Bases de la conception orientée objet

Concepts objet - Diagrammes de Classes

Petru Valicov
petru.valicov@univ-amu.fr

http://pageperso.lif.univ-mrs.fr/~petru.valicov/Teaching.html

2017-2018

Institut Universitaire de Technologie
Aix-Marseille Université
Département Informatique, Aix

Motivation

- Diagramme de cas d'utilisation ≈ à QUOI sert le système.
- Le système est composé d'**objets** qui interagissent entre eux et avec les acteurs pour réaliser les cas d'utilisation.
- Les **diagrammes de classes** permettent de spécifier la structure et les liens entre ces objets

Les objets - vous avez dit bizarre ?

- Un **objet** informatique définit une représentation simplifiée, une abstraction d'une entité du monde réel
- Impact sur la façon de penser

Les abstractions :

- Mettent en avant les caractéristiques essentielles
- Dissimulent les détails

Les objets

Exemple d'objet *Voiture* :

- Identité : numéro d'identification, code-barres etc
- Services rendus par l'objet : Démarrer, Arrêter, Accélérer, Freiner, Climatiser, ...
- État : marque, modèle, couleur, vitesse...

Définition

Objet = identité + état + comportement

Fonctionnement interne ?

À priori en tant qu'utilisateur vous n'avez aucune idée

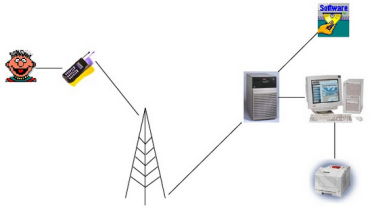
Les objets

Définition

Objet = identité + état + comportement

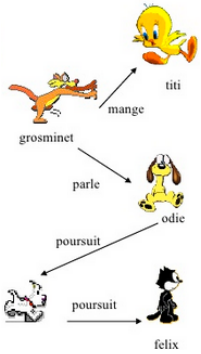
- Chaque objet possède une **identité** unique qui lui est propre et qui le caractérise
  - exemple : deux étudiants homonymes n'ont pas la même identité
  - dans une base de données : clef primaire
  - en `C++` et `Java` : référence unique (mot clef `this`)
- L'**état** : valeur de tous les attributs d'un objet à un instant T
  - évolue dans le temps
  - à un instant T il est la conséquence des comportements passés
  - par exemple, l'état d'un étudiant : nom, prénom, date de naissance, adresse, diplômes obtenus, diplôme en cours, ...
- Le **comportement** regroupe toutes les compétences d'un objet

Application



C'est un ensemble d'objets collaborant entre eux.

- Le comportement d'une application (orientée objet) repose sur la communication entre les objets qui la composent.
- Les objets communiquent en échangeant des messages à travers des appels de fonctions

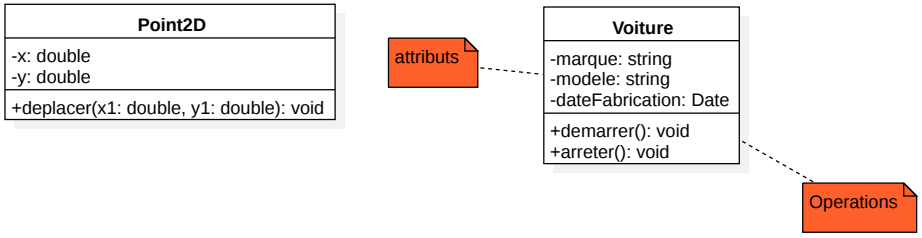


Les classes

Une classe est une description abstraite d'un ensemble d'objets "*de même nature*" :

- n'a pas d'état, ni d'identité (vue statique du système)
- c'est un "**modèle**" pour la création de nouveaux objets
- "**contrat**" garantissant les compétences minimales d'un objet

Classe : **nom**, **attributs** et **opérations**



Les classes

- des éléments concrets (ex. livre, client)
- des éléments abstraits (ex. commande de marchandise)
- des composants d'une application (ex. les éléments de l'interface graphique)
- des structures informatiques (ex. tables, liste de contactes)
- des éléments comportementaux (ex. des tâches, paiement)

Classe vs Objet

Un objet est une concrétisation d'une classe.

- Tout objet appartient à une classe et connaît de façon implicite la classe à laquelle il appartient.
- Tout objet est une **instance** de sa classe.
- Un objet est une instance d'une et une seule classe.

Une classe est une **défintion** (ou **modèle**) pour définir des objets

Exemples :

- L'objet *Philippe* est une instance de la classe `Humain`.
- L'objet *RenaultQuatreL* est une instance de la classe `Voiture`.
- L'objet *M2104BCOO* est une instance de la classe `Cours`.

Classe vs Objet

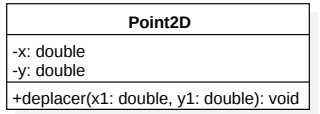
Objet/Classe a deux aspects :

- Interface : vue externe de l'objet
- Corps : implémentation des comportements (opérations) et des attributs.

L'utilisateur/programmeur ne connaît que l'interface de l'objet.

L'implémentation est masquée et non accessible à l'utilisateur. (l'encapsulation)

Classe vs Objet – exemples



En base de données :

ID	x	y
0	1.2	-0.6
1	0.5	0.5
2	0	1
3	3.4	0.3

- Table ≈ Classe
- Chaque ligne est un objet (instance de la classe)
- Important : **pas d'équivalent des méthodes**

En `C++` :

```
class Point2D {
   private :
      double x;
      double y;

   public :
      void deplacer(double, double);
      /* Le code des méthodes à écrire
      au moment où on programme */
}
```

En `Java` :

```
class Point2D {
   private double x;
   private double y;

   public void deplacer(double nouvX, double nouvY) {
   /* Le code des méthodes à écrire
   au moment où on programme */
   }

}
```

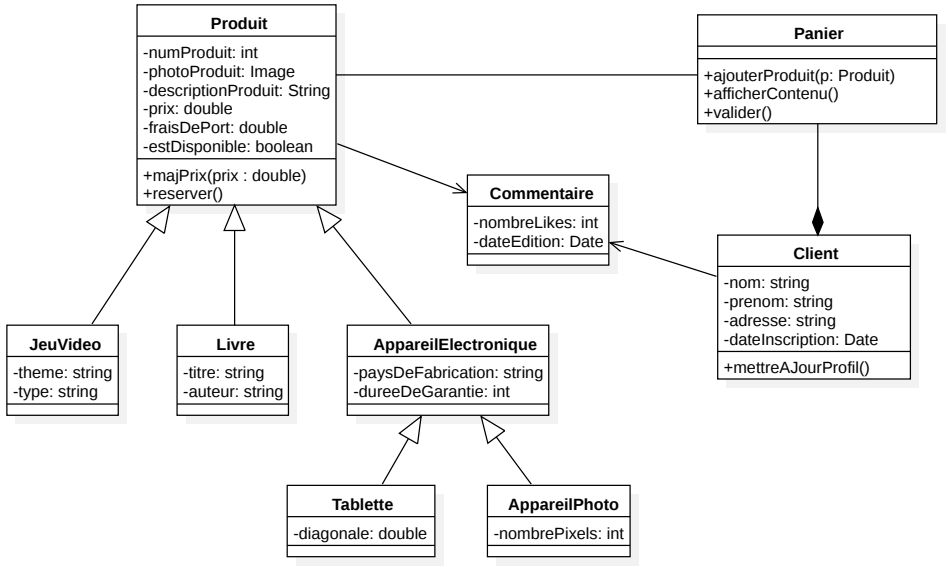
Diagrammes de classes (modèle statique)

Ce sont les plus importants pour la modélisation orientée objet.

Le diagrammes de classes décrit la **structure interne** du système contrairement au diagramme de cas d'utilisation qui montre le système du point de vue des acteurs.

Il permet de fournir une représentation abstraite des objets du système qui vont interagir ensemble pour **réaliser** les cas d'utilisation.

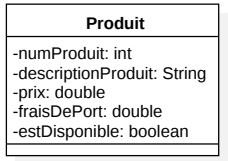
Diagramme de classes - exemple jouet

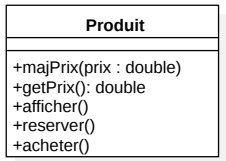


Propriétés : attributs et opérations

Les attributs et les opérations sont les **propriétés** d'une classe. Par convention leurs noms commencent par une minuscule.

Un **attribut** décrit une donnée de la classe - c'est une généralisation de la notion de *variable*.



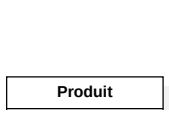


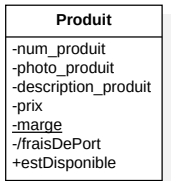
Une **opération** est un service offert par la classe (un traitement que les objets correspondants peuvent effectuer - une méthodes).

Un attribut est défini par son nom et son type.
Une opération est définie par sa *signature* :
nom + paramètres + type de retour

Diagramme de classes

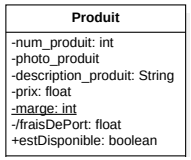
Plusieurs niveaux de détails

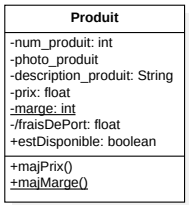




⟵ Détails d'analyse

Détails de conception ⟶





Syntaxe

- Attribut
  - <visibilité><nom> : <type> :=<valeur par défaut>
- Opération
  - <visibilité><nom>(<liste param>) : <type retour>
- Liste paramètres
  - <param1>, <param2>, <param3>
  - <nom> : <type>=<valeur par défaut>

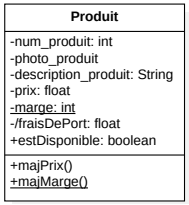
Type

- Type d'attribut
  - *integer*, *boolean*, *double*, *string*, etc.
  - *date*, *time*, *currency*
  - pour les attributs de type *objet*, on utilise une association et des noms de rôle
- Type de retour et de paramètres pour les opérations (type de base ou `void`)

Attributs et opérations

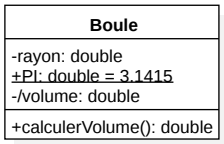
Attributs et opérations de classe :

- Souligné (le terme *static* en C++ et Java)
- Attribut de classe partagé par **toutes** les instances de la classe (garde une valeur unique et partagée par toutes les instances de classe)
- Opération de classe sur des attributs de classe uniquement



Attributs dérivés :

- peuvent être calculés à partir d'autres attributs et de formules de calcul
- Exemple : Π vs Volume
- peu utilisés

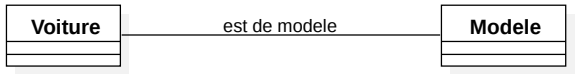


Relations entre classes

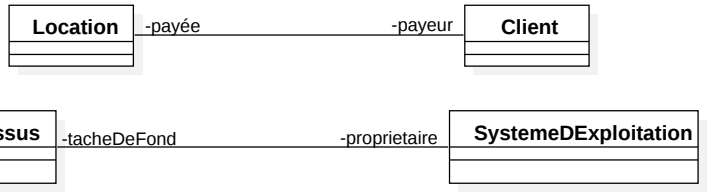
- Une relation d'**héritage** est une relation de **généralisation/spécialisation** permettant l'abstraction de concepts.
- Une **dépendance** est une relation unidirectionnelle exprimant une dépendance sémantique entre les éléments du modèle (flèche ouverte pointillée).
- Une **association** représente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

Relations entre classes - Association

- connexion sémantique entre des classes représentée par un trait
- le plus souvent entre deux classes (binaire)
- parfois on peut indiquer le sens de la lecture (par un verbe)

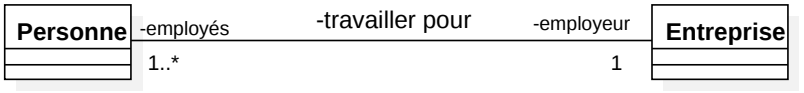


- l'association peut être nommée par **ses rôles**.



Les rôles d'une association

- très utilisés
- le rôle décrit comment une classe voit une autre classe à travers l'association
  - devient le nom d'un champ en C++ et Java
  - il y a deux rôles (ne vous trompez pas de côté ! ! !)
- multiplicité ou cardinalité



Une possible réalisation C++ :

```
class Personne {
   private: Entreprise *employeur;
}
```

```
class Entreprise {
   // un tableau d'employés
   private : vector<Personne> employés;
}
```

Une possible réalisation en Java :

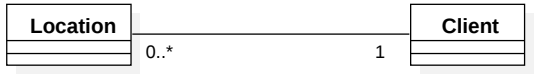
```
class Personne {
   private Entreprise employeur;
}
```

```
class Entreprise {
   // un tableau d'employés
   private ArrayList<Personne> employés;
}
```

Multiplicités des associations

La notion de **multiplicité** (ou **cardinalité**) permet de contraindre le nombre d'objets intervenant dans les instanciations des associations.

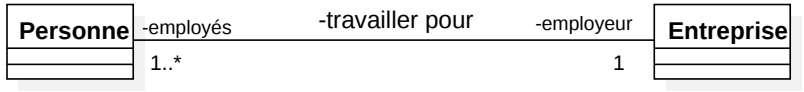
Exemple : *une location est payée par un et un seul client, alors que le client peut réserver plusieurs locations.*



La syntaxe de m

- **1** : toujours un et un seul (dès la création de l'objet)
- **0..1** : zéro ou un
- `m..n` : de *m* à *n* (entiers > 0)
- ***** ou ***..0** : de zéro à plusieurs
- **1..*** : au moins un

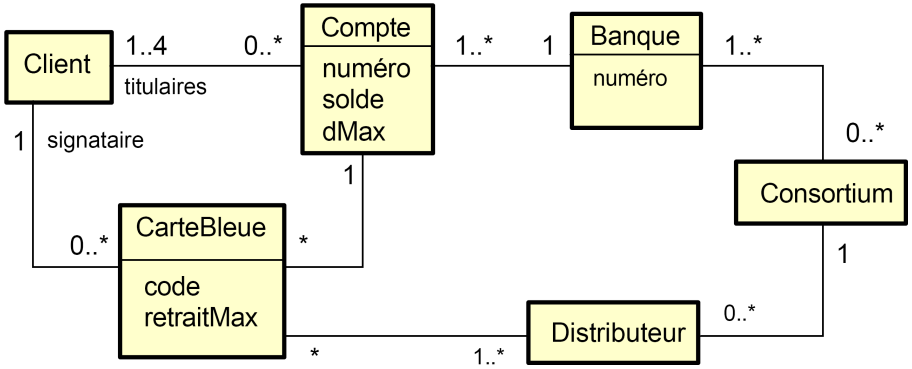
} vecteur/liste en `C++/Java`



Rôles et multiplicités des associations

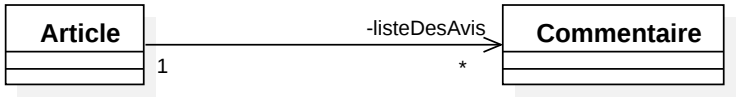
Règle générale :

Exemple :



Navigabilité d'une association

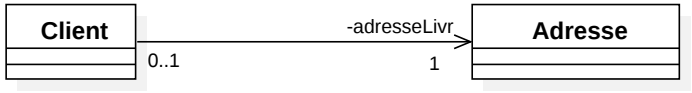
- La **navigabilité** permet de spécifier dans quel(s) sens il est possible de traverser l'association à l'exécution.
- On restreint la navigabilité d'une association à un seul sens à l'aide d'une flèche.



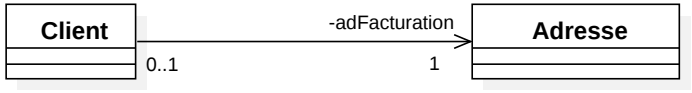
Exemple : Connaissant un article on connaît les commentaires, mais pas l'inverse. En `C++` ou `Java`, `listeDesAvis` pourrait être un tableau de références vers des objets de type `Commentaire`.

Navigabilité et multiplicité

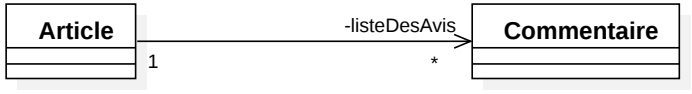
- Association unidirectionnelle de 1 vers 1



- Association bidirectionnelle de 1 vers 1

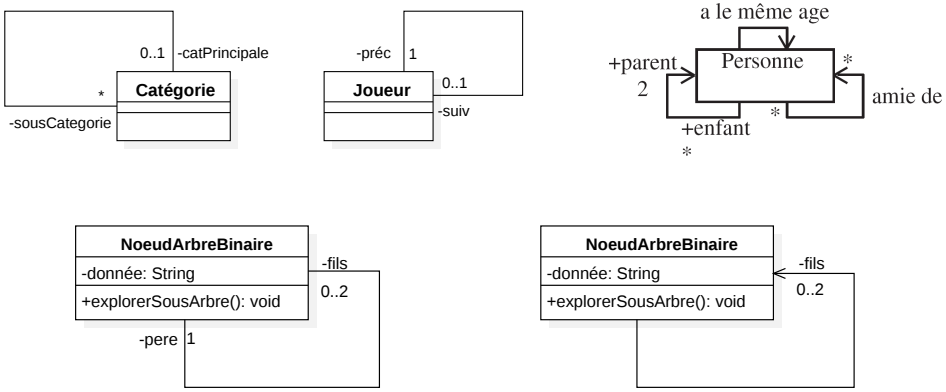


- Association unidirectionnelle de 1 vers *



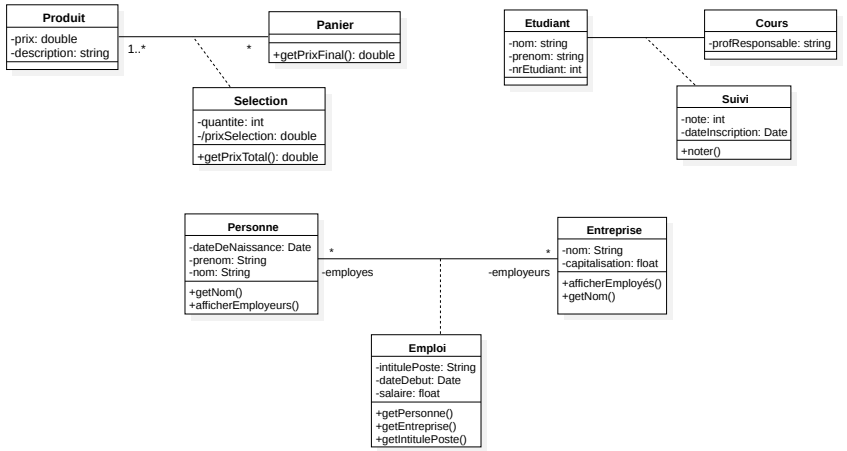
Associations réflexives

- 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** ».



Classe-association

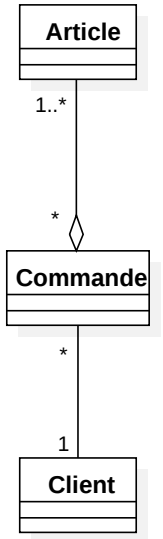
- C'est une association qui possède des propriétés
- Elle est plus *riche* qu'une association simple
- L'identifiant de la classe-association est déterminé par les deux classes reliées par l'association



Associations : agrégation

- Une **agrégation** est une forme d'association plus forte que l'association simple
- Représente la relation d'**inclusion faible** d'un élément dans un ensemble
- On représente l'agrégation par l'ajout d'un losange vide du côté de l'agrégat
- On utilise souvent le terme **composition faible**

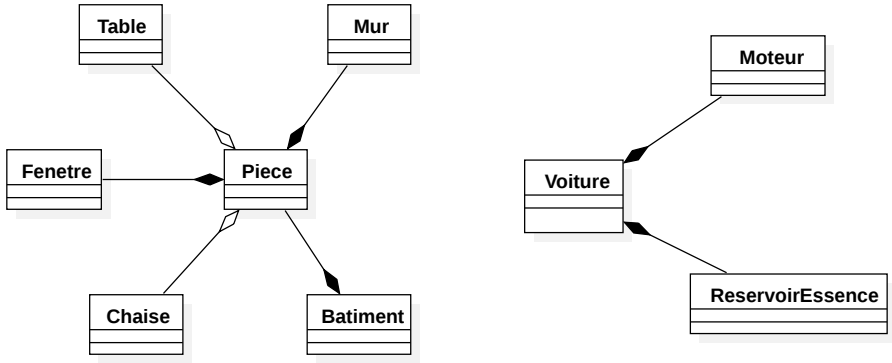
Elle dénote une relation d'un ensemble à ses parties. L'ensemble est l'**agrégat** et la partie l'**agrégé**.



Associations : composition

- L'association la plus "forte". On utilise un losange plein
- Décrit une **contenance** structurelle entre instances
- Cardinalité maximum de 1 obligatoire

La **destruction** et la **copie** de l'objet composite (l'ensemble) impliquent respectivement la destruction ou la copie de ses composants (les parties).



Composition vs agrégation

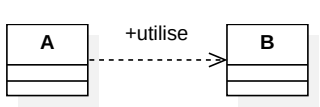
- Dès que il y a la notion de contenance on utilise une agrégation ou une composition
- La composition est aussi dite **agrégation forte**

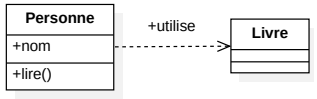
Comment décider entre la composition et l'agrégation ?

Si les composants ont une autonomie vis-à-vis du composite alors préférez l'agrégation. Mais tout dépend de l'application que vous développez...

Dépendance

- Relation unidirectionnelle exprimant une dépendance sémantique entre deux classes/interfaces
- Représenté par un trait discontinu orienté

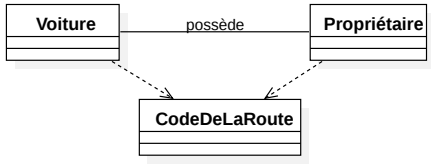




- Généralement `A` dépend de `B` si :
  - `A` utilise `B` comme argument dans la signature d'une méthode
  - `A` utilise `B` comme variable locale d'une méthode

Exemple : la modification du code de la route a un impact sur

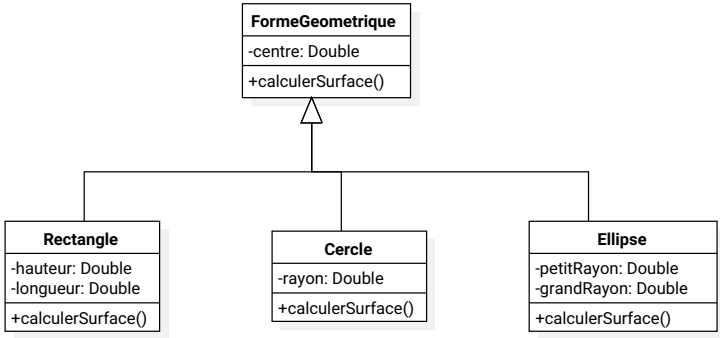
- l'attitude du conducteur
- des caractéristiques des voitures



Relation très générale : toutes les relations possibles entre les classes sont des dépendances

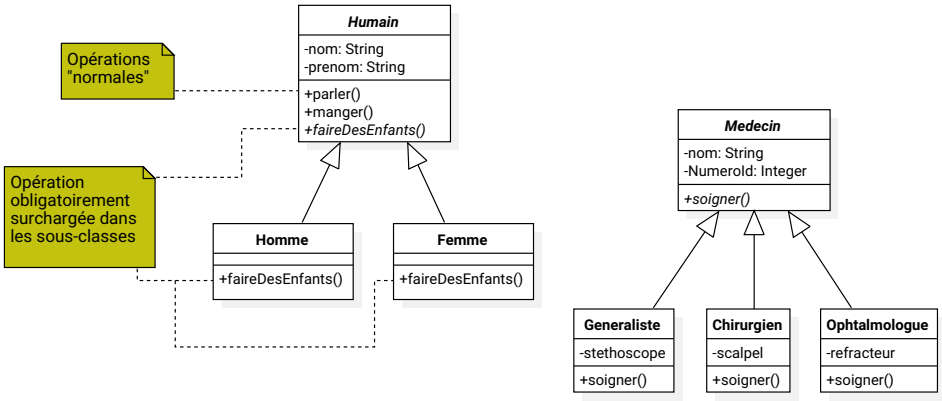
Relations entre classes - Héritage/Généralisation

- indique qu'une classe est un cas plus *générale* d'une autre
- se traduit par la notion d'**héritage** en Programmation Objet
- non-réflexive
- transitive : les classes spécialisées héritent de la structure et du comportement des classes plus générales (attributs, opérations, associations et nouveaux héritages)



Classe abstraite

- Classe qui ne peut pas être instanciée
- Sert uniquement de super classe à d'autres classes
- Exemple : les hommes et les femmes vs *humains*
- Peut contenir des méthodes abstraites (pour garantir que toutes les classes filles aient ces méthodes)



Visibilité - Encapsulation

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

- La visibilité définit le degré de protection

  « + » : visibilité **publique** *i.e.* toutes les classe y ont accès (cas le plus fréquent pour les méthodes)

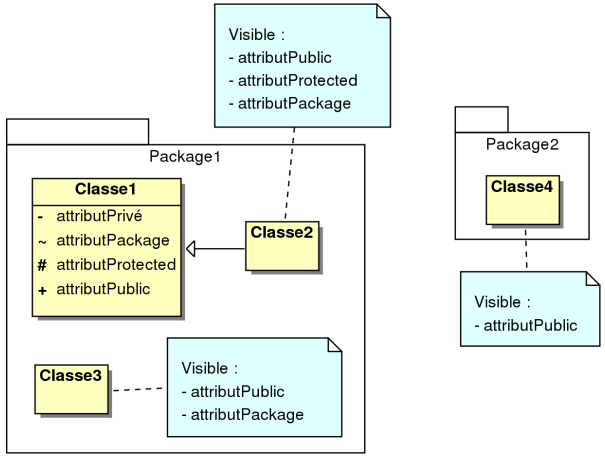
  « - » : visibilité **privée** *i.e.* inaccessible à tout objet hors de la classe (attributs et opérations internes à la classe)

  « # » : visibilité **protégée** *i.e.* les sous-classes y ont accès (cas le plus fréquent pour les attributs)

  « ∼ » : visibilité **de paquetage**

- Pas de visibilité par défaut en UML
- Visibilité publique : la classe s'engage à toujours fournir ce service ("contrat")

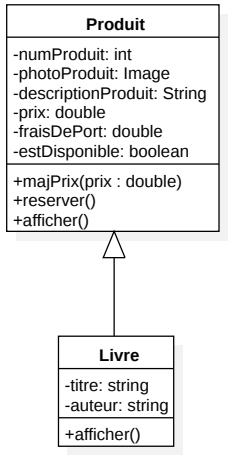
Exemple d'encapsulation



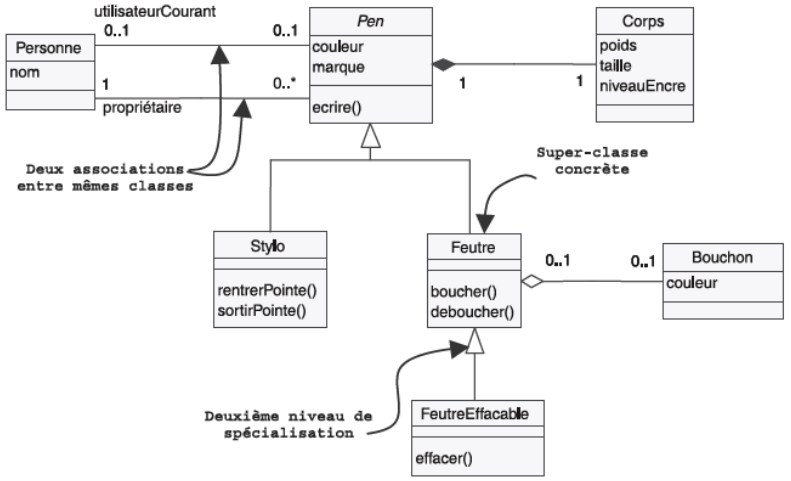
Règle d'or : à priori tous les attributs devraient être privés ! Tout autre choix doit être justifié.

Relation d'héritage et propriétés

- "Héritage" des propriétés des classes parents
  - La classe **enfant** est la classe spécialisée (ici *Livre*)
  - La classe **parent** est la classe générale (ici *Produit*)
- La classe enfant n'a pas accès aux propriétés privées
- La classe enfant peut redéfinir des méthodes de la classe parent (ex : la méthode *afficher()* est redéfinie dans la classe *Livre*)
- **Principe de substitution** - toute opération acceptant un objet de type `Article` doit accepter un objet de type `Livre`.



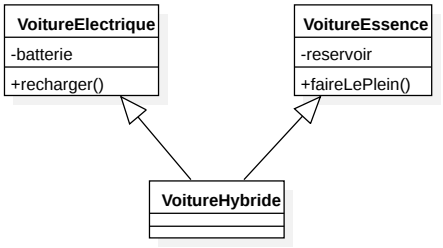
Relations entre classes – résumons



Source : UML2 par la pratique de P. Roques

Héritage multiple

- Une classe peut avoir plusieurs classes parents

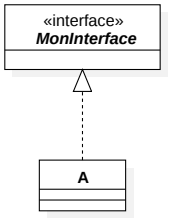


- Dangereux - problème du diamant
- Présents dans les langages `C++`, `Python`...
- Interdit en `Java`, `C#`... mais on peut ruser en employant des *interfaces*.

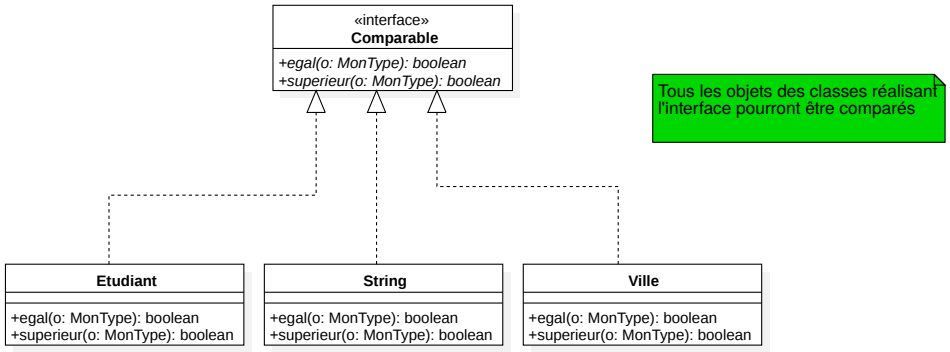
Interfaces

- Classe ≈ nom + données + opérations (privées, publiques etc.)
- Souvent on s'intéresse uniquement aux *services* rendus par la classe *i.e.* à son **interface**

- Une interface est définie comme une classe, avec les mêmes compartiments
- On dit que `A` est une **réalisation** de l'interface `Ia`
- Équivalent à une classe abstraite pure en `C++`



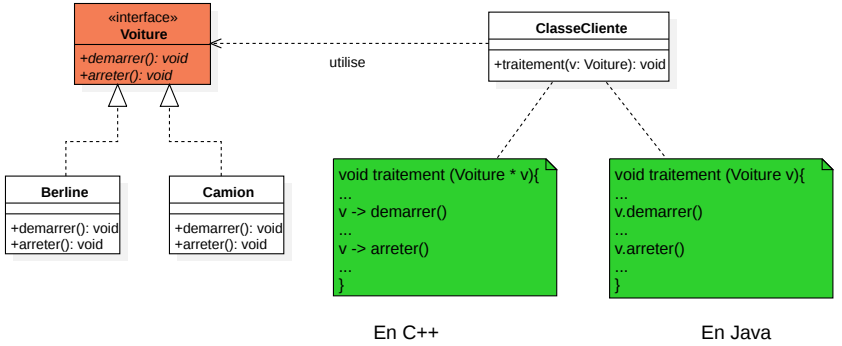
Exemple d'interface



- Les méthodes `égal()` et `supérieur()` sont abstraites.
- Les méthodes abstraites **doivent être implémentées** dans chaque réalisation de `Comparable`.

Classe cliente d'une interface

- Quand une classe dépend d'une interface pour réaliser ses opérations, elle est dite **classe cliente de l'interface**
- On utilise une relation de dépendance entre la classe cliente et l'interface requise

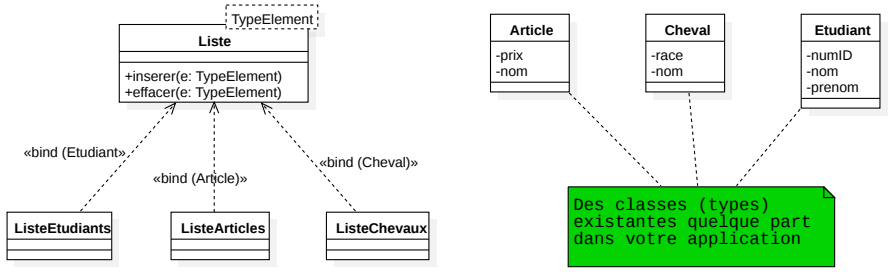


- Toute classe implémentant l'interface pourra être utilisée - principe de substitution

Classe paramétrée

Généricité

En Orienté Objet, correspond à la possibilité de paramétrer une Classe, une fonction, ou une méthode d'une classe par un paramètre générique formel représentant un type arbitraire.



Une classe paramétrée est essentiellement utilisée pour représenter une "structure de données"

Relations entre classes : bilan

Association

- Relation entre plusieurs classes
- Elle représente une abstraction d'un lien structurel qui peut exister entre les différents objets

Agrégation

- Forme particulière d'association entre deux classes
- L'agrégation est une association non symétrique
- Une agrégation peut notamment (mais pas nécessairement) exprimer qu'une classe fait partie d'une autre ("l'agrégat")

Composition

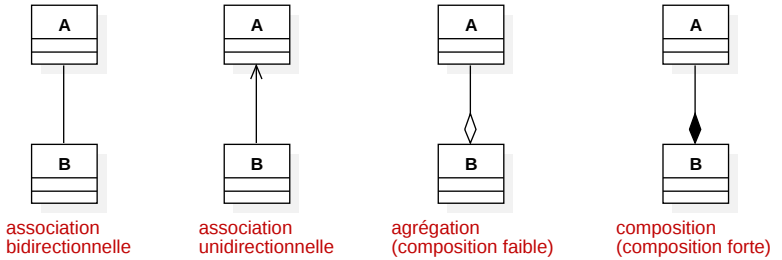
- La composition est une agrégation forte (agrégation par valeur)
- Exprime le fait qu'une classe est composée d'une ou plusieurs autres classes

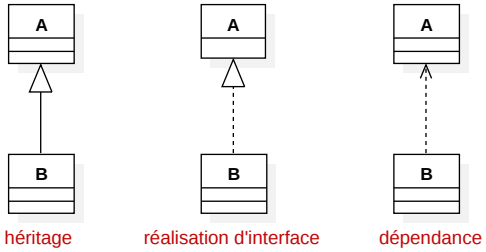
Relations entre classes : bilan

Spécialisation et généralisation : Le mécanisme qui permet de définir de nouvelles classes à partir de classes déjà existantes.

- **Généralisation** : factorisation dans une classe (appelée super-classe) de propriétés de plusieurs classes
- **Spécialisation** : inverse de le généralisation, consiste à créer à partir d'une classe plusieurs classes spécialisées

Relations entre classes : bilan





Ne vous trompez pas de flèche ! ! !

Les classes - bilan

Encapsulation

- Rassembler les attributs et les méthodes en masquant les détails d'implémentation
- Garantir l'intégrité des données de l'objet
- Peut être réalisée en particulier par des constituants **privés** des objets

Polymorphisme

- **Poly** = plusieurs, **morphisme** = forme
- Propriété d'un élément de pouvoir se présenter sous plusieurs formes
- Capacité donnée à une même opération de s'effectuer différemment suivant le contexte de la classe où elle se trouve

Diagrammes de classes - différentes étapes de conception

- Les diagrammes de classes peuvent représenter un système à différents niveaux d'abstraction :
  - Le point de vue **spécification** - plutôt les interfaces des classes
  - Le point de vue **conceptuel** - les concepts du domaine et les relations qui les lient
  - Le point de vue **implantation** - le contenu et l'implantation de chaque classe
- On étoffe à mesure qu'on va de hauts niveaux à de bas niveaux d'abstraction (de la spécification vers l'implantation)

Construction d'un diagramme de classes

1. Trouver les **classes du domaine étudié** ;
   Souvent, concepts et substantifs du domaine.
2. Trouver les **associations entre classes** ;
   Souvent, verbes mettant en relation plusieurs classes.
3. Trouver les **attributs des classes** ;
   Souvent, substantifs correspondant à un niveau de granularité plus fin que les classes. Les adjectifs et les valeurs correspondent souvent à des valeurs d'attributs.
4. **Organiser et simplifier** le modèle en utilisant l'héritage ;
5. **Tester** les chemins d'accès aux classées ;
6. **Itérer et raffiner** le modèle.

Conseils pratiques

- Réfléchir au diagramme de cas d'utilisation avant de commencer
- Bien nommer les classes et les relations
- Raisonner objets/classes et pas penser à l'implémentation
- Encapsulation : les informations relatives à la classe doivent être "*centralisées*" dans cette classe
- **KISS** - **K**eep **I**t **S**imple and **S**tupid
- Les modèles ne sont pas justes ou faux ; ils sont seulement plus ou moins utiles

Un bon modèle n'est pas un modèle où l'on ne peut rien ajouter, mais un modèle où on ne peut plus rien enlever