Diagrammes de classes

QUI ?

Les diagrammes de cas d'utilisation modélisent à **QUOI** sert le système.

Le système est composé d'objets qui interagissent entre eux et avec les acteurs pour réaliser ces cas d'utilisation :

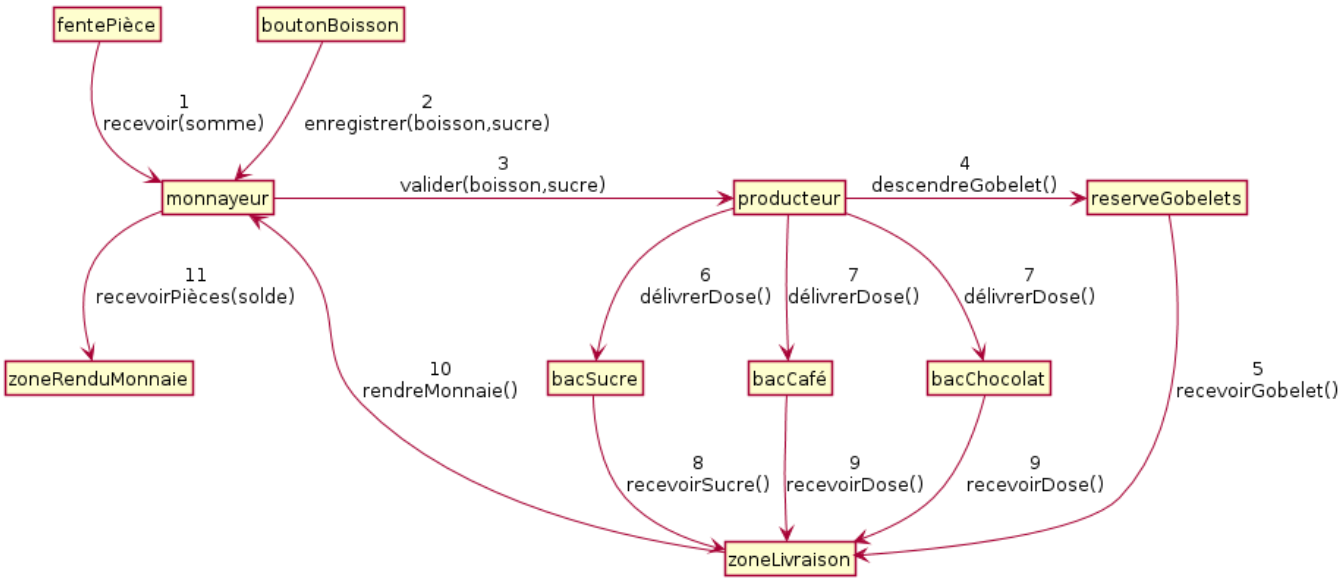
- Les diagrammes de classes permettent de spécifier **QUI** intervient à l'intérieur du système
- Il spécifie également quels liens peuvent entretenir les objets du système

Chaque nouveau diagramme répond à une question différente :

- Ils sont *complémentaires* et pas du tout interchangeable
- Ils doivent être cohérents les uns avec les autres
- C'est en répondant à *toutes* les questions qu'on définit complètement un système

Diagrammes d'objets

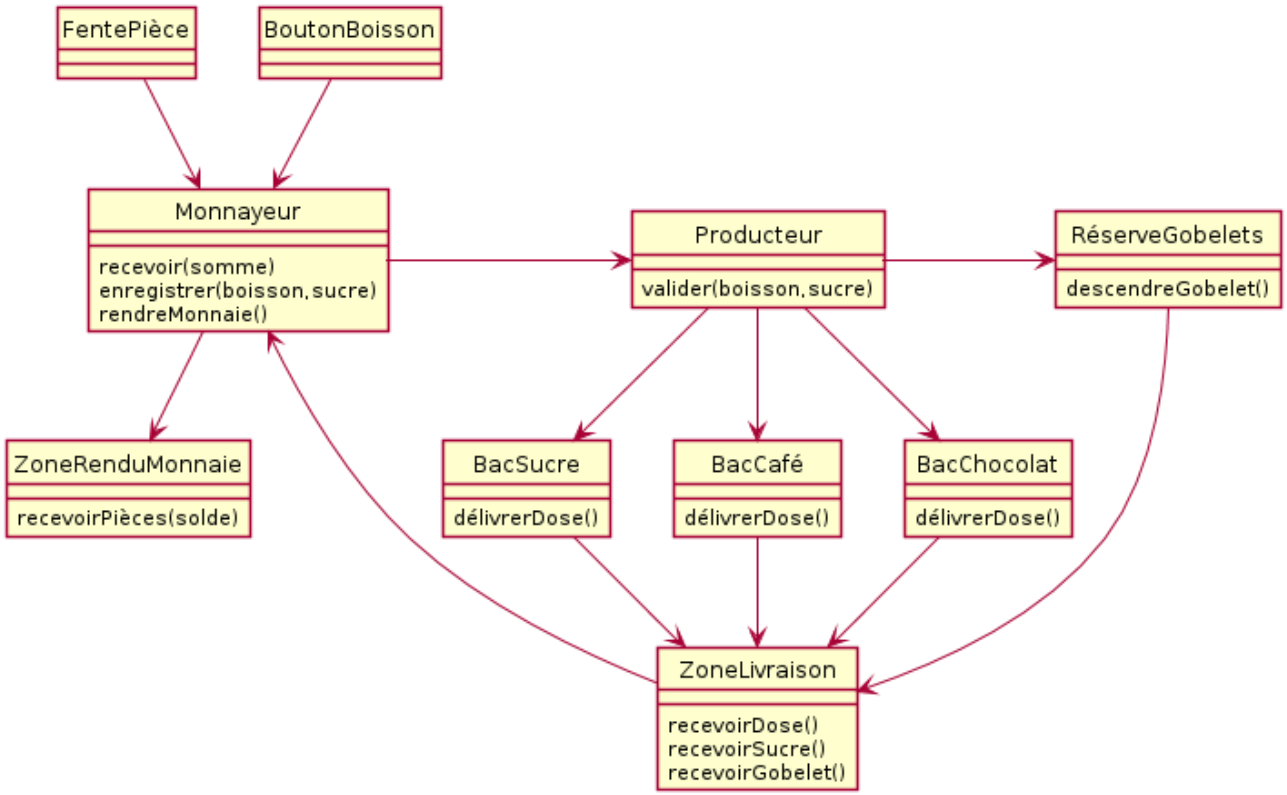
Les objets sont les éléments constitutifs du système, avec leurs données propres et des moyens de de traitement pour réaliser les services attendus.



Attention : non corformité avec la représentation des objets en UML de ce schéma issu d'un TD préliminaire.

Diagrammes de classes

Les classes sont des *types* d'objets.



Objets

Objets du monde réel :

- Le stylo que l'un tient en main.
- Le style qu'un autre a dans sa trousse.
- La facture papier que vous avez reçue à la livraison de votre dernière commande.

Objets informartiques

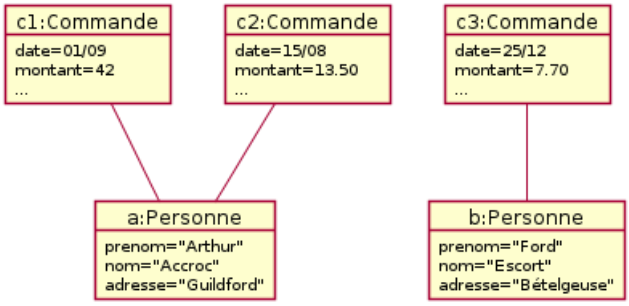
En informatique, les objets ne désignent pas de éléments matériels concrets.

Un objet est un conteneur, qui possède sa propre existence et incorpore des données et des mécanismes en rapport avec une chose tangible. C'est le concept central de la programmation et de la conception orientée objet.

Objets informatiques

- Une structure de données rassemblant les informations disponibles sur une personne (adresse="Guildford", nom="Accroc", prénom="Arthur" etc).
- Une structure de données rassemblant les informations disponibles sur une autre personne (adresse="Bételgeuse", nom="Escort", prénom="Ford" etc).
- Une représentation informatique du stylo tenu en main (avec son propre niveau d'usure, sa propre couleur etc).
- L'enregistrement dans une base de données de votre dernière commande (avec sa propre date, son propre montant, ses propres articles etc).

Objets en UML



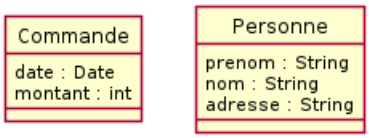
- c1, c2 et c3 sont des commandes
- a et b sont des personnes
- c1, c2 et c3 ont un montant, une date etc…
- a et b ont un nom, un prénom et une adresse

Syntaxe du nom des objets : `nom:Type`

- Le nom peut-être omis, ou le type, mais pas les deux
- Les deux points restent dans tous les cas de figure, même sans rien devant ou derrière, la cas échéant

Classes

Une classe définit un type d'objet.



Une classe déclare donc des *propriétés* communes à un ensemble d'objets :

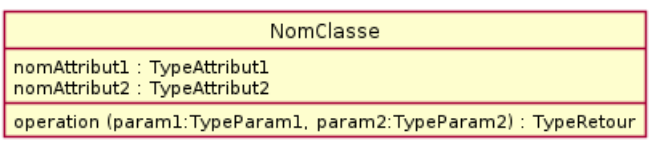
- Les *attributs* correspondant à des variables associées aux objets de la classe
- Les *opérations* correspondant à des opérations (fonctions) associées aux objets de la classe

Exemples de propriétés :

- Toutes les commandes ont une date (attribut).
- Toutes les personnes ont un nom (attribut).
- Toutes les personnes ont un numéro (attribut).
- Tous les articles peuvent être achetés (opération).

Représentation d'une classe en UML

Une classe est composée d'un nom, d'attributs et d'opérations.



- Le nom d'une classe commence par une majuscule
- Le nom d'une propriété commence par une minuscule
- Les types de base (int, long, float, double, boolean) sont en minuscules
- Il n'a pas d'espace dans les noms de classes ou de propriétés
- Pour les noms composés, on fait commencer chaque mot par une majuscule

Concepts et instance

Une instance est la concrétisation d'un concept abstrait.

Instanciation d'une classe :

- Une *classe* est un concept en celà qu'elle n'est que la spécification d'un type
- Un *objet* est une instance avec ses propres attributs et son propre état

Instanciation d'une association :

- Une *association* entre deux classes est un concept : elle décrit quels types d'objets et combien peuvent être associés à d'autres
- Un *lien* est une instance : c'est un lien concret entre deux objets particulier

Attributs d'une classe

Syntaxe (entre accolades, les mentions optionnelles) :

```
{-,#,+,~} nomAttribut : TypeAttribut {[multiplicité]} {=valeurInitiale}
```

- {-,#,+,~} : un symbole parmi les quatre pour définir la *visibilité* de l'attribut : privée, protégée, publique, paquetage
- la *multiplicité* définit le nombre de valeurs dans une collection (tableau, liste)

Exemples :

- `- i : int` pour un attribut privé
- `+ pp : PointPlan` pour un attribut public
- `tabI : int[*]` pour un tableau d'entiers de taille quelconque et à la visibilité non définie
- `tabPP : PointPlans[4]` pour un tableau de 4 points exactement et à la visibilité non définie
- `x : float = 0.0` pour un nombre à virgule initialisé à 0 et à la visibilité non définie

Encapsulation

L'encapsulation est un principe de conception consistant à protéger le coeur d'un système des accès intempestifs venant de l'extérieur.

En UML, les modificateurs d'accès permettent de définir la visibilité des propriétés :

- Un attribut *privé* (−) limite la visibilité d'une propriété à la classe elle-même
- Un attribut *public* (+) ne limite pas la visibilité d'une propriété

- Un attribut *protégé* (#) limite la visibilité d'une propriété à la classe elle-même et à ses sous-classes
- Le symbole (~) permet de limiter la visibilité d'une propriété au package de la classe

Il n'y a pas de visibilité *par défaut*.

Opérations d'une classe

Syntaxe (entre accolades, les mentions optionnelles) :

```
{-,#,+,~} nomOpération ({LISTE_PARAMS}) {:valeurRetour}
```

avec pour LISTE_PARAMS, les paramètres séparés par des virgules. Chaque paramètre s'écrit :

```
nomParamètre : TypeParamètre
```

Exemples :

- `+ toString() : String` pour une opération publique, sans paramètre et retournant une chaîne de caractères
- `~ setAbscisse(x:float)` pour une opération visible dans le paquetage, et prenant une nombre à virgule en paramètre

Notation abrégée d'une classe

Si une classe est déjà définie, il est possible de la représenter simplement, sans ses propriétés.

Relations entre classes

- La relation d'*héritage* est une relation de généralisation/spécialisation permettant l'abstraction de concepts.
- Une *association* représente une relation possible entre les objets d'une classe.
- Une relation de *composition* décrit une relation de contenance et d'appartenance.
- Une *dépendance* est une relation unidirectionnelle exprimant une dépendance sémantique entre les éléments du modèle (flèche ouverte pointillée).

Association

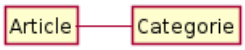
Une association est une relation structurelle entre objets.

- Une association est souvent utilisée pour représenter les liens posibles entre objets de classes données
- Elle est représentée par un trait entre classes
- Elle est souvent dirigée par une flèche

La flèche indique ici que la relation est *uni-directionnelle* : les objets de classe `Article` connaissent ceux de la la classe `Commentaire` auxquels il sont liés, mais pas l'inverse.

Association bi-directionnelle

Certaines associations sont bi-directionnelles mais comme une telle association est plus complexe à implémenter, on prefère l'aviter autant que possible.

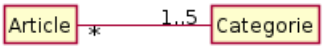


L'absence de flèche indique ici que l'on peut accéder aux catégories à partir des articles qui leur sont liés, et inversement.

Multiplicités

Les multiplicités permettent de contraindre le nombre d'objets intervenant dans les instanciations des associations. On en place de chaque côté des associations.

- Une multiplicité d'un côté spécifie combien d'objets de la classe du côté considéré sont associés à un objet donné de la classe de l'autre côté.
- Syntaxe : `min..max`, où `min` et `max` sont des nombres représentant respectivement les nombre minimaux et maximaux d'objets concernés par l'association.



Ici, le `1..5` s'interprète comme *à un objet donné de la classe `Article`, on doit associer au minimum 1 objet de la classe `Categorie` et on peut en associer au maximum 5.*

Ecriture des multiplicités

Certaines écritures possibles :

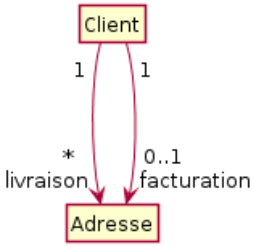
- `*` à la place de `max` signifie *plusieurs* sans préciser de nombre.
- `n..n` se note aussi `n` pour *exatement n*
- `0..*` se note aussi `*`

Exemples :

- `1..*` : au minimum 1 mais sans maximum
- `1..2` : entre un et deux (mais jamais 0, par exemple)
- `*` : autant qu'on le souhaite

Rôles

On peut donner à une classe un rôle dans uns association. C'est surtout utile quand plusieurs associations concernent les mêmes classes en qu'en conséquence, de mêmes objets peuvent être liés par des modalités différentes.



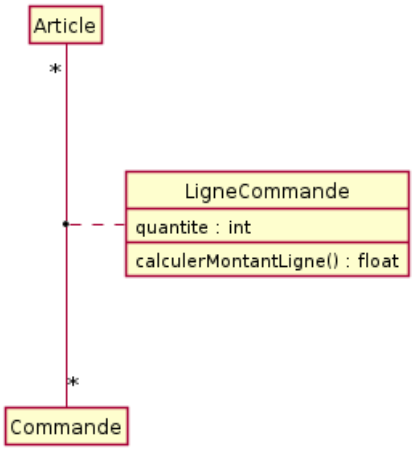
Ici, des adresses peuvent être liées aux clients :

- Les adressers jouent le rôle d'adresses de livraison ou d'adresses de facturation
- Rien n'empêche qu'une adresse soit à la fois une adresse de livraison et un adresse de facturation

Classe-association

Pour faire porter des informations par une association, on emploie une *classe-association*.

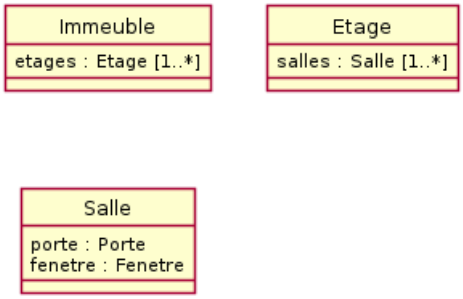
Graphiquement, on la relie à l'association avec des pointillés.



Relations *est une partie de*

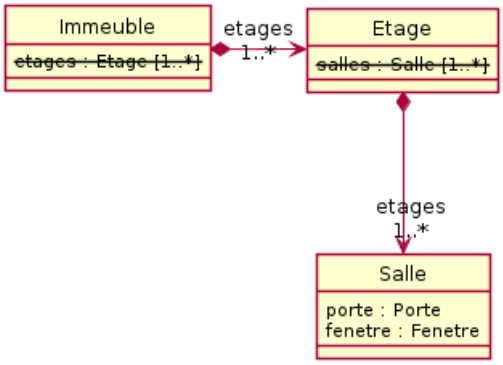
Les objets correspondant aux attributs d'une classe *font partie* des objets de la classe en question :

- Les attributs définissent des parties de la classe
- Ils la *composent*



Expression des compositions avec des relations

Il est possible de représenter plus explicitement les relations de *composition* entre classes.

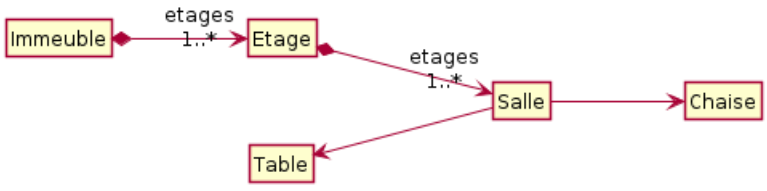


De manière équivalente à la définition d'attributs, on peut utiliser des compositions *unidirectionnelles* et

- Il ne faut pas représenter *à la fois* des attributs et des compositions pour la même relation
- Le nom de l'attribut devient le rôle de la classe composante
- La multiplicité de l'attribut devient le rôle de la classe composante (sans multiplicité explicte, `1`)

Les *compositions* peuvent aussi être bi-directionnelles, mais il n'y a alors plus d'aquivalence avec les attributs : on retient la notion de partie.

Compositions et associations classiques



Il faut remplir deux critères pour décider d'une composition au lieu d'une association classique :

- La relation est de type *du tout à sa partie*
- L'existence des objets composants et composée est liée :

- La destruction d'un objet composite (le tout) implique la destruction des objets composants (les parties)
- La copie d'un objet composite (le tout) implique la copie des objets composants (les parties)
- Plus généralement, les composants ne sont pas partagés par les composites

Traduction Java de la notion d'attribut

Un attribut est implémenté en Java par une variable d'instance de même nom.

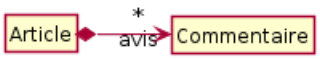

```java
class Article {
    protected designation : String
    protected prix : float
    ...
    (opérations)
}
```

Une multiplicité maximale supérieure à 1 en UML donne lieu en Java à un tableau ou à une collection.

Modificateurs d'accès

UML	***Java***
+	public
-	private
#	protected
~	package

Compositions unidirectionnelles




```java
class Article {
    ...
    Commentaire[] avis ;
    ...
}
```

Les compositions unidirectionnelles sont implémentées par des variables d'instance en Java

- Les éventuels rôles donnent lieu à des noms de variable
- Sans rôle explicite, la variable d'instance prend le nom de la classe, avec une minuscule

- Une multiplicité maximale supérieure à 1 donne lieu en Java à un tableau ou à une collection (ArrayList, par exemple)

Associations unidirectionnelles

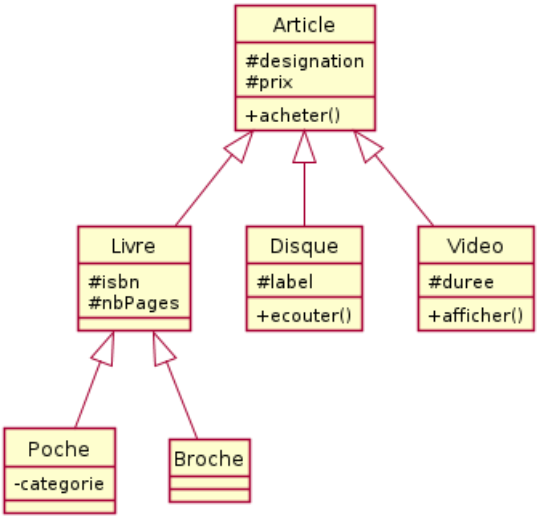
Les associations sont représentées comme des compositions, c'est à dire avec des attributs.

Héritage et abstraction et interfaces

Relation d'héritage

L'héritage une relation de spécialisation/généralisation.

- Les éléments spécialisés héritent de la structure et du comportement des éléments plus généraux (propriétés, associations et autres héritages)
- Exemple : Par héritage d'Article, un livre a d'office un prix, une désignation et une opération acheter(), sans qu'il soit nécessaire de le préciser

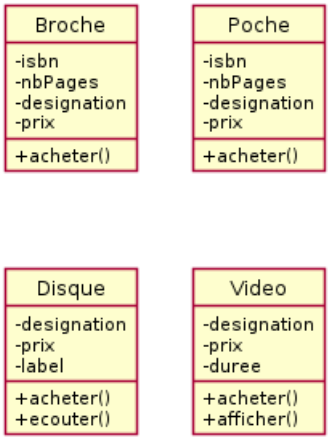


Héritage en Java

```
class Livre extends Article {
    ...
}
```

Le mot clé `extends` en Java n'a rien à voir avec le mot clé `extends` en UML, employé en particulier dans les diagrammes de cas d'utilisation.

Sans héritage



- Pas de types généraux et intermédiaires
- Pas de réutilisation de méthodes comme `afficher()`

Héritage des parents

La classe *enfant* (celle qui hérite) possède toutes les propriétés de ses classes *parents* (attributs et opérations)

- La classe enfant est la classe spécialisée (ici Livre)
- La classe parent est la classe générale (ici Article)

Elle n'a accès qu'aux propriétés publiques (comme tout le monde) et *protégées* de ses parents. Le propriétés publiques restent masquées pour les enfants.

Terminologie

La *signature* d'une opération est constituée de son nom et de ses paramètres (pas de sa valeur de retour).

La *surcharge* d'opérations (même nom, mais signatures des opérations différentes) est possible dans toutes les classes.

Polymorphisme :

- Une classe enfant peut *redéfinir* (même signature) une ou plusieurs méthodes de la classe parent
- Un objet utilise en principe la définition la plus spécialisée dans la hiérarchie des classes.

Substitution : une instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue (par exemple, toute opération acceptant un objet d'une classe Animal doit accepter tout objet de la classe Chat (l'inverse n'est pas toujours vrai).

Classe abstraite

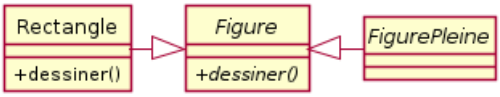
Une opération est dite *abstraite* lorsqu'on connaît sa signature mais pas la manière dont elle peut être réalisée (en UML, on écrit son nom en italique).

- Il appartient aux classes enfant de définir les opérations abstraites.

Une classe est dite *abstraite* lorsqu'elle définit au moins une méthode abstraite ou lorsqu'une classe parent contient une méthode abstraite non encore réalisée (en UML, on écrit son nom en italique).

- Tant qu'au moins une méthode demeure abstraite (même héritée), la classe ne peut être qu'abstraite.

Il est impossible d'instancier directement une classe abstraite.

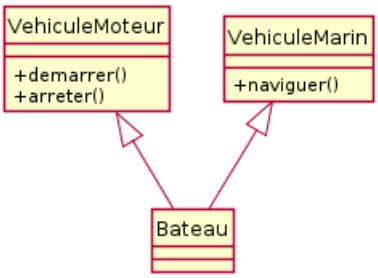


Héritage multiple

Une classe peut avoir plusieurs classes parents. On parle alors d'*héritage multiple*.

- Le langage C++ est un des langages objet permettant son implantation effective.
- Java ne le permet pas et on doit employer des interfaces.

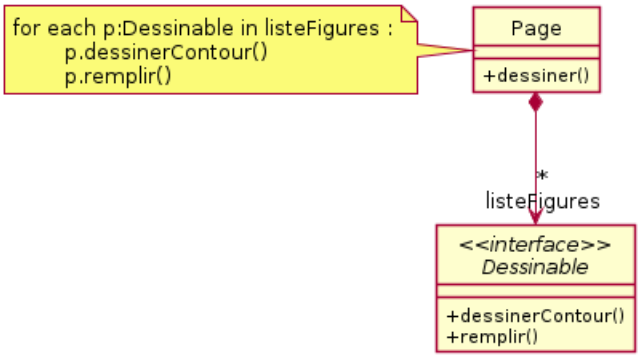
L'emploi de l'héritage multiple est souvent peu justifié et d'autres solutions de conception sont souvent préférables.



Interface : utilisation

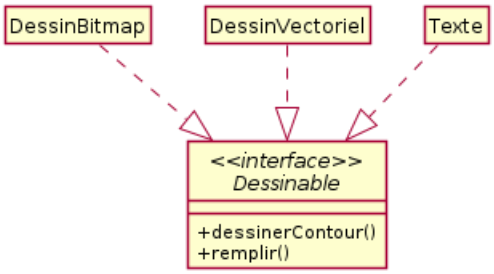
Les interfaces sont très utilisées en COO et en POO. On les utilise d'ailleurs plus régulièrement que l'héritage, par exemple.

Le rôle d'une interface est de regrouper un ensemble d'opérations assurant un service cohérent que des classes sont susceptibles d'offrir. On peut ainsi définir des classes clientes (ici, `Page`) avant d'avoir défini toutes les implémentations requises de l'interface.



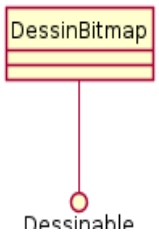
Selon le principe de `substitution`, on pourra employer une classe réalisant une interface partout où l'interface est attendue.

Interface : définition



- Une interface est définie comme une classe, avec les mêmes compartiments. On ajoute le stéréotype `<< interface >>` avant le nom de l'interface.
- On utilise une relation de type *réalisation* entre une interface et une classe qui l'implémente (flèche d'héritage en pointillés).
- Les classes implémentant une interface doivent implémenter toutes les opérations décrites dans l'interface.

Notation lollipop des interfaces

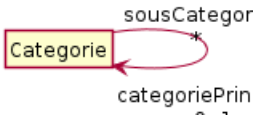


Compléments sur les diagrammes de classes

Associations réflexives

L'association la plus utilisée est l'association binaire (reliant deux classes).

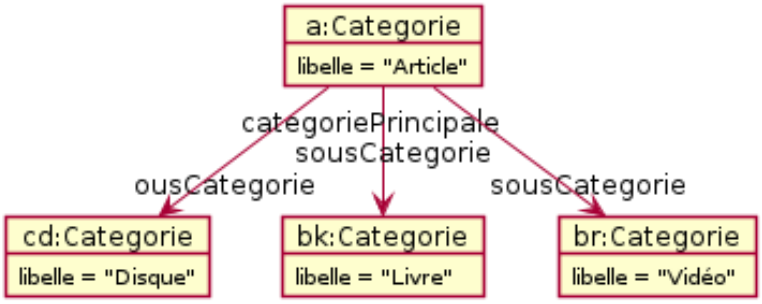
Parfois, les deux extrémités de l'association pointent vers la même classe. Dans ce cas, l'association est dite *réflexive*.



Instantiation d'associations réflexives

Instanciée, une association réflexive fait intervenir plusieurs objets. Par exemple :

- Un premier objet de classe `Categorie` joue le rôle de `categoriePrincipale`
- Un second objet joue le rôle de `sousCategorie`



Associations n-aires

Artité des associations :

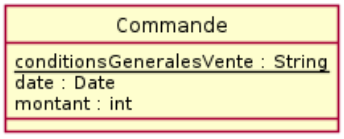
- La plupart des associations sont binaires
- Les associations réflexives sont unaires.

Il arrive que ce ne soit pas suffisant. Dans ce cas, on peut définir des relations n-aires.

- Pour déterminer les multiplicités d'un côté, on considère le nombre d'objets liés à un (n-1)-uple quelconque formé d'objets de tous les autres côtés
- Ces associations se rencontrent assez rarement en pratique. Dans la plupart des cas, il est en effet plus pertinent d'utiliser une classe-association.

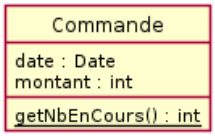
Attributs de classe

- Par défaut, les attributs correspondent à des variables d'instance, dont la valeur peut changer pour chaque instance de la classe considérée.
- Les attributs de classe sont des variables dépendant des classes et pas de leurs instances.
    - Graphiquement, un attribut de classe est souligné
    - De tels attributs correspondent aux variables de classe (`static` en Java)



Opérations de classe

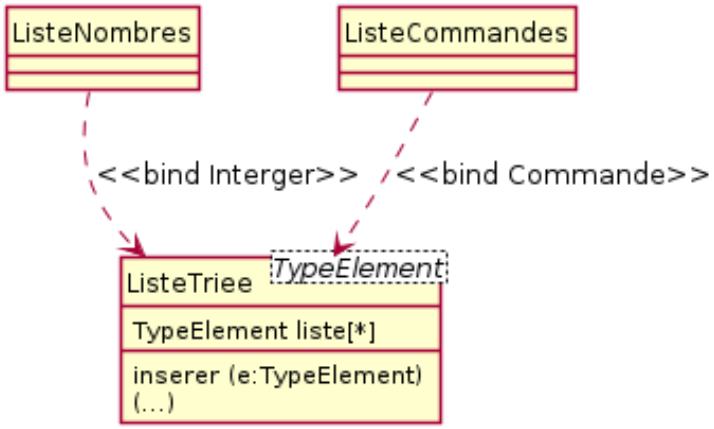
- Une opération de classe est une propriété de la classe, et non de ses instances.
- Elle n'a pas accès aux attributs des objets de la classe.



Classe paramétrée

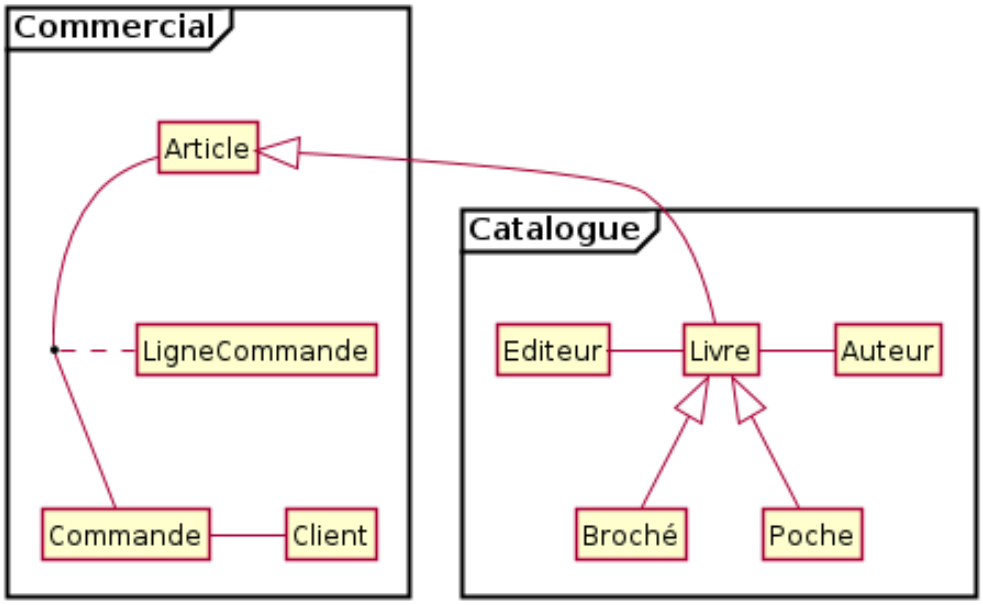
Pour définir une classe générique et paramétrable en fonction de valeurs et/ou de types :

- Définition d'une classe paramétrée par des éléments spécifiés dans un rectangle en pointillés
- Utilisation d'une dépendance stéréotypée « bind » pour définir des classes en fonction de la classe paramétrée.



Paquetages

Il est recommandé de mettre dans des *paquetages (packages)* différents les éléménts qui ont un raport plus étroi les uns avec les autres.



Ici, par exemple, on identifie un package pour les éléments liés aux aspects commerciaux, et un autre pour les éléménts concernant le catalogue de produits.

A la manière de dossiers, on peut imbriquer des packages dans d'autres packages.

Chaque classe dispose d'un nom absolu et non ambigu `Package::SousPackage::NomClasse`. Exemple : `java.lang.String`.

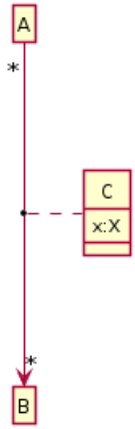
Impact des associations sur un code en java

Association unidirectionnelle * vers 1

Ce type d'association est le plus courant, et heureusement le plus simple à impélmenter.


```
class A {
    private B b;
    (getter et setter publics)
}
```

Association unidirectionnelle * vers *




```
class A {
    List<B> b;
    (getter, adder et remover publics)
}
```

Association bididirectionnelle 1 vers 1

Il faut maintenir la cohérence des référencements dans les deux sens. Un simple

```
public class A {
    private B b;
    public void setB(B b) {
        if ((b!=null) && (b.getA()!=null))   // si b est déjà connecté à un autr
            b.getA().setB(null);             // cet autre A doit se déconnecter
            this.b=b;
            b.setA(this);
    }
    public B getB() {return(b);}
}
```

Il faut également définir l'opération symétrique dans `B`.

Association unidirectionnelle vers 1

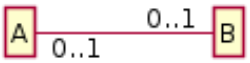
C'est paradoxalement plus complexe que le cas bidirectionnel :

- Les instances de B ne doivent en principe pas référencer des instances de A
- L'unicité de l'instance de A est donc assez délicate à contrôler

En pratique, on peut réfléchir à la nécessite de la multiplicité maximale de 1 à la source.

Si elle est vraiment justifiée, on peut admettre qu'un développeur code en réalité une association bi-directionnelle et joue sur les modificateurs d'accès pour rendre la relation plus asymétrique.

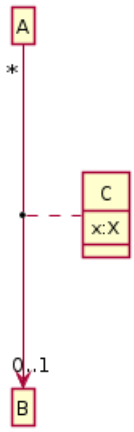
Association bidirectionnelle 1 vers *




```
public class A {
    private List <B> b;
    public A() {this.b = new List<B>(); }
    public List<B> getList() {return(this.b);}
    public void remove(B b){this.b.remove(b);}
    public void addB(B b){
        if(!this.b.contains(b)){
            if (b.getA()!=null) b.getA().remove(b);
            b.setA(this);
            this.b.add(b);
        }
    }
}

public class B {
    private A a;
    public A getA() {return (this.a);}
    public void setA(A a){
        if((a!=null) && (!a.get().contains(this)) {
            if (this.a != null) this.a.remove(this);
            this.setA(a);
            this.a.get().add(this);
        }
    }
}
```

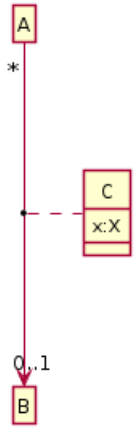
Association bidirectionnelle * vers *

Classe association unidirectionnelle * vers 1



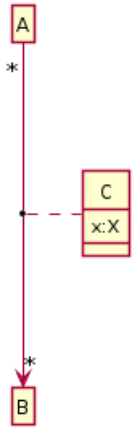

```
class A {
    private B b;
    private C c;
    (getB, setB et éventuellement getC)
    public getX {return this.x.getX();}
    public setX(x:X) {this.x.setX(x);}
}

class C {
    private X x;
    (getter et setter publics)
}
```

Classe association unidirectionnelle * vers 1 (plus simple)



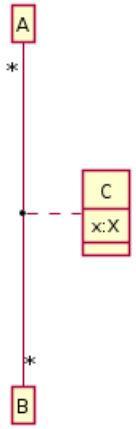

```
class A {
    private B b;
    private X x;
    (getter et setter publics)
}
```

Classe association unidirectionnelle * vers *




```
class A {
    private List<C> c;
    (getter et setter publics)
    public getB {return this.c.getB();}
    public setB(b:B) {this.c.setB(b);}
    public getX {return this.c.getX();}
    public setX(x:X) {this.c.setX(x);}
}
class C {
    private B b;
    private X x;
    (getters et setters publics)
}
```

Classe association bidirectionnelle * vers *



En pratique, on se ramène à un cas comme :