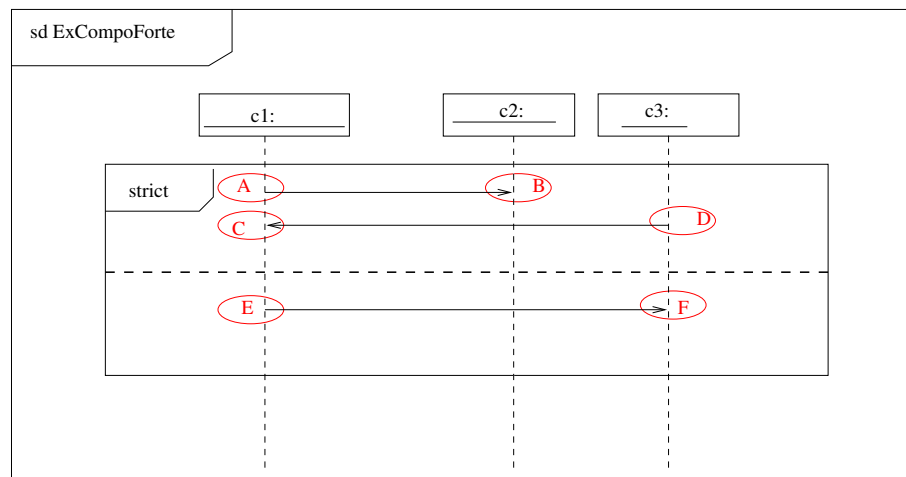


FIG. 6.10 – Composition séquentielle forte (strict)

## Boucle

L'opérateur `loop` permet d'itérer des comportements. On doit pour cela spécifier :

- le nombre minimum `minInt` de tours de boucles,
- le nombre maximum `maxInt` de tours de boucle (\* signifie infini),
- une condition de garde,
- une unique opérande représentant le comportement sur lequel on boucle. On itère sur le comportement spécifié au minimum `minInt` fois, avant de tester la condition de garde. Si celle-ci est fausse, on sort. Sinon, on continue d'itérer, soit jusqu'à ce que la condition de garde soit fausse, soit jusqu'à avoir itéré `maxInt` fois.

La syntaxe de la boucle est la suivante :

```
loop[ (minInt [ , maxInt ] ) ]
```

Si les nombres `minInt` et `maxInt` ne sont pas spécifiés, alors par défaut, `minInt=0` et `maxInt=*`. La condition de garde est placée entre crochets sur la ligne de vie.

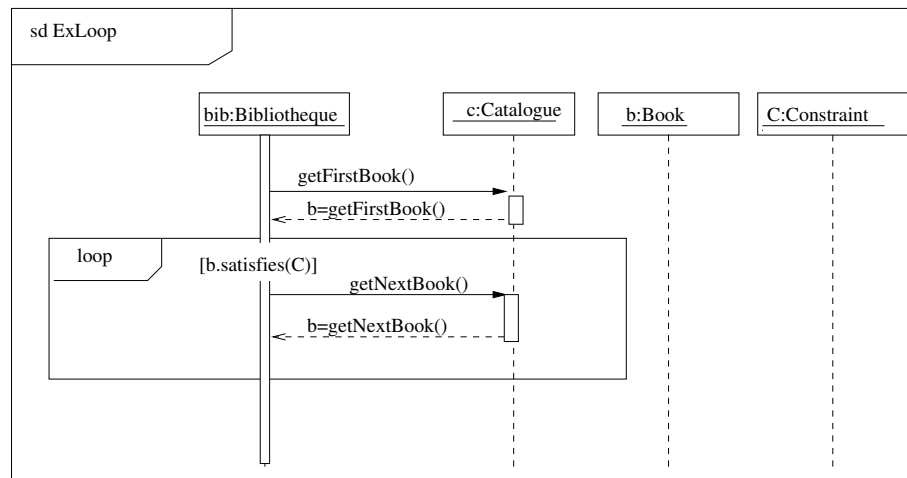


FIG. 6.11 – Boucles dans les diagrammes de séquence

La figure 6.11 illustre une boucle.

### Section critique

L'opérateur **critical** désigne une section critique. Une section critique est une section ininterrompible par aucune autre interaction décrite dans le diagramme. Cet opérateur impose un traitement atomique des interactions qu'il contient.

### Neg

L'opérateur **Neg** (ou **negative**) permet de spécifier des interactions non souhaitées : le comportement opérante de l'opérateur **neg** est spécifié comme invalide.

### Break

L'opérateur **break** permet de définir un fragment à réaliser quand une certaine condition est satisfaite au lieu du reste de l'interaction normale.

## Chapitre 7

# Spécialisation/généralisation et héritage

### 7.1 Généralisation - Spécialisation

Nous abordons dans ce chapitre, un des atouts majeurs de la programmation objet et qui a pour point de départ un « double » mécanisme d'inférence intellectuelle : la généralisation et la spécialisation, deux mécanismes relevant d'une démarche plus générale qui consiste à « classifier » les concepts manipulés.

#### 7.1.1 Classer les objets

La généralisation est un mécanisme qui consiste à réunir des objets possédant des caractéristiques communes dans une nouvelle classe plus générale appelée **super-classe**.

Prenons un exemple décrit par la figure 7.1. Nous disposons dans un diagramme de classes initial d'une classe **Voiture** et d'une classe **Bateau**. Une analyse de ces classes montre que leurs objets partagent des attributs et des opérations : on abstrait une super-classe **Véhicule** qui regroupe les éléments communs aux deux sous-classes **Voiture** et **Bateau**. Les deux sous-classes héritent les caractéristiques communes définies dans leur super-classe **Véhicule** et elles déclarent en plus les caractéristiques qui les distinguent (ou bien elles redéfinissent selon leur propre point de vue une ou plusieurs caractéristiques communes).

Le mécanisme dual, la spécialisation est décrit à partir de l'exemple de la figure 7.2.

Ici la spécialisation consiste à différencier parmi les bateaux (la distinction s'effectuant selon leur type), les sous-classes **Bateau\_à\_moteur** et **Bateau\_à\_voile**. La spécialisation peut faire apparaître de nouvelles caractéristiques dans les sous-classes.

Il faut retenir que du point de vue de la modélisation, une classe  $C_{mere}$  généralise une autre classe  $C_{fille}$  si l'ensemble des objets de  $C_{fille}$  est inclus dans l'ensemble des objets de  $C_{mere}$ .

Du point de vue des objets (instances des classes), toute instance d'une sous-classe peut jouer le rôle (peut remplacer) d'une instance d'une des super-classes de sa hiérarchie de spécialisation-généralisation.

#### 7.1.2 Discriminants et contraintes

Les relations de spécialisation/généralisation peuvent être décrites avec plus de précision par deux sortes de description UML : les contraintes et les discriminants.

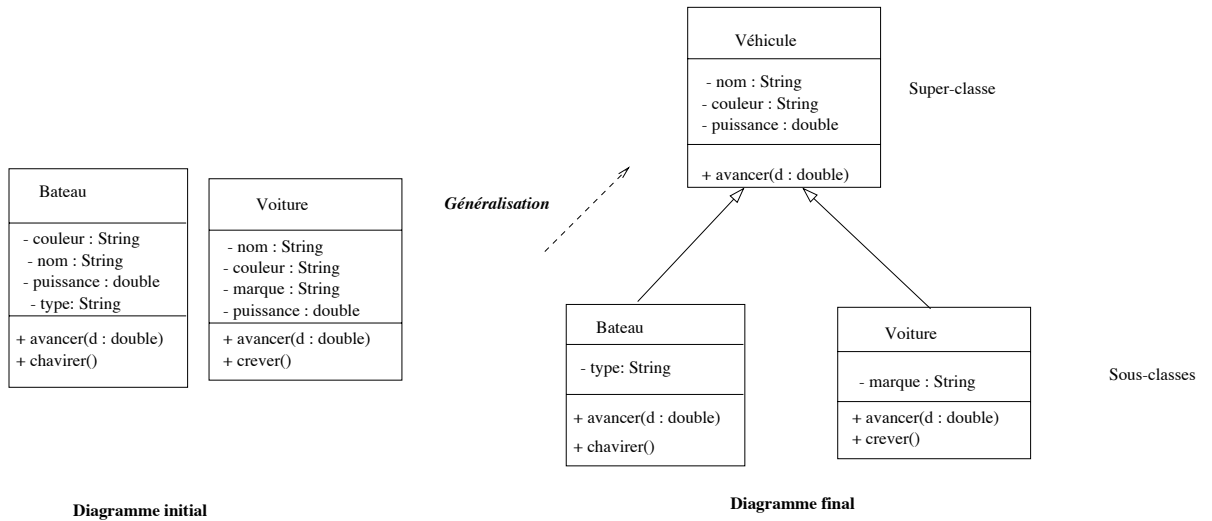


FIG. 7.1 – Une généralisation

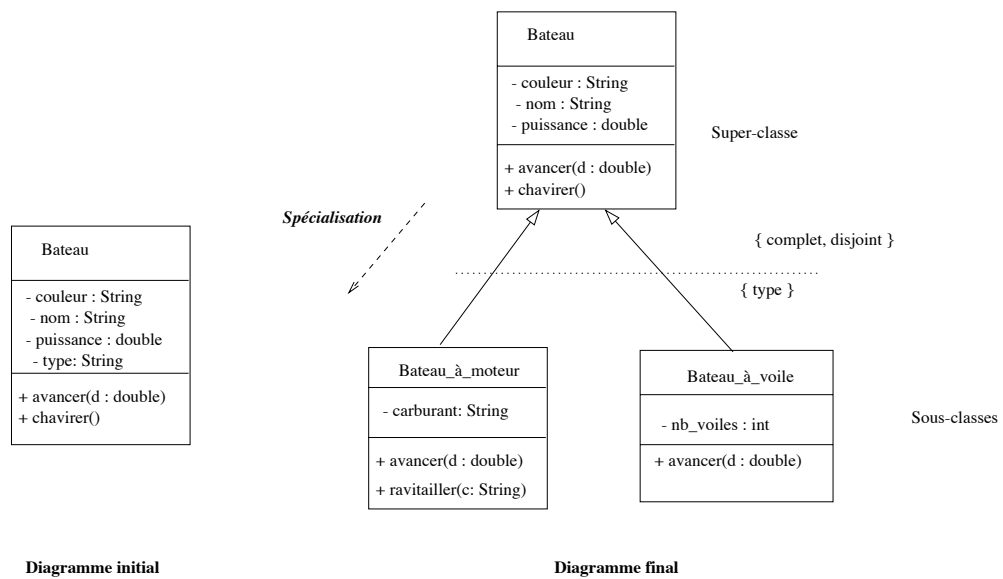


FIG. 7.2 – Une spécialisation

**Discriminant** Les discriminants sont tout simplement les critères utilisés pour classer les objets dans des sous-classes. Ils étiquettent ainsi les relations de spécialisation et doivent correspondre à une classe du modèle. Dans l'exemple de la figure 7.3, deux critères différents de classification sont utilisés : **TypeContrat** partage les employés en salariés et vacataires, tandis que **RetraiteComplémentaire** partage les employés en cotisants et non cotisants. Notez qu'UML autorise qu'un objet soit classé à la fois comme cotisant et comme vacataire : on parle alors de multi-instanciation.

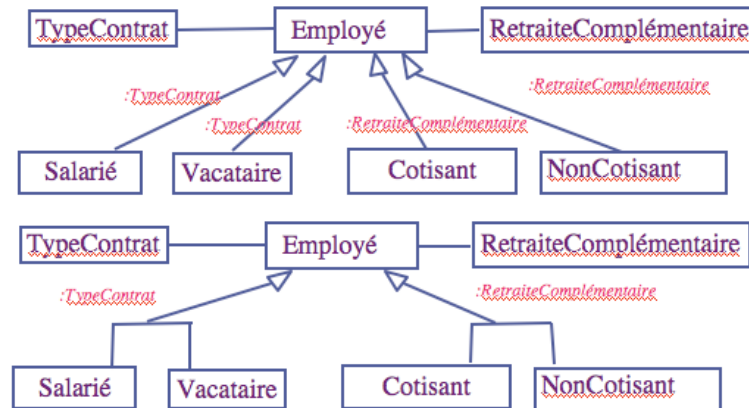


FIG. 7.3 – Discriminants

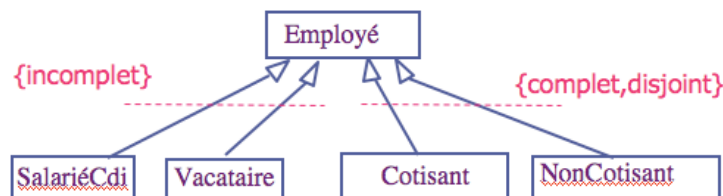


FIG. 7.4 – Contraintes

**Contraintes** Les contraintes décrivent la relation entre un ensemble de sous-classes et leur super-classe en considérant le point de vue des extensions (ensemble d'instances des classes).

Il en existe quatre, dont trois sont illustrées figure 7.4 et une figure 7.5 :

- **incomplete (incomplet)** : l'union des extensions des sous-classes est strictement incluse dans l'extension de la super-classe ; par exemple il existe des employés qui ne sont ni salariés, ni vacataires ;
- **complete (complet)** : l'union des extensions des sous-classes est égale à l'extension de la super-classe ; par exemple tout employé est cotisant ou non cotisant ;
- **disjoint** : les extensions des sous-classes sont d'intersection vide ; par exemple aucun employé n'est cotisant et non cotisant ;
- **overlapping (chevauchement)** : les extensions des sous-classes se rencontrent, par exemple si on avait spécialisé une classe **Vehicule** en **VehiculeTerrestre** et **VehiculeAquatique**, certains véhicules se trouveraient dans les extensions des deux sous-classes.

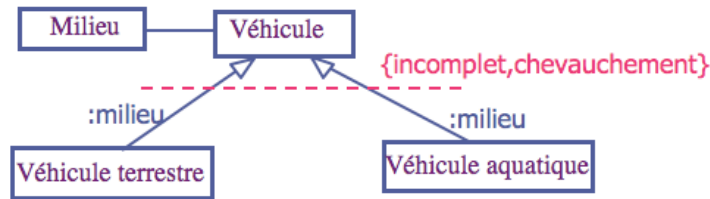


FIG. 7.5 – Contraintes

## 7.2 Hiérarchie des classes et héritage dans Java

Pour les langages à objets, le programmeur définit des hiérarchies de classes provenant d'une conception dans laquelle il a utilisé de la généralisation/spécialisation, on dit le plus souvent la sous-classe hérite de la super-classe (ou qu'elle étend la super-classe ou qu'elle dérive de la super-classe).

Un des avantages des langages à objets est que le code est réutilisable. Il est commode de construire de gros projets informatiques en étendant des classes déjà testées. Le code produit devrait être plus lisible et plus robuste car on peut contrôler plus facilement son évolution, au fur et à mesure de l'avancement du projet. En effet, grâce à la factorisation introduite par la spécialisation-généralisation, on peut modifier une ou plusieurs classes sans avoir à les réécrire complètement (par exemple en modifiant le code de leur super-classe).

Du point de vue des objets (instances de classes), pour Java, une instance d'une sous-classe possède la partie structurelle définie dans les superclasses de sa classe plus la partie structurelle définie dans celle-ci. Au niveau du comportement, les objets d'une sous-classe héritent du comportement défini dans les superclasses, avec quelques possibilités de variations, comme nous le verrons plus tard.

Au niveau de l'exécution, les langages à objets présentent un mécanisme dénommé liaison (ou ligature) dynamique qui consiste à résoudre l'envoi de message sur les objets (instances), c'est-à-dire à trouver et exécuter le code adéquat correspondant au message.

En Java :

- toutes les classes dérivent de la classe `Object`, qui est la racine de toute hiérarchie ;
- une classe ne peut avoir qu'une seule super-classe directe ; on parle d'héritage simple ;  
Il existe des langages à objets, tels que C++ ou Eiffel qui autorisent l'héritage multiple ;
- le mot-clef permettant de définir la généralisation/spécialisation entre classes est `extends`.

Pour la suite du chapitre nous prenons une hiérarchie de classes représentée dans le diagramme 7.6. Discriminants et contraintes ne se traduisent pas directement en Java.

Voici le code permettant de créer la structure de la hiérarchie.

```

package Prog.SPEC;
import java.io.*;
public class Personne{
.....} // fin de la classe Personne
.....
public class Etudiant extends Personne {
.....
} // fin de la classe Etudiant
  
```

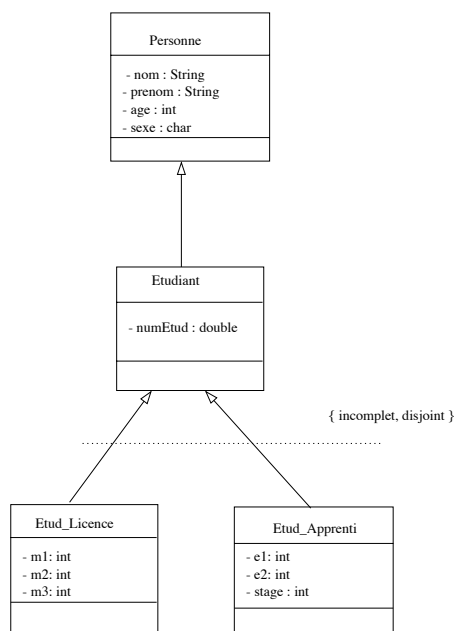


FIG. 7.6 – Une hiérarchie de classes

```

.....
public class Etud_Licence extends Etudiant {
    .....
}    // fin de la classe Etud_Licence
.....
public class Etud_Apprenti extends Etudiant {
    .....
}    // fin de la classe Etud_Apprenti
  
```

Tous les étudiants sont des personnes, mais on peut les spécialiser en étudiants de licence ou étudiants apprentis. Les attributs définis dans la classe **Personne** sont nom, prénom, âge, sexe. Bien sûr, on définit pour cette classe les constructeurs, les accesseurs, et une opération ou méthode **incrémenterAge()** (cette méthode vieillit d'un an).

Le seul attribut défini dans la classe **Etudiant** est numEtud. La classe **Etudiant** définit des opérations de calculs de moyenne, d'admission, de mention.

Les étudiants de licence (classe **Etud\_Licence** passent 3 modules (M1, M2, M3) de coefficient 1. Ils sont admis si leur moyenne générale est  $\geq 10$  et conservent les modules obtenus s'ils ne sont pas admis.

Les étudiants apprentis (classe **Etud\_Apprenti**) passent 2 modules (E1, E2) de coefficient 1 et un stage de coefficient 2. Ils sont admis si leur moyenne est  $\geq 10$  et si leur note de stage est  $> 8$ .

Une instance d'étudiant apprenti (ou une instance d'étudiant licence) a la possibilité d'utiliser des méthodes définies dans **Personne** :

```

//exemple de code pouvant figurer dans la méthode main d'une classe application
Etud_Apprenti a = new Etud_Apprenti ("Einstein", "Albert", 22, 'M', 123, 8, 8, 10);
  
```

```
// L'age de cet apprenti est 22 ans
a.incrementsAge();
System.out.println(a.getName()+"est âgé de " + a.getAge()+" ans");
```

A l'exécution, on obtient :

Einstein Albert est âgé de 23 ans

### 7.3 Redéfinition de méthodes - Surcharge - Masquage

Une opération (ou une méthode) est déclarée dans une classe, possède un nom et une liste de paramètres, on parle de sa signature (nom de classe + nom de l'opération + liste des types des paramètres et type de retour). A priori, dans une hiérarchie de classes, chaque classe ayant un nom différent, deux opérations de même signature sont interdites. Mais le programmeur peut :

- au sein d'une même classe donner le même nom à une opération si la liste des paramètres est différente (surcharge statique),
- dans des classes distinctes, donner le même nom et la même liste de paramètres. Lorsqu'une méthode d'une sous-classe a même nom et même liste de paramètres qu'une méthode d'une super-classe, on dit que la méthode de la sous-classe redéfinit la méthode de la super-classe.

Lorsque l'on code une sous-classe, on a la possibilité de redéfinir une ou plusieurs méthodes de la super-classe, simplement en redéfinissant une méthode qui a le même nom et les mêmes paramètres que celle de la super-classe.

*Remarque :* La méthode de la super-classe sera encore accessible mais en la préfixant par le mot-clé **super**, pseudo-variable définie par le système et qui représente l'objet courant vu comme étant du type de la super-classe.

```
// Classe Etudiant
.....
public class Etudiant extends Personne{
.....
public boolean admis()
    {return (moyenne() >= 10;)}
}

// Classe Etud_Apprenti

public class Etud_Apprenti extends Etudiant {
.....
public boolean admis()
    {
        return (super.admis() && getnoteSt() > 8);
        // on fait appel ici à l'opération admis() définie dans Etudiant
        // on aurait pu aussi définir admis dans Etud_Apprenti par
        //return (moyenne() >= 10 && getnoteSt() > 8;)
    }
}
```



*Remarque :* La forme de l'opération `admis` qui utilise `super` est plus réutilisable, en effet si les conditions d'admission générale sur tous les étudiants changent, le code de l'opération `admis` dans la classe `Etudiant` sera modifié, entraînant la modification automatique de toutes les opérations qui font appel à elle via `super.admis()`.

## 7.4 Les constructeurs

Dès qu'un constructeur a été défini dans une classe de la hiérarchie de classes, il est nécessaire de définir des constructeurs lui correspondant dans toutes les sous-classes. D'autre part, il est commode et intéressant de faire appel aux constructeurs des super-classes que l'on invoque par le mot clé `super` suivi de ses arguments entre parenthèses. L'invocation du constructeur de la super-classe doit se faire obligatoirement à la première ligne.

```
// la classe Personne

public Class Personne{
    .....
    // constructeur par défaut
    public Personne() {
        nom = "";prenom = "";age = 0;sexe = 'F';}
    // autre constructeur
    public Personne(String n, String p, int a, char s) {
        nom = n;prenom = p;age = a;sexe = s;}
    ....
}

// la classe Etudiant
public Class Etudiant extends Personne{
    .....
    // constructeur par défaut
    public Etudiant()
    {
        // par défaut appelle super();
        // qui est toujours la première instruction du constructeur
        numEtu=-1;
    }
    // autre constructeur
    public Etudiant(String n, String p, int a, char s, double n)
    {
        super(n, p, a, s); // appel au 2ième constructeur
                           // de la super-classe Personne;
        numEtu=n;
    }
    ....
}

// la classe Etu_Apprenti
public Class Etu_Apprenti extends Etudiant{
    // constructeur par défaut
    public Etu_Apprenti ()
```

```
    {
        // Par défaut, super() est appelée.
        E1=-1; E2=-1; stage=-1;
    }
// autre constructeur
public Etu_Apprenti (String n, String p, int a, char s, double n, int e1, int e2, int s
    {
        super(n, p, a, s, n); // appel au 2ième constructeur de Etudiant
        E1 = e1; E2 = e2; stage = st;
    ....
}
```

*Remarque* : L'utilisation de **super** dans les constructeurs permet de mieux gérer l'évolution du code, une modification d'un constructeur d'une super-classe, provoquant automatiquement la modification des constructeurs des sous-classes faisant appel à lui par **super**.

## 7.5 Protection/contrôle d'accès statique

Nous avons déjà introduit les directives de protection, nous pouvons donc compléter ici leur description par quelques principes généraux :

- **public** : la méthode (ou l'attribut) est accessible par tous et de n'importe où.
- **protected** : la méthode (ou l'attribut) n'est accessible que par les classes du même package et par les sous-classes (même dans un autre paquetage). *Remarque* : pour le contrôle d'accès **protected**, il semblait logique de n'autoriser l'accès qu'à la classe concernée et à ses sous classes. Mais SUN en a décidé autrement.
- **private** : la méthode (ou l'attribut) est accessible uniquement par les méthodes de la classe. *Remarque* : Cependant les instances d'une même classe ont accès aux méthodes et attributs privés des autres instances de cette classe.

## 7.6 Classes et méthodes abstraites

Dans une hiérarchie de classes, plus une classe occupe un rang élevé, plus elle est générale donc plus elle est abstraite. On peut donc envisager de l'abstraire complètement en lui ôtant d'une part le rôle de génitrice (elle ne sera pas autorisée à créer des instances), et en lui permettant d'autre part de factoriser structures et comportements (sans savoir exactement comment les réaliser) uniquement pour rendre service à sa sous-hiérarchie.

Une méthode *abstraite* est une méthode déclarée avec le mot-clef **abstract** et ne possède pas de corps (pas de définition de code). Par exemple, le calcul de la moyenne d'un étudiant ne peut pas être décrit au niveau de cette classe.

Néanmoins, on sait que les étudiants de licence et les étudiants apprentis doivent être capables de calculer leur moyenne. La méthode **moyenne** doit être déclarée abstraite au niveau de la classe **Etudiant**. Par contre, il faudra obligatoirement définir le corps de la méthode **moyenne** dans les sous-classes de la classe **Etudiant**.

Si une classe contient au moins une méthode abstraite, alors il faut déclarer cette classe abstraite, mais on peut aussi définir des classes abstraites (parce qu'on ne veut pas qu'elles engendrent des objets) sans aucune méthode abstraite.

```
.....
// la classe est déclarée abstraite
```

```
public abstract class Etudiant extends Personne{
.....
// la méthode est déclarée abstraite
public abstract double moyenne();
}
```

*Remarques* : Dans le formalisme UML les classes et méthodes abstraites ont leur nom en italique ou bien assorti du mot-clef **abstract**.

Une classe abstraite peut être sous-classe d’une classe concrète (ici **Etudiant** est abstraite et sous-classe de **Personne** concrète).

L’intérêt de définir une méthode abstraite est double, il permet au développeur de ne pas oublier de redéfinir une méthode qui a été définie **abstract** au niveau d’une des super-classes, il permet de mettre en œuvre le processus de polymorphisme.

## 7.7 Le polymorphisme

### 7.7.1 Coercition, Transformation de type ou Casting

Une référence sur un objet d’une sous-classe peut toujours être implicitement convertie en une référence sur un objet de la super-classe.

```
Personne p;
Etud_Licence e=new Etud_Licence();
p=e;
// tout étudiant de licence est un étudiant et a fortiori une personne
```

L’opération inverse (cast-down) est possible de manière explicite (mais avec précaution). Elle est devenue beaucoup moins importante en Java depuis l’introduction de la généricité paramétrique. La transformation de type est par exemple utilisée si l’on récupère des objets dans **Vector<Object>** ou dans **Vector**. Dans ce contexte là, le casting permet de préciser, si nécessaire, au compilateur, la classe présumée de l’objet récupéré.

Cette opération de transformation de type explicite peut être utilisée (indépendamment des hiérarchies de classes) sur les types primitifs.

```
double x = 9.9743;
int xEntier = (int) x; // alors xEntier=9
```

Mais cette opération devient plus fondamentale dans le cas de l’utilisation de hiérarchies de classes. Complétons l’exemple précédent par la notion de promotion.

```
//la classe Promotion
.....
public class Promotion {
private Vector listeEtudiants = new Vector();
.....
public double moyenneGenerale() {
    double somme = 0;
    for (int i=0;i<listeEtudiants.size();i++)
    {
        //le casting a lieu ici :
```

```
        Etudiant etud = (Etudiant)(listeEtudiants.get(i));
        somme = somme + etud.moyenne();
    }
    if (listeEtudiants.size()>0)
        return (somme / listeEtudiants.size());
    else return 0;
}
```

Les éléments d'un Vector sont des instances d'Object. La transformation de type `(Etudiant)(listeEtudiants.get(i))` permet de préciser à `listeEtudiants.elementAt(i)` que l'on va s'adresser à lui non pas en tant qu'instance d'Object mais en tant qu'instance d'Etudiant, car l'appel de la méthode `moyenne()` sur une instance de la classe Object entraîne une erreur de compilation, cette classe ne définissant pas de calcul de moyenne. A l'exécution, il est cependant notable de remarquer que chaque objet extrait du vecteur `listeEtudiants` exécutera le code de la méthode `moyenne()` afférent à sa classe mère, c'est-à-dire que les étudiants de licence exécuteront le code de la classe `Etud.Licence` et les étudiants apprentis exécuteront le code de la classe `Etud.Apprenti`. L'appel `etud.moyenne()` est donc polymorphe (puisqu'il recouvre diverses formes de calcul).

```
.....
public class Personne{
    .....
    public void f1() {
        System.out.println("Methode f1(), de la classe Personne");}
}
.....
public class Etud_Apprenti extends Etudiant { //qui est, elle-même, sous
classe de Personne
//Redéfinition de f1()
    public void f1() {
        System.out.println("Methode f1(), de la classe Etud_Apprenti");}

//Nouvelle methode f2 introduite dans Etud_Apprenti (pas de redéfinition)
    public void f2() {
        System.out.println("Methode f2(), de la classe Etud_Apprenti");}
}
.....
// la classe Appli_Cursus
public class Appli_Cursus {

    public static void main(String args[]) {
        Personne p;
        Etud_Apprenti a1 = new Apprenti(); // constructeur par défaut

        p = a1;
        // OK - transformation implicite autorisée car
        // un Etud_Apprenti est une personne !

        p.f1();
        // affiche : Methode f1(), de la classe Etud_Apprenti (eh oui ...)
```

```

    p.f2();
    // Attention  ERREUR à la compilation: methode f2() not found in Personne

    Etud_Apprenti a2 = (Etud_Apprenti) p;
    // OK - cast down autorisé à la compilation

    a2.f1();
    // affiche : Methode f1(), de la classe Etud_Apprenti
    // en fait p référençait un Etud_Apprenti a1 et a2 référence donc
aussi a1
    a2.f2();
    // affiche : Methode f2(), de la classe Etud_Apprenti

    p = new Personne();
    p.f1();
    // affiche : Methode f1(), de la classe Personne
    Etud_Apprenti a3 = p;
    // ERREUR de compilation
    // incompatible type found: Personne, required: Apprenti
    // une Personne n'est pas un  Etud_Apprenti

    Etud_Apprenti a3 = (Etud_Apprenti) p;
    // le compilateur accepte
    a3.f1();
    // le compilateur accepte
    // MAIS ERREUR d'exécution java.lang.ClassCastException: Personne
}
}

```

A l'exécution (une fois supprimé les erreurs), on obtient :

```

Methode f1(), de la classe Etud_Apprenti
Methode f1(), de la classe Etud_Apprenti
Methode f2(), de la classe Etud_Apprenti
Methode f1(), de la classe Personne

```

Une transformation de type d'objet ne peut s'appliquer qu'à des objets instances de classes d'un même chemin d'héritage au sein d'une hiérarchie de classe.

```
Etud_Apprenti a=new Etud_Apprenti();
```

```
Voiture v = (Voiture) a    //ne marchera pas !
```

Pour vérifier qu'une instance appartient bien à une classe précise d'une hiérarchie, on peut utiliser l'opérateur `instanceof`, mais cela devrait être absolument exceptionnel, tout comme les cast-down depuis Java 1.5. La nécessité d'utiliser `instanceof` révèle souvent une mauvaise conception.

Par exemple, dans la classe `Promotion`, ajoutons une méthode qui incrémente les notes E1 des étudiants apprentis :

\\ dans la classe Promotion

```
public void incrNoteE1Apprentis() {
    for (int i=0;i<listeEtudiants.size();i++)
    {
        if (listeEtudiant.elementAt(i) instanceof Etud_Apprenti)
        {Etud_Apprenti a = (Etud_Apprenti) listeEtudiant.elementAt(i);
         a.setE1(a.getE1()+1);
        // setE1 est une méthode qui n'existe que dans la classe Etud_Apprenti
        }
    }
}
```

### 7.7.2 Le polymorphisme

Le fait que l'on puisse définir dans plusieurs classes des méthodes de même nom revient donc à désigner plusieurs formes de traitement derrière ces méthodes. Le code de la méthode qui sera réellement exécuté n'est donc pas figé, un appel de message autorisé à la compilation donnera des résultats différents à l'exécution car le langage retrouvera selon l'objet et la classe à laquelle il doit son existence le code à exécuter (nous avons vu que l'on parle de **liaison dynamique**). La capacité pour un message de correspondre à plusieurs formes de traitements est appelé **polymorphisme**

Quelques exemples pour fixer les idées.

//une autre hiérarchie de classes

```
public abstract class Felin {
    .....
    public abstract String pousseSonCri();
    .....
}

public class Lion extends Felin {
    .....
    public String pousseSonCri() {return "rouaaaaah";}
    public String toString() {return "lion";}
    .....
}

public class Chat extends Felin {
    .....
    public String pousseSonCri() {return "miaou";}
    public String toString() {return "chat";}
    .....
}

public class AppliCriDeLaBete {
    public static void main(String args[]) {
        Felin tab[] = new Felin[2];
```

```
        tab[0] = new Lion();
        tab[1] = new Chat();
        for( int i=0; i<2; i++)
            System.out.println("Le cri du "+tab[i]+" est : "+ tab[i].pousseSonCri());
    }
}
```

Remarquer ici, que le programme `main` utilise un tableau de `Felin`, que le casting implicite est utilisé, et que `tab[i]` étant un `Felin` pour le compilateur et la méthode `pousseSonCri` étant définie dans `Felin` (sous forme abstraite), il n'y a pas d'erreur de compilation et à l'exécution, on obtient :

```
Le cri du lion est : rouaaaaah
Le cri du chat est : miaou
```

Relisons et reprenons avec de la généricité paramétrique l'exemple de la promotion d'étudiants.

```
.....
public class Promotion {
    private Vector<Etudiant> listeEtudiants = new Vector<Etudiant>();
    .....
    public double moyenneGenerale() {
        double somme = 0;
        for (int i=0;i<listeEtudiants.size();i++)
            {Etudiant etud = listeEtudiants.get(i);
// Le polymorphisme a lieu ici: le calcul de la méthode moyenne() sera
// différent si etud est une instance de Etud_Licence ou de Etud_Apprenti.
            somme = somme + etud.moyenne();
            }
        if (listeEtudiants.size()>0)
            return (somme / listeEtudiants.size());
        else return 0;
    }

    public String nomDesEtudiants() {
        String listeNom = "";
// Le polymorphisme a lieu ici: la méthode toString() est appelée sur tous les
// objets du vecteur (listeEtudiants).
        for (int i=0;i<listeEtudiants.size();i++)
            listeNom = listeNom +listeEtudiants.get(i).toString() + " \n";
// Object définit toString()
        return listeNom;
    }
} // fin classe Promotion

public class AppliPromo {
    public static void main(String args[]) {
        Promotion promo = new Promotion (2000);
```

```
Etud_Licence e1 = new Etud_Licence("Cesar", "Julio", 26, 'M', 127, 14, 12, 10);
promo.inscrit (e1);
Etud_Licence e2 = new Etud_Licence("Curie", "Marie", 22, 'F', 124, 8, 17, 20);
promo.inscrit (e2);
Etud_Apprenti a1 = new Etud_Apprenti ("Bol", "Gemoï", 22, 'M', 624, 8, 8, 10);
promo.inscrit (a1);
Etud_apprenti a2 = new Etud_Apprenti ("Einstein", "Albert", 22, 'M', 123, 13, 17, 1);
promo.inscrit (a2);

System.out.println("La moyenne de la promotion est : " + promo.moyenneGenerale() );
promo.incrementeNoteE1DesApprentis();
System.out.println("La nouvelle moyenne de la promotion est : "
+ promo.moyenneGenerale() );
System.out.println("Les étudiants de la promotion sont : "
+ promo.nomDesEtudiants() );
    }
}
```

À l'exécution, on obtient :

```
La moyenne de la promotion est : 13
La nouvelle moyenne de la promotion est : 13.5
Les étudiants de la promotion sont :
Cesar Julio (Etud_Licence)
Curie Marie (Etud_Licence)
Bol Gemoï (Etud_Apprenti)
Einstein Albert (Etud_Apprenti)
```



## Chapitre 8

# Les interfaces de Java

### 8.1 Définition et objectif

Une « interface » est par définition une zone de contact, dans le cas qui nous intéresse, il s'agit d'interfaces entre des parties d'un logiciel et plus précisément une interface est comme une façade pour une classe, décrivant ce que la classe peut offrir comme services à un programme, ou encore l'un des rôles qu'elle peut jouer.

En Java, une interface se présente comme un ensemble de :

- signatures de méthodes publiques (méthodes d'instances abstraites),
- variables de classes constantes et publiques (variables statiques).

Les interfaces permettent d'améliorer le code pour plusieurs raisons que nous détaillerons après avoir vu les éléments syntaxiques qui les constituent et les relient aux classes :

- on décrit des types de manière plus abstraite qu'avec les classes et par conséquent ces types sont plus réutilisables ;
- c'est une technique pour masquer l'implémentation puisqu'on découple la partie publique d'un type de son implémentation ;
- on peut écrire du code plus générique (plus général), au sens où il est décrit sur ces types plus abstraits ;
- les relations de spécialisation entre les interfaces (d'une part) et entre les classes et les interfaces (d'autre part) relèvent de la spécialisation multiple, ce qui facilite l'organisation des types d'un programme.

### 8.2 Eléments syntaxiques

#### 8.2.1 Définition d'une interface

Le premier exemple qui vous est proposé est une interface simplifiée destinée à représenter les quadrilatères. On déclare avec cette interface qu'un objet qui prétendrait être un quadrilatère doit être capable de répondre au message `perimetre`. De plus les quadrilatères sont décrits par un nombre de côtés. Vous pouvez noter l'utilisation des mots-clefs `public`, `abstract`, `static` et `final` qui indiquent que toute propriété d'une interface est publique, que les méthodes sont forcément abstraites et que les variables sont des variables de classe constantes.

```
public interface Iquadrilatere
{
    public static final int nbCotes = 4;
```

```
    public abstract float perimetre();  
}
```

Les mots-clefs peuvent être omis, puisque toutes ces caractéristiques sur les propriétés sont obligatoires, comme c'est illustré ci-dessous.

```
public interface Iquadrilatere  
{  
    int nbCotes = 4;  
    float perimetre();  
}
```

### 8.2.2 Spécialisation d'une interface

Les interfaces sont des types qui peuvent être organisés par spécialisation/généralisation comme les classes. Par exemple, une interface `Irectangle` vient spécialiser l'interface `Iquadrilatere`. On peut noter le même mot-clef `extends` qui est utilisé pour déclarer la spécialisation entre interfaces comme entre classes.

```
interface Irectangle extends Iquadrilatere  
{  
    float angle = 90;  
    float angle();  
    float largeur();  
    float hauteur();  
}
```

### 8.2.3 Lien classe/interface

Si nous désirons à présent créer des rectangles, il faut créer une classe *concrète* pour les représenter, c'est-à-dire une classe qui implémente les différentes méthodes qui ont été proposées dans l'interface `Irectangle` ou ses généralisations (ici l'interface `Iquadrilatere`).

Notez que le mot-clef qui relie la classe et l'interface est cette fois `implements`. Il indique bien que la classe n'est pas une spécialisation quelconque de l'interface, mais une implémentation.

```
public class Rectangle implements Irectangle  
{  
    private float largeur, hauteur;  
    public Rectangle(){}  
    public Rectangle(float l, float h){largeur=l; hauteur=h;}  
  
    public float perimetre(){return 2*largeur()+2*hauteur();}  
    public float angle(){return Irectangle.angle;}  
    public float largeur(){return largeur;}  
    public float hauteur(){return hauteur;}  
}
```

Le fait d'avoir déclaré la classe comme implémentant l'interface la contraint à implémenter toutes les méthodes (si certaines ne sont pas implémentées, la classe doit être déclarée abstraite).

### 8.3 Description de comportements génériques

Le code de certaines méthodes assez générales peut être écrit, même si tout le code nécessaire pour avoir le comportement complet n'existe pas encore. Ci-dessous une classe `StockRectangle` propose une méthode `sommePerimetres`, capable de fonctionner sur un vecteur d'objets de type `Irectangle`.

```
class StockRectangle
{
    Vector<Irectangle> listeRectangle = new Vector<Irectangle>();

    public void ajoute(Irectangle r){listeRectangle.add(r);}

    public float sommePerimetres()
    {
        float sp=0;
        for (int i=0; i<listeRectangle.size(); i++)
            {sp+=listeRectangle.get(i).perimetre();}
        return sp;
    }
}
```

La méthode `sommePerimetres` fonctionne bien entendu pour des objets de la classe `Rectangle` définie ci-dessus, mais également pour des objets d'une classe `RectangleTab` qui serait une autre implémentation de l'interface `Irectangle`, utilisant un tableau pour stocker la largeur et la hauteur du rectangle.

```
class RectangleTab implements Irectangle
{
    private float tab[]=new float[2];
    public RectangleTab(){}
    public RectangleTab(float l, float h){tab[0]=l; tab[1]=h;}

    public float perimetre(){return 2*largeur()+2*hauteur();}
    public float angle(){return Irectangle.angle;}
    public float largeur(){return tab[0];}
    public float hauteur(){return tab[1];}
}
```

Ce principe est général à l'approche par objets : vous l'avez déjà appliqué lorsque vous avez invoqué dans une méthode d'une certaine classe une méthode abstraite pour cette même classe. Simplement il atteint ici sa plus grande utilité car dans tous les cas, les méthodes invoquées sur l'interface sont des méthodes abstraites, donc écrites systématiquement indépendamment de la partie de code générique.

### 8.4 Description plus abstraite des types

L'une des utilisations les plus classiques des interfaces est la représentation des types abstraits de données. Nous l'illustrons avec l'exemple du type `Pile` qui a l'avantage de la concision et de la simplicité. Nous en profitons, histoire de mettre un peu de piment dans

l'affaire, pour introduire la syntaxe permettant de définir un type générique, comme le fait Java pour les collections en général. Vous découvrirez que c'est très simple!

Voici pour commencer une interface décrivant les opérations disponibles sur les piles. Notez la partie de la déclaration contenant `<T>` qui exprime ce que l'on appelle le paramètre de généricité, ici c'est un type (T) qui sera passé en paramètre à un autre type, en l'occurrence le type `Pile`.

```
public interface Pile<T>
{
    void empile(T t);
    void depile();
    T sommet();
    boolean estVide();
}
```

Puis voici une classe implémentant l'interface `Pile` à l'aide d'un vecteur pour stocker les éléments.

```
public class PileVector<T> implements Pile<T>
{
    Vector<T> v=new Vector<T>();
    public PileVector(){}
    public void empile(T t){v.add(t);}
    public void depile(){v.remove(v.size()-1);}
    public T sommet(){return v.get(v.size()-1);}
    public boolean estVide(){return v.isEmpty();}
    public String toString(){return "Pile "+v.toString();}
}
```

Le programme `main` montre comment on crée une pile en passant un paramètre de type réel (`Integer`).

```
public class programmePile
{
    public static void main(String[] a)
    {
        Pile<Integer> p1 = new PileVector<Integer>();
        p1.empile(7);
        p1.empile(5);
        p1.empile(4);
        System.out.println(p1);
        p1.depile();
        System.out.println(p1);
    }
}
```

Quel est l'intérêt d'avoir fait une interface dans ce cas? Imaginons un programme utilisateur qui ait besoin d'une pile pour effectuer une descente en profondeur dans un graphe, l'évaluation d'une expression arithmétique, etc. Ce programme peut être écrit en utilisant l'interface `Pile` comme elle est utilisée pour déclarer la pile `p1` dans le programme

précédent. Il n'y a aucun moyen de tricher et de s'appuyer sur une quelconque hypothèse d'implémentation de la pile pour écrire ce programme utilisateur. Seule l'instruction de création de la pile concrète (comme `new PileVector<Integer>()`) devra être modifiée si on décide pour des raisons d'efficacité pour certaines opérations (comme l'ajout ou le retrait à une position interne) de changer de mode d'implémentation de la pile, et que l'on préfère une `PileListe`, implémentée à l'aide d'une liste chaînée (à faire en exercice ...).

## 8.5 Spécialisation multiple

Contrairement aux classes qui doivent être organisées dans une hiérarchie de spécialisation simple (une arborescence dans laquelle une classe ne peut avoir qu'une seule super-classe directe), les interfaces peuvent être structurées dans une hiérarchie de spécialisation multiple (un graphe orienté sans circuits quelconque dans lequel une classe peut avoir plusieurs super-classes directes).

Cette spécialisation multiple qui est moins contraignante est également plus naturelle pour exprimer des relations de classifications quelconques. Nous illustrons ce point en approfondissant l'exemple des polygones.

Examinons pour commencer une hiérarchie de classes représentant les polygones. Si elle est écrite en Java, elle ne contient que de l'héritage simple.

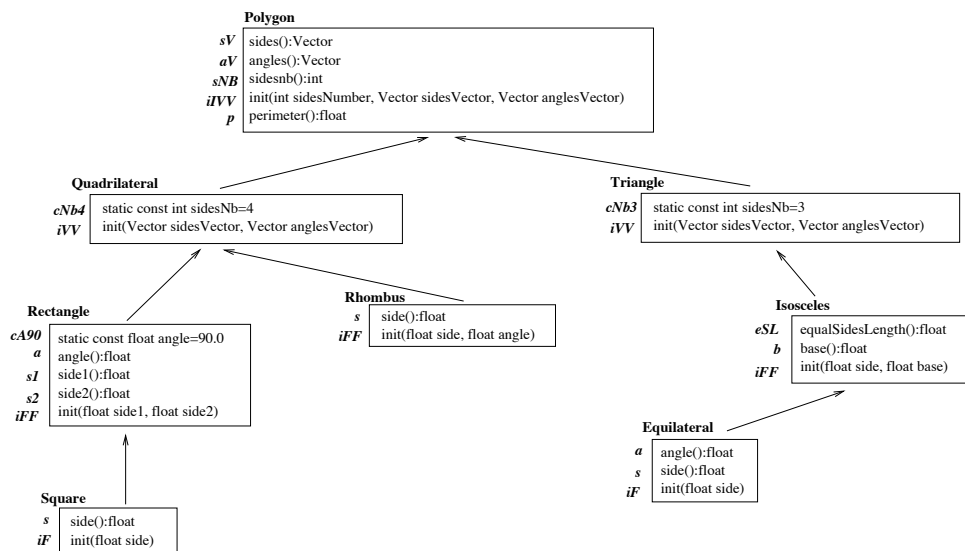


FIG. 8.1 – Une hiérarchie de classes représentant certains polygones convexes

Cette hiérarchie contient quelques défauts d'organisation.

- Les carrés ne peuvent être à la fois des rectangles et des losanges, comme c'est pourtant le cas dans le domaine des mathématiques.
- Toutes les propriétés ne sont pas factorisées, certaines sont redondantes, comme `side()` ou `angle()`.

En Java, on ne pourra guère faire mieux avec les classes, mais les interfaces offrent un bon moyen de présenter une organisation plus intéressante, notamment avec une absence de redondance des propriétés (attributs ou méthodes), la mise à jour de nouveaux concepts intéressants comme les polygones réguliers et la possibilité de représenter tous les liens de spécialisation manquants, comme entre les carrés et les losanges.

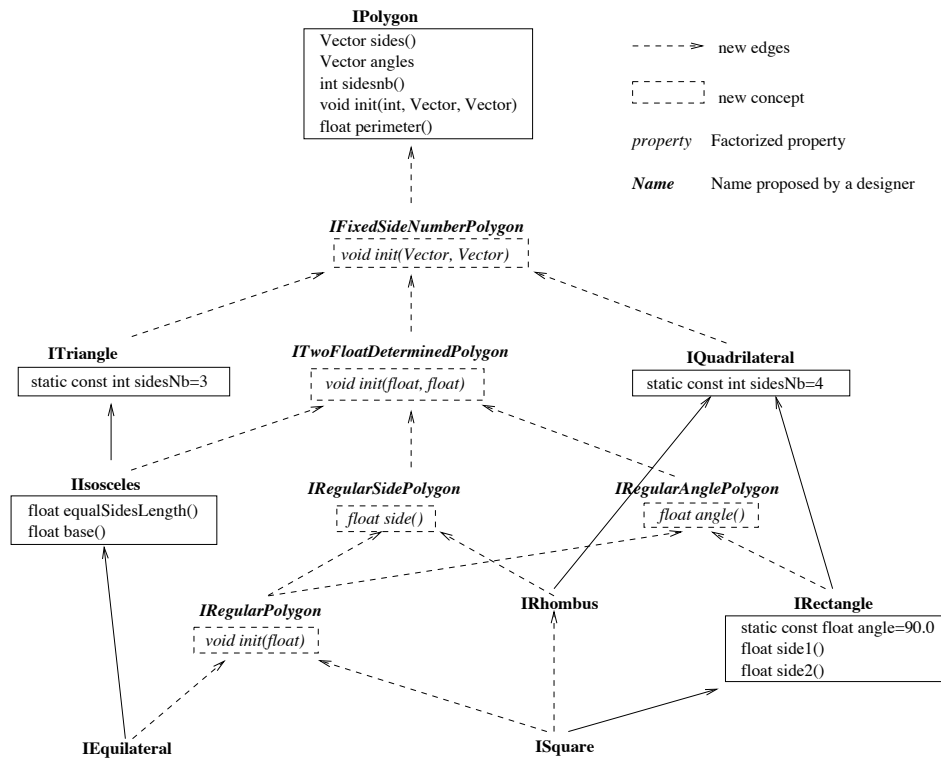


FIG. 8.2 – Une classification possible des polygones avec des interfaces

Et enfin on peut associer les classes abstraites ou concrètes avec les interfaces qu'elles implémentent. On dispose ainsi d'une structuration élégante des concepts d'un domaine grâce aux interfaces et de leur contre-partie opérationnelle avec les classes.

## 8.6 Quelques interfaces importantes de l'API Java

Vous découvrirez de nombreuses interfaces dans l'API Java car c'est une technique couramment utilisée pour écrire des programmes Java d'un assez haut niveau de généralité.

### 8.6.1 Interfaces marqueurs

Certaines interfaces sont vides d'opérations et de constantes de classes, ce sont simplement des marqueurs des classes, précisant leur sémantique et indiquant dans quel contexte leurs objets peuvent être utilisés. Implémenter ces interfaces marqueurs n'est pas toujours suffisant pour obtenir le comportement attendu, mais c'est nécessaire.

**cloneable** Lorsqu'une classe implémente cette interface, ses objets peuvent être clonés : on peut leur appliquer une méthode `clone`, `protected` dans la classe `Object`, et qui doit être redéfinie `public` dans la classe concernée.

**serializable** L'interface `serializable` permet d'indiquer que l'on autorise les objets d'une classe à être « sérialisés » c'est-à-dire écrits dans un flux de données. Des méthodes `readObject` et `writeObject` doivent alors être réécrites si on désire une sérialisation particulière pour les objets de la classe concernée.

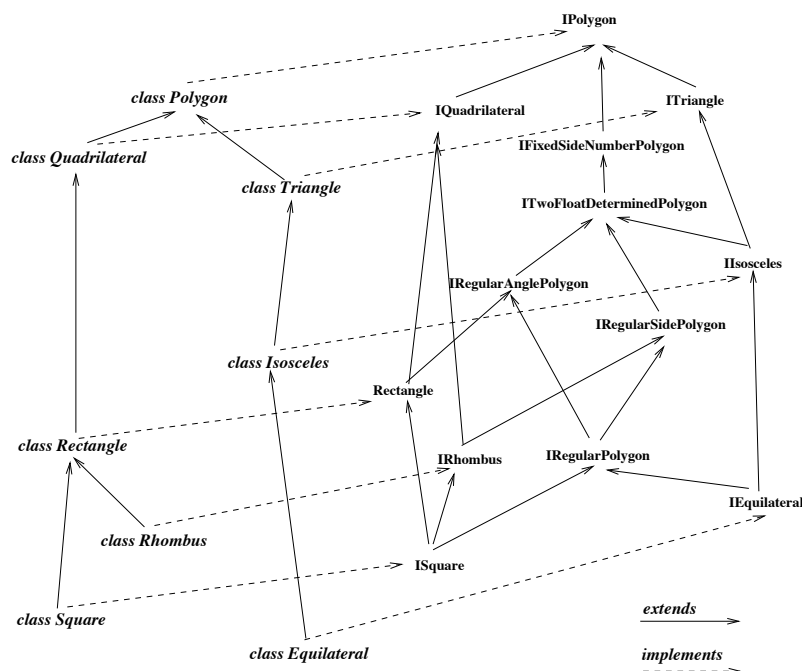


FIG. 8.3 – Une hiérarchie associant les classes et les interfaces

### 8.6.2 Comparaison d'objets et tris

L'API définit une interface `Comparable` dont le code est le suivant.

```
public interface Comparable<T>
{
    int compareTo(T o);
}
```

`compareTo` retourne -1, 0, ou 1 suivant si l'objet receveur est plus petit, égal ou plus grand que le paramètre. Cette opération est notamment utilisée pour les opérations de tri de la classe `Collections`

### 8.6.3 Collections et itérateurs

L'API 1.5 a ajouté pour les collections une interface bien pratique qui se définit comme suit.

```
public interface Iterable<T>
{
    Iterator<T> iterator();
}
```

Elle permet de parcourir les objets qui l'implémentent avec les méthodes usuelles de `Iterator`, en l'occurrence `hasNext()` et `next()` :

```
public float sommePerimetres()
{
    float sp=0;
```

```

    Iterator<Irectangle> It=listeRectangle.iterator();
    while (It.hasNext()){sp+=It.next().perimetre();}
    return sp;
}

```

... mais également avec une forme particulière de l'instruction for :

```

public float sommePerimetres()
{
    float sp=0;
    for (Irectangle r:listRectangle)
    {sp+=r.perimetre();}
    return sp;
}

```

Et pour finir, la figure 8.4 vous donne un petit aperçu d'un extrait de la hiérarchie des collections, qui mêle interfaces (fond gris) et classes.

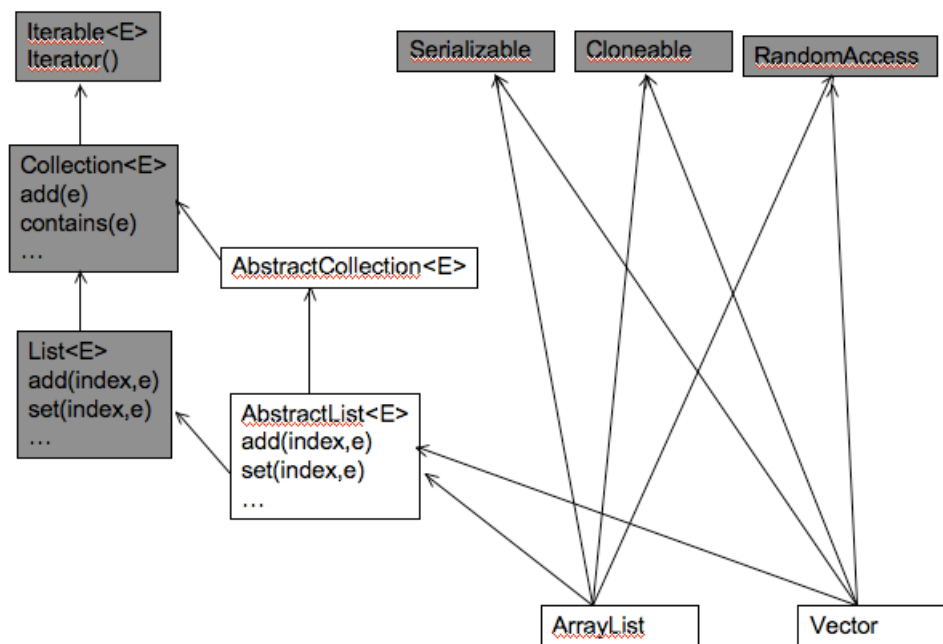


FIG. 8.4 – Extrait des collections