

UML : Les Concepts Objets

F.-Y. Villemin (f-yv@cnam.fr)



<http://deptinfo.cnam.fr/Enseignement/CycleSpecialisation/MAI/index.html>

Les Concepts Objets

Objet

- Classe
- Type abstrait
- Abstraction
- Encapsulation

Message

Liens

- Association
- Agrégation

Héritage

Autres Concepts

- Contraintes
- Association et attribut dérivés
- Conteneurs ("Packages")
- Interfaces

© F.-Y. Villemin 2013

2

Objets

Le **but de la modélisation objet** est de **décrire les objets**

La **description d'un objet** est une **abstraction** ayant des limites claires et un sens précis dans le contexte du problème étudié

Un **objet** possède une **identité** et peut être distingué des autres

Classe

En groupant les instances en classes, on décrit les instances par leurs propriétés générales, de manière abstraite

Dans une classe, on ne décrit qu'une fois la structure et le comportement communs d'un ensemble d'objets

Une **classe** précise :

- les **propriétés (attributs)** des instances
- le **comportement (méthodes)** des instances

Classe

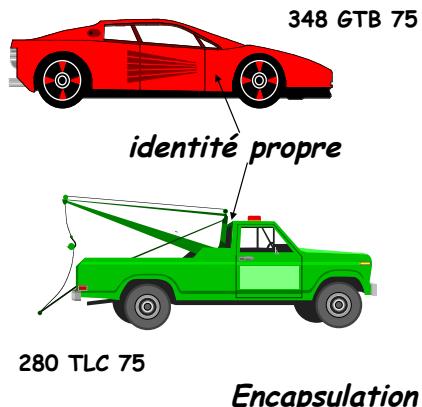
Une classe **définit** les caractéristiques et le comportement communs à un ensemble d'objets (les instances de la classe)

Une classe est une **abstraction** et une **encapsulation** des caractéristiques communes de cet ensemble d'objets

Tout objet est instance d'une classe (et d'une seule)
C-à-d, un **objet** ne peut avoir qu'**une seule définition**

Objet - Classe

Objets VEHICULE :



Classe VEHICULE :

Attributs :
immatriculation
puissance
poids
....

Méthodes :
faire le plein
immobiliser
.....

Abstraction

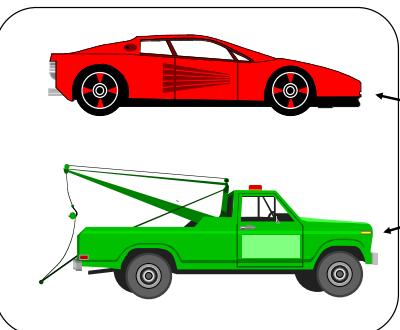
Instances

Une instance est la **concrétisation** d'une classe

Classe

VEHICULE

- Attributs :
immatriculation
puissance
poids
- Méthodes :
faire le plein
immobiliser



Instances

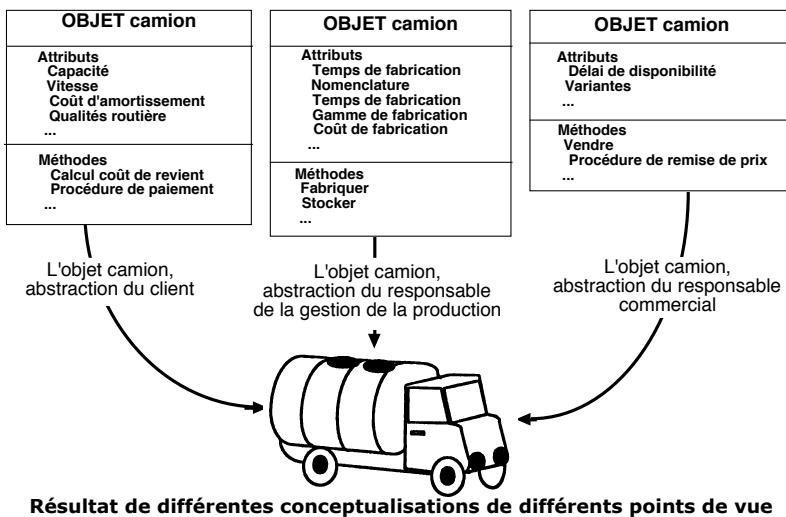
Classe

Les instances d'une **classe** partagent un objectif sémantique commun qui **dépend du point de vue de modélisation**

Exemples :

- point de vue des actifs financiers : véhicule et micro-ordinateur peuvent appartenir à une même classe
- point de vue inventaire matériel : véhicule et micro-ordinateur appartiendront à une classe différente

Classe



Les diagrammes d'objets et de classes

- Les **diagrammes d'objets** ou **diagrammes d'instances** ("object Diagram") permettent de modéliser les instances et les relations entre instances
- Les **diagrammes de classes** ("class Diagram") permettent de modéliser les classes et les relations entre classes

Classe

La **Classe** décrit :

- les méthodes (comportement)

Une méthode est une fonction pouvant comporter des paramètres et une valeur de retour

Le nom de la méthode, le type de valeur renvoyée et le type des paramètres (et les exceptions) forment la **signature** de la méthode

Méthodes particulières :

- ◆ **constructeur** (création d'instance)
- ◆ **destructeur** (suppression d'instance)

- le type des variables (structure) et valeurs initiales
- les valeurs partagées (variables de classes)

Les diagrammes de classes

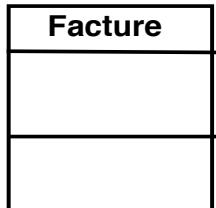
Les classes sont représentées par un rectangle
Il est possible de les représenter avec plus ou moins de "détail" :

- uniquement par leur nom
- par leur nom et noms d'attributs, et au besoin d'opérations
- de manière complète, avec indication du type des attributs, de valeurs par défaut et des signatures d'opérations

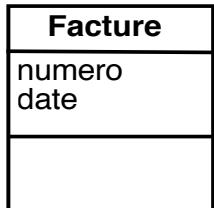
Les diagrammes de classes

Notation pour les classes

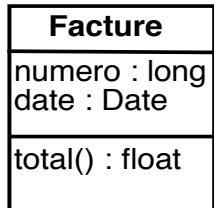
- Le **nom de la classe** commence par une **majuscule**
- Les **autres noms** débutent par une **minuscule**



Juste le nom de la classe



Le nom de la classe et des attributs



Le nom de la classe et des attributs avec leur type et les nom des méthodes avec leur signature

Les diagrammes d'instances

Les diagrammes d'instances décrivent un ensemble particulier d'objets, dans un cas donné

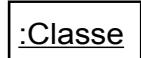
Ces diagrammes servent à expliquer des cas particuliers, et à raisonner sur la base d'exemples

Ils sont en général intégrés au **Diagrammes de Collaboration (1.X)** ou de **Communication (2.0)** (qui sont aussi des diagrammes d'instances)

Les diagrammes d'instances

Une instance peut être nommée ou anonyme

La classe de l'instance peut être indiquée ou non

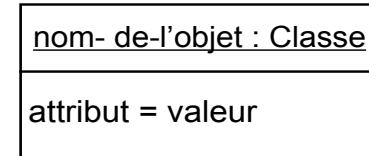


nommée

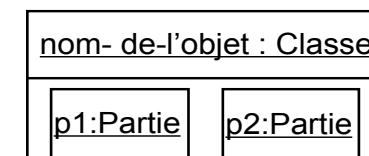
anonyme

Les diagrammes d'instances

Une instance peut être préciser la valeur d'un attribut



Une instance peut être préciser la composition d'un objet composé



Constructeur

Création de l'instance par activation du constructeur :

new <nom classe> (<paramètres>)

Les valeurs des paramètres peuvent être affectées aux attributs de l'objet par le constructeur

L'état de l'objet créé dépend :

- de la méthode de fabrication définie dans le constructeur
- des paramètres fournis au constructeur

Destructeur

L'appel du destructeur permet de supprimer l'instance:

delete(objet)

La destruction d'un objet complexe peut impliquer la destruction d'autres objets pour respecter des contraintes d'intégrités, sinon l'ensemble des instances devient **incohérent**

Type Abstrait

Un type abstrait peut être utilisé sans que l'on connaisse la manière dont sont implantées les opérations

Un type abstrait se définit indépendamment de toute implantation, uniquement en termes de propriétés et d'opérations abstraites

Certains langages, comme ADA, sont basés sur les types abstraits

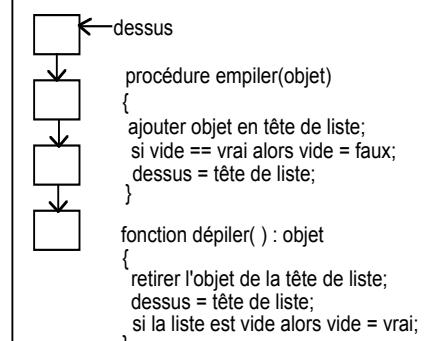
Type Abstrait

Un type abstrait de données est une structure de donnée dotée d'opérations permettant de manipuler les données de cette structure (exemple : Pile)

Structure de données :
{ dessus; /* l'objet au dessus de la pile */
vide; /* vaut vrai si pile vide */
}

Opérations :
empiler(objet);
dépiler() -> objet;

Définition du Type abstrait Pile



```
procédure empiler(objet)
{
    ajouter objet en tête de liste;
    si vide == vrai alors vide = faux;
    dessus = tête de liste;
}

fonction dépiler() : objet
{
    retirer l'objet de la tête de liste;
    dessus = tête de liste;
    si la liste est vide alors vide = vrai;
}
```

Une implantation de Pile à l'aide d'une liste Chaînée

Encapsulation

L'**encapsulation** est l'opération qui consiste à prendre un "morceau" de logiciel et à séparer ses fonctionnalités (sa "**définition abstraite**") de la manière dont elles sont implantées (son **implantation**)

Par extension, on parle d'**encapsulation**, dans un processus de modélisation, lorsque l'on spécifie un composant applicatif en **définissant son interface**

Encapsulation

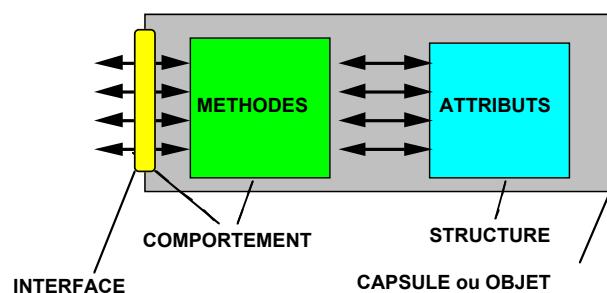
Les notions d'**Abstraction** et d'**Encapsulation** sont étroitement liées

L'**abstraction** est plus au moins liée au domaine de l'application (à la **modélisation du problème**, et donc à la phase d'analyse)

L'**encapsulation** est plus au moins liée au domaine de la **solution** (à la phase de conception)

Encapsulation

Dans la description d'un objet, le but de l'**encapsulation** est de **masquer les attributs et les méthodes**, c'est à dire, la manière dont est réalisé le comportement de l'objet



Objet

Objet :

Objet = Type Abstrait + Encapsulation + OID

OID ("Object identifier") est l'identifiant (interne) unique de l'objet

Au niveau programme, l'OID est le plus souvent un pointeur (au besoin persistant)

Encapsulation

L'**interface** de l'objet décrit les **méthodes et propriétés visibles**

Interface de Pile : décrit **dessus**, **vide**, **empiler()**, **dépiler()**, mais ne décrit pas ce qui est propre à la liste chaînée

L'implantation de l'objet est cachée par l'**encapsulation**

Ce qui n'est pas dans l'interface n'est pas visible des autres objets

Encapsulation

Les méthodes de l'interface peuvent être vues comme des **services**

Un **service** suppose un objet **Fournisseur** (du service) et un (ou plusieurs) objets **Clients** :

Un objet client fait appel, dans l'implantation de ses propres méthodes aux services d'objets fournisseurs

L'**interface** est un **contrat** qui définit le service de l'objet fournisseur

Encapsulation

L'environnement d'un programme est le contenu de la pile d'exécution de ce programme

L'environnement d'exécution d'un programme change à chaque exécution d'une instruction.

Quand une nouvelle instance d'un objet est créée à l'aide de l'instruction **new**, cette instance est ajoutée à l'environnement du programme

Quand une instance est détruite par l'instruction **delete**, cette instance est retirée de l'environnement du programme

Encapsulation

La **portée** d'une variable ou d'une fonction caractérise la portion de code source dans laquelle la variable ou la fonction existe dans l'**environnement du programme**, lorsque ce code est exécuté

Dans les langages objets, la **portée** des variables est contrainte par l'**encapsulation**

Encapsulation

On peut définir au niveau d'une classe des **attributs de classe** :

- permettent à toutes les instances d'une classe de partager le même attribut
- une variable de classe est partagée entre toutes les instances de la classe
- sa portée est celle de la classe

Exemple :

l'attribut **nb_produits** de la classe **Produit** pour désigner le nombre d'instances

Encapsulation

Portée de la variable
"mon_article"

Portée de l'instance
"mon_article"

La portée des
attributs
est celle de l'instance

```
{  
ARTICLE *mon_article;  
/* déclaration de la variable mon_article de type *ARTICLE */  
float prix; /* déclaration de la variable prix de type réel */  
....  
mon_article = new ARTICLE(124, "Eau minérale", 3.24, 0.206);  
/* création d'une instance de la classe ARTICLE */  
....  
mon_article->prix_net(); /* calcul du prix net */  
....  
delete mon_article;  
/* destruction de l'instance mon_article de classe ARTICLE */  
....  
}
```

float ARTICLE::prix_net(void)

```
{  
float resultat;  
/* déclaration de la variable locale resultat de type réel */  
resultat = prix_brut + (prix_brut * TVA);  
return resultat;  
}
```

Niveaux de visibilité

Le **niveau de visibilité** des méthodes et attributs détermine ceux qui participe de l'interface ou de l'implantation de l'objet

Les trois principaux niveaux différents de visibilité :

- **Public** : les méthodes et attributs définie au niveau public font partie de l'**interface**
- **Privé** : les méthodes et attributs définie au niveau privé font partie de l'**implantation**
- **Protégé** : niveau de visibilité intermédiaire. Les méthodes et attributs définie au niveau protégé font partie de l'**implantation**, mais seront hérités par les descendants de l'objet

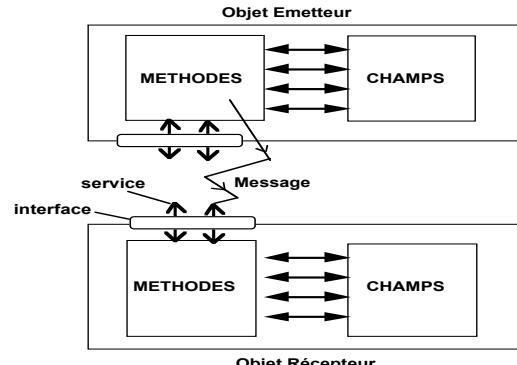
Niveaux de visibilité

Niveau de visibilité	Définition
Public noté : +	Fait partie de l'interface Visible par tout objet de l'application : peut être référencé dans n'importe quelle opération de n'importe quel objet de l'application
Protégé noté : #	Fait partie de l'implantation, transmissible par héritage aux descendants Visible uniquement par l'objet qui contient la méthode ou l'attribut, et par ses descendants, ne peut être référencé que dans les opérations de ces objets
Privé noté : -	Fait partie de l'implantation Visible uniquement par l'objet qui contient la méthode ou l'attribut : ne peut être référencé que dans les opérations de cet objet

Messages

La communication entre objets s'effectue au moyen de **messages**

Un message est l'appel d'une **méthode publique** de l'objet récepteur



© F.-Y. Villemin 2013

33

Messages

Dans le code implantant le corps de la fonction on peut :

- lire ou modifier la valeur des attributs de l'objet récepteur, ou d'attributs d'autres objets s'ils sont accessibles à partir de l'objet récepteur
- envoyer des messages à l'objet récepteur, ou à d'autres objets s'ils sont accessibles à partir de l'objet récepteur

© F.-Y. Villemin 2013

34

Messages

L'**envoi d'un message** peut modifier l'état de l'objet récepteur, ou d'autres objets

L'**envoi d'un message** peut avoir pour résultats l'**envoi d'autres messages** (réactions en chaîne)

L'**envoi d'un message** est comme la **demande d'un service** puisque l'objet émetteur ne sait pas précisément comment le calcul ou l'action sera réalisé

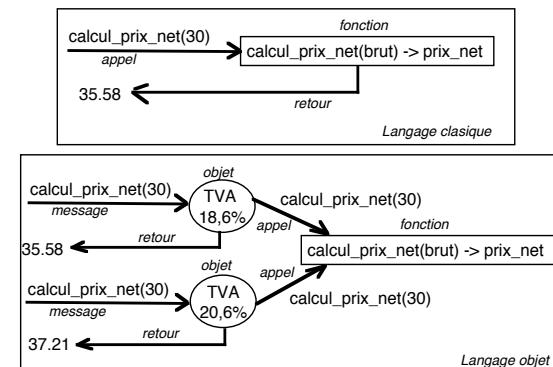
L'objet émetteur est le **client**, et l'objet récepteur est le **fournisseur du service**

© F.-Y. Villemin 2013

35

Messages

L'**envoi d'un message** est similaire à l'appel d'une fonction, mais le message est envoyé à un objet
Indirection ⇐ Le résultat dépend donc de l'objet récepteur

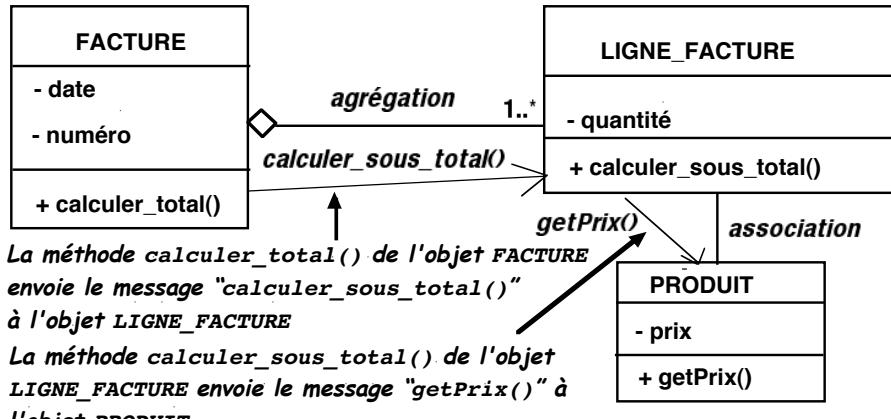


© F.-Y. Villemin 2013

36

Messages et liens

Pour qu'un objet A puisse envoyer un **message** à un objet B, il faut qu'il y ait un **lien de A vers B**



© F.-Y. Villemin 2013

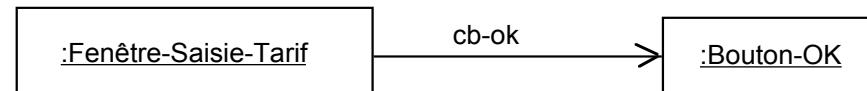
37

Liens entre objets

Les **liens entre objets** permettent d'**assurer la navigation** inter-objets et donc l'**envoi d'un message** d'un objet vers un autre

Au niveau logique, les liens sont nommés

Les liens sont **directionnels**

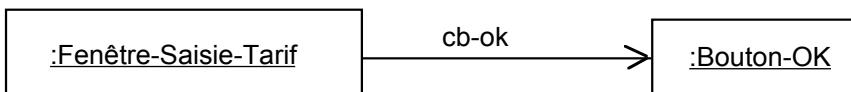


© F.-Y. Villemin 2013

38

Liens entre objets

La "**navigation**" se note par l'opérateur **".."**



Par exemple:

:Fenêtre-saisie-tarif.cb-ok pour noter
l'objet **:Bouton-OK** accédé via le lien **cb-ok**
L'expression "**:Fenêtre-saisie-tarif.cb-ok**"
constitue un chemin d'accès pour l'objet

© F.-Y. Villemin 2013

39

Liens entre objets

Pour autoriser cette navigation entre objets, un lien doit posséder les propriétés de :

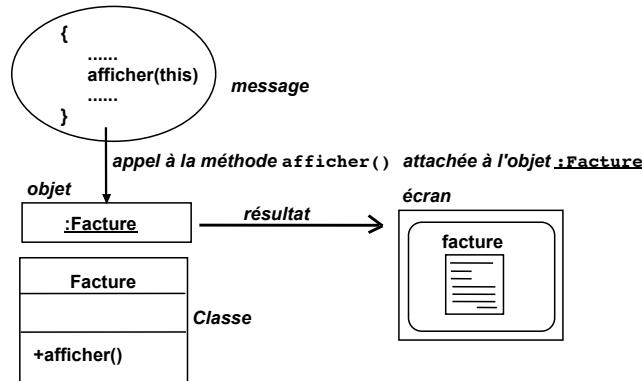
- **réflexivité**
- **anti-symétrie**
- **transitivité**

© F.-Y. Villemin 2013

40

Liens entre objets

Réflexivité : Tout objet est atteignable de lui-même
Généralement, le mot-clé "**this**" est utilisé pour désigner l'objet lui-même



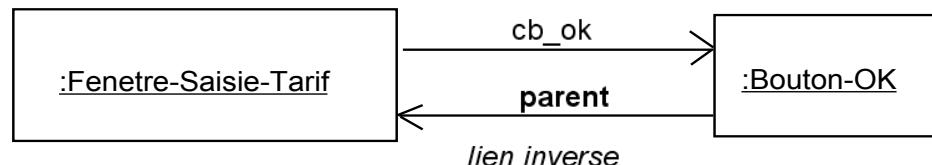
© F.-Y. Villemain 2013

41

Liens entre objets

Anti-symétrie : les liens sont uni-directionnels

Si l'on désire naviguer en sens inverse, il faut prévoir un lien "inverse" (par rapport au premier)



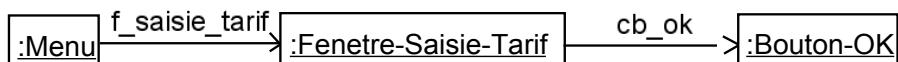
© F.-Y. Villemain 2013

42

Liens entre objets

Transitivité : On peut naviguer au travers des liens, et passer d'un objet à un autre

Exemple : "**:Menu.f_saisie_tarif.cb_ok**" permet d'atteindre l'objet "**:Bouton-OK**" de l'objet "**:Menu**"



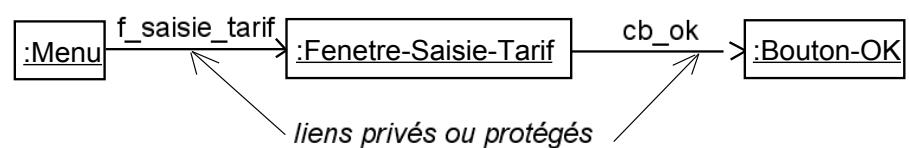
© F.-Y. Villemain 2013

43

Liens entre objets

Attention : les liens privés ou protégés peuvent limiter la navigation

Exemple : si le lien "**cb_ok**" est privé ou protégé, l'expression : "**:Menu.f_saisie_tarif.cb_ok**" n'est pas valide !



© F.-Y. Villemain 2013

44

Liens et Associations

Un lien représente une relation entre deux objets

Un lien est une connexion physique ou conceptuelle entre des instances d'objets : un lien est un tuple d'instances

Un **lien** est une instance d'**association**

Associations

Une **Association** décrit un **groupe de liens** ayant une structure et une sémantique commune

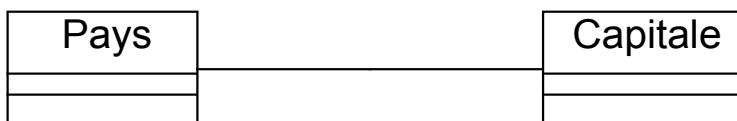
Une **association** réunit des **classes**

Tous les liens d'une association mettent en relation des objets de même classe

Les **associations** sont bidirectionnelles (binaires)

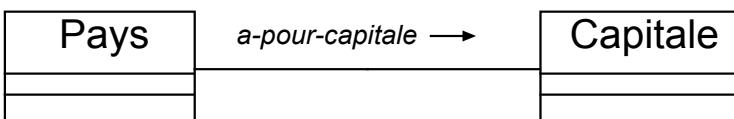
Associations

Notations pour les Associations :



Les associations peuvent être nommées :

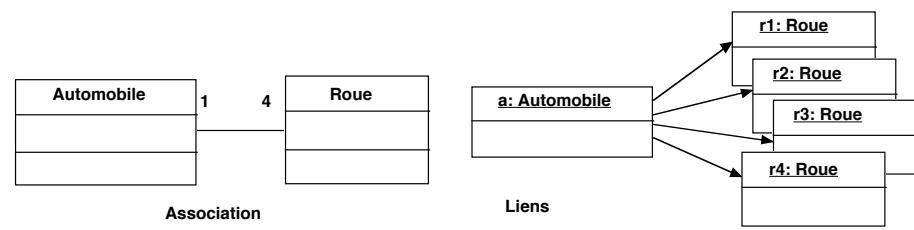
"**a-pour-capitale**" nomme l'association dans le sens "**Pays**" vers "**Capitale**"



Multiplicités des associations

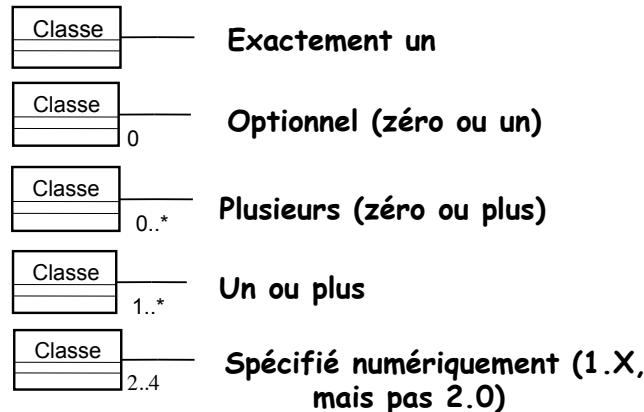
La **multiplicité** d'une association exprime le **nombre de liens** entre les instances de chaque classe de l'association

Exemple: dans l'association "**Automobile**" à "**Roue**", une instance d'**Automobile** est liée à 4 instances de **Roues**, et, une instance de **Roue** est liée à une seule instance d'**Automobile**



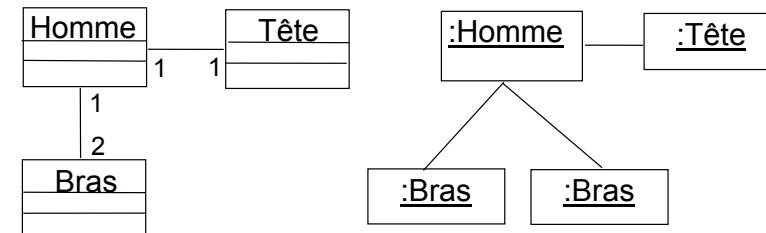
Multiplicités des associations

Multiplicités des associations :



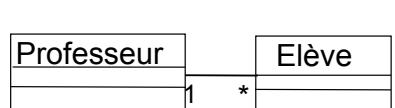
Liens et Associations

Les objets sont liés par des liens qui sont des instances des relations entre les classes :

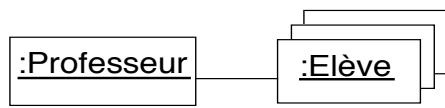


Liens et Associations

Les liens correspondants à des relations de cardinalité N entre les classes sont notés :



Modèle de classes



Modèle d'instances

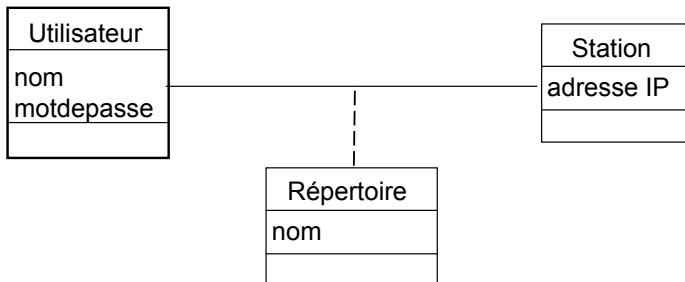
Classe-association

Les associations avec attributs de liens peuvent être modélisé sous la forme d'attributs d'une classe : l'association est considérée comme une classe ou nommée "**Classe-association**" (**Link Association**)

Dans une association modélisée sous la forme d'une "classe-association", chaque lien est une instance de cette "classe-association"

Classe-association

Modélisation d'une association en classe :
("Link Association")



Agrégation

L'**agrégation** est la forme particulière d'association entre deux classes indiquant que des instances d'une classe (par ex. `LIGNE_FACTURE`) sont contenues (ou agrégées) dans une instance d'une autre classe (par ex. `FACUTE`)

Dans l'**agrégation** les deux objets en relation sont distingués : l'un est un **composant** de l'autre

Au niveau logique, on considère que le **composé** est responsable de la gestion de ses **composants** (c'est le composé qui crée, modifie, ou détruit ses composants)

Agrégation

L'**agrégation** est la forme particulière d'association entre deux classes indiquant que des instances d'une classe sont contenues (ou agrégées) dans une instance d'une autre classe

Une **agrégation** est une relation "**composé-composant**" ou "**partie-de**"

Un des objets participant à l'**agrégation** est un **composé**, un assemblage de **composants** ou de parties

Exemple, une facture est un composé de lignes factures

Agrégation

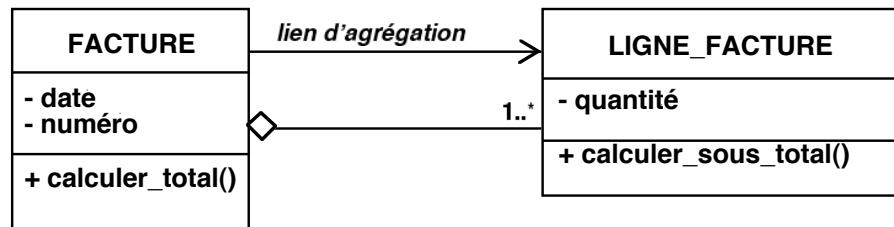
L'**agrégation** peut être **transitive** : un objet composant peut lui même être un objet composé

Exemple, une automobile se compose (entre autres) d'un bloc-moteur et d'un châssis. Le bloc-moteur se compose d'une boîte de vitesse, d'un carburateur...

L'**agrégation** n'est ni **symétrique** ni **bi-directionnelle** : car l'**agrégation** distingue un **composé** d'un **composant**

Agrégation

Notation pour une **agrégation** (en phase d'analyse) :



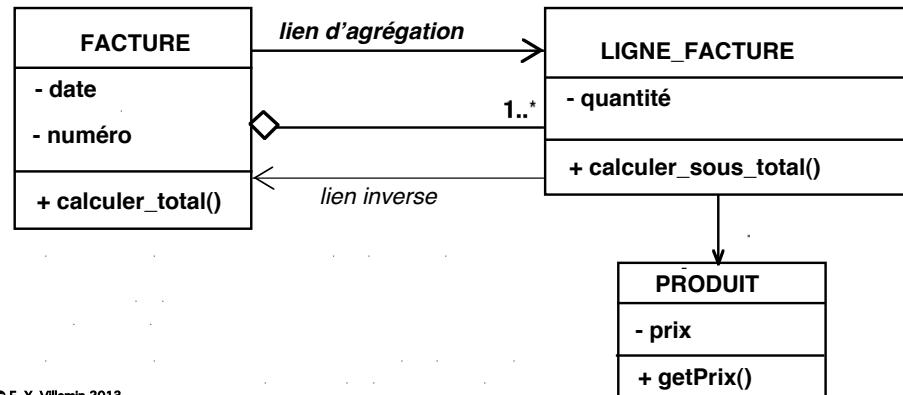
© F.-Y. Villemin 2013

57

Agrégation

L'**agrégation** n'est pas porteuse de sens

L'**agrégation** est orientée du **composé** vers le **composant** (création d'un lien inverse si nécessaire)



© F.-Y. Villemin 2013

58

Agrégation

L'agrégation est une association dotée d'une sémantique additionnelle

Une agrégation est :

- **transitive** (relation "partie-de")
- **antisymétrique** (à la différence de l'association qui est bidirectionnelle)

© F.-Y. Villemin 2013

59

Agrégation

Reconnaître une agrégation

Une association est une agrégation si :

- Peut-on utiliser l'expression partie-de ?
- Des opérations appliquées sur le composé sont-elles appliquées automatiquement aux composants (le composé gère ses composants) ?

Exemple : totaliser une facture

© F.-Y. Villemin 2013

60

Agrégation

Reconnaître une agrégation

Des valeurs d'attributs sont-elles propagées du composé vers des composants

Exemple : la date d'une facture "concerne" les lignes factures

- Y a-t-il une asymétrie intrinsèque dans l'association dans laquelle une classe d'objet est subordonnée à l'autre ?

Objet Complexe

Un **objet complexe** (par opposition, un objet "**simple**" ou **non complexe**) est un objet dont au moins un attribut a pour valeur l'**oid** d'un autre objet

Un **schéma d'objets** est un ensemble d'objets (d'instances), simples ou complexes, reliés entre eux par des valeurs d'oid contenues dans des attributs

Objet Complexe

ECHELON
n_échelon : entier
indice : entier

Objet "simple"

GRADE
code : String
libelle : String
categorie : String
grille : tableau de ECHELON

Objet complexe

AGENT

nom : String
prenom : String
grade : GRADE
echelon : ECHELON
age() : entier

Objet complexe

Objet Complexe

AGENT
matricule : alpha
nom : alpha
prénom : alpha

GRADE
code : alpha
libellé : alpha
catégorie : alpha

1..*

1

ECHELON
indice : entier
n° échelon : entier

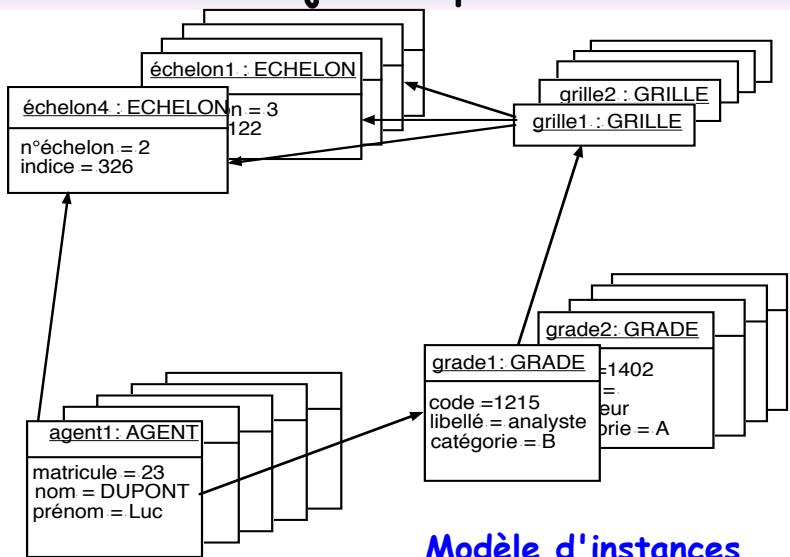
1..*

1

GRILLE
libellé : alpha
nb échelon : entier

Modèle de classes

Objet Complexe



© F.-Y. Villemin 2013

65

Modèle d'instances

Cohérence d'un schéma objet

Un schéma est cohérent (ou consistant) si :

- Il n'existe pas d'objets distincts ayant le même oid
- Pour chaque oid référencé dans le schéma, il existe un objet possédant cet oid (pas de référence "folle")
- Deux objets seront équivalents ou égaux s'ils possèdent le même oid

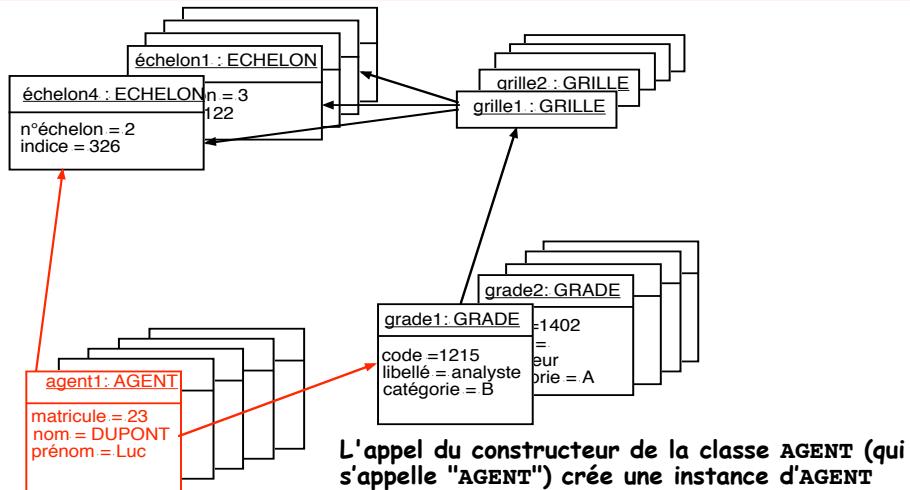
Exemple : l'échelon de "Agent1" est l'échelon de "Agent3" = Echelon1

- Les contraintes d'intégrités sont respectées

© F.-Y. Villemin 2013

66

Cohérence d'un schéma objet



© F.-Y. Villemin 2013

67

Cohérence d'un schéma objet

Les constructeurs et destructeurs doivent assurer :

- la cohérence du schéma
- le maintien des contraintes d'intégrité

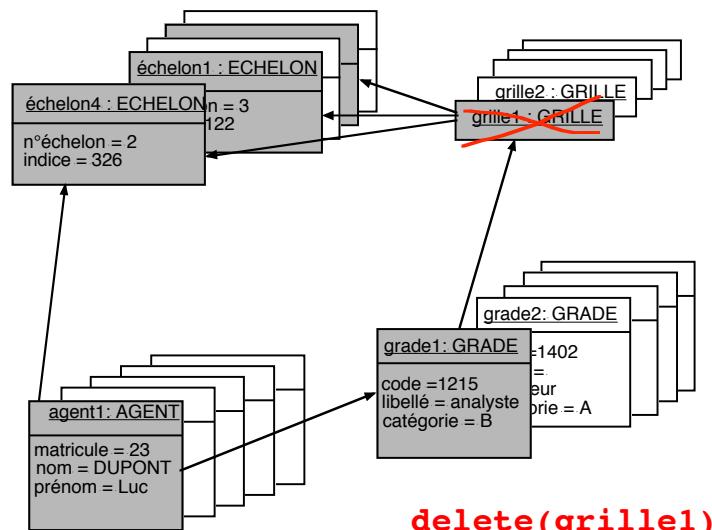
L'appel du destructeur permet de supprimer l'instance : **delete(objet)**

- La destruction d'un objet complexe peut impliquer la destruction d'autres objets pour respecter des contraintes d'intégrités, sinon le schéma d'objet complexe devient **incohérent**

© F.-Y. Villemin 2013

68

Cohérence d'un schéma objet



© F.-Y. Villemain 2013

69

Héritage

L'**héritage** est la transmission de caractéristiques à ses descendants

L'**héritage** correspond à la relation "est-un" :

- Un **CONDUCTEUR** est une **PERSONNE**
- Un **CONDUCTEUR** est une spécialisation de **PERSONNE**
- Une **PERSONNE** est une généralisation d'un **CONDUCTEUR**

© F.-Y. Villemain 2013

70

Héritage

La **classe qui hérite** dispose :

- Des **méthodes** de niveau **public et protégé**
- Des **attributs** de niveau **public et protégé**

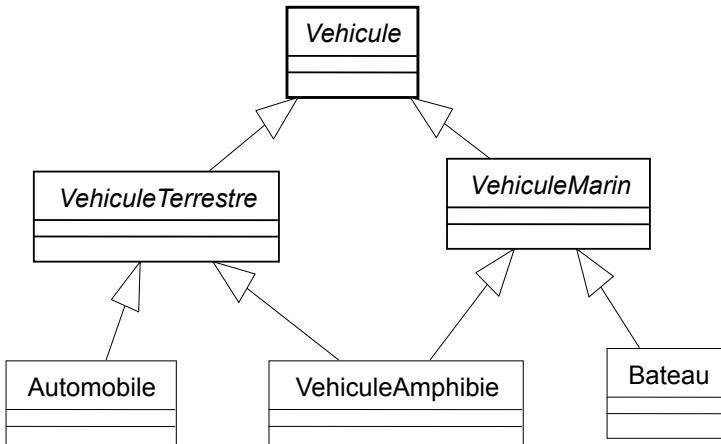
Par contre les **attributs et méthodes de niveau privé** ne sont pas hérités

© F.-Y. Villemain 2013

71

Héritage

Notation pour l'**héritage** :

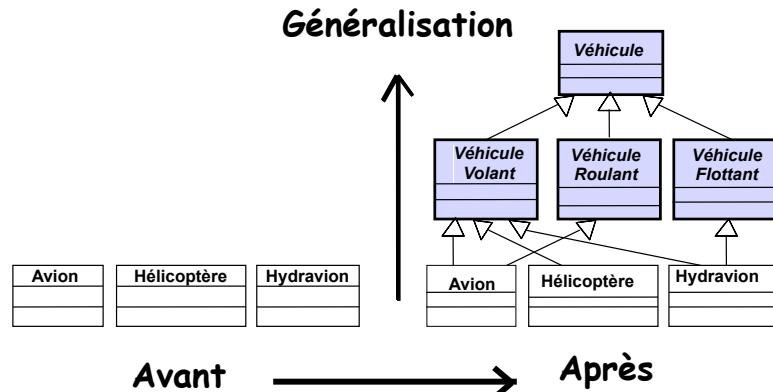


© F.-Y. Villemain 2013

72

Héritage : Généralisation

La **généralisation** est l'abstraction des caractéristiques communes à plusieurs classe d'objets

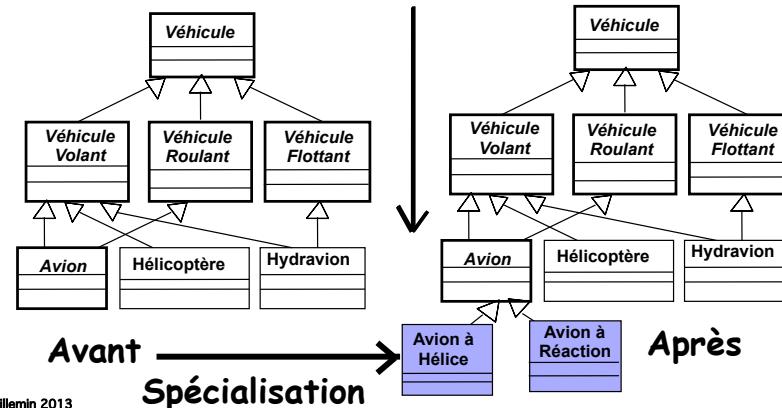


© F.-Y. Villemin 2013

73

Héritage : Spécialisation

La **spécialisation** est la dérivation de nouvelles abstractions contenant des caractéristiques supplémentaires



© F.-Y. Villemin 2013

74

Héritage : Définition

Une classe définit un **type**

- La classe CONDUCTEUR héritant de la classe PERSONNE, le type CONDUCTEUR est un **sous-type** du type PERSONNE

Pour que l'affectation d'un sous-type à un type soit valide il faut que :

- une instance de CONDUCTEUR soit aussi une instance de PERSONNE
- l'ensemble des instances de CONDUCTEUR soit inclus dans l'ensemble des instances de PERSONNE

Héritage : Définition

Une classe B **hérite** d'une classe A si l'ensemble des instances de la classe B peut être inclus dans l'ensemble des instances de la classe A

- la classe CONDUCTEUR héritera de la classe PERSONNE si l'ensemble des instances de CONDUCTEUR peut être inclus dans l'ensemble des instances de PERSONNE

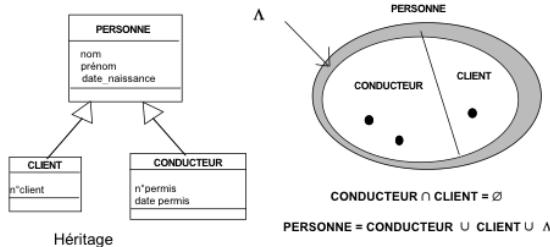
© F.-Y. Villemin 2013

75

© F.-Y. Villemin 2013

76

Héritage - Définition



Si le sous-ensemble Δ est vide, alors PERSONNE est une classe abstraite

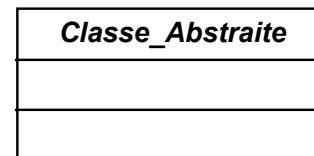
L'intersection entre CONDUCTEUR et CLIENT est nécessairement vide, car un objet est instance d'une seule classe

Les Classes Abstraites

Une classe abstraite est une classe qui ne possède pas d'instances mais dont les sous-classes possèdent des instances directes

Une classe abstraite sert à définir des méthodes qui factorisent un comportement, ou pour factoriser des structures (ensembles d'attributs) communes à plusieurs classes

Le non d'une classe abstraite est écrit en italique :



Héritage

Une instance d'une classe est instance de tous les ancêtres de sa classe

- Une instance hérite de tous les attributs de ses classes ancêtres
- Toute opération définie dans toute classe ancêtre est applicable sur une instance, à une surcharge près

Surcharge

La surcharge est la réutilisation d'un nom pour désigner un élément de logiciel (généralement une fonction)

But : Diminuer le nombre de noms à gérer par le programmeur, et donc la complexité des programmes

Surcharge

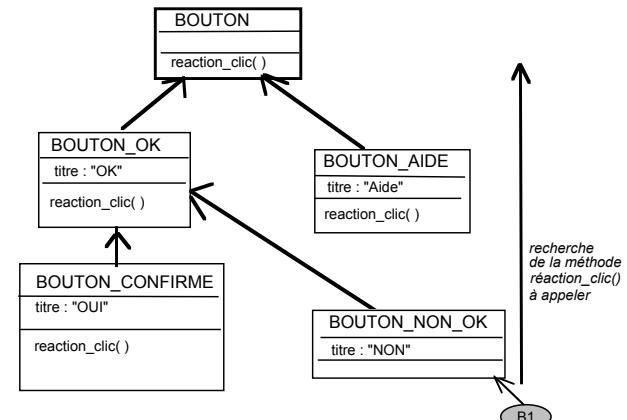
Une classe descendante peut redéfinir un attribut (valeurs initiales) ou une opération d'une classe ancêtre

Une classe descendante ne peut pas supprimer un attribut

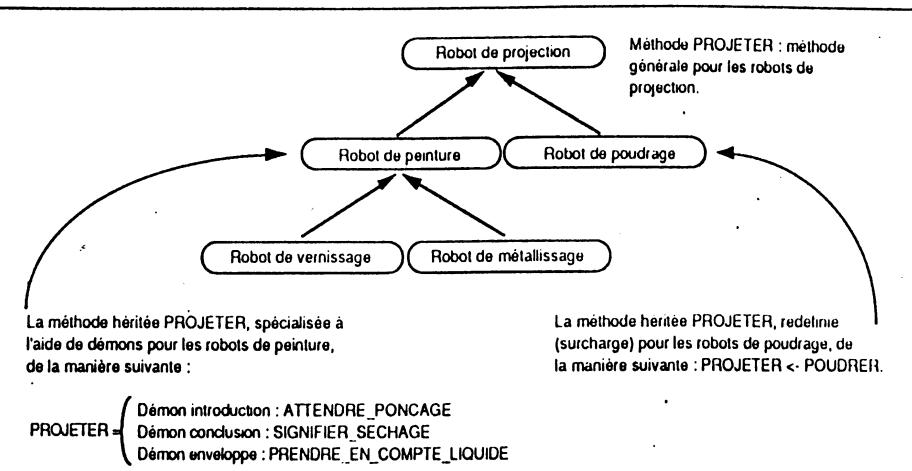
Une sous-classe peut réimplanter une opération, mais ne peut pas changer sa **signification** (visible de l'interface)

Surcharge

Exemple : la méthode "reaction_clic" des classes "BOUTONS"



Surcharge



Surcharge

Suivant les langages, la signature de la méthode qui surcharge doit respecter certaines règles :

- **covariance** : Les types de la signature de la méthode surchargée peuvent être plus spécialisés que les types de la méthode héritée
- **contra-variance** : Les types de la signature de la méthode surchargée peuvent être moins spécialisés que les types de la méthode héritée
- **non-variance** : Les types de la signature de la méthode surchargée doivent être les mêmes que les types de la méthode héritée

Surcharge

- **covariance** : typage non sûr, mais intuitif
- **contra-variance** : typage sûr, mais contre-intuitif
- **non-variance** : typage sûr, mais peu de souplesse

C++ est non-variant, mais autorise la covariance sous responsabilité du programmeur (cast + contrôle dynamique du type à l'exécution)

Polymorphisme

Suivant que l'objet récepteur est déterminé :

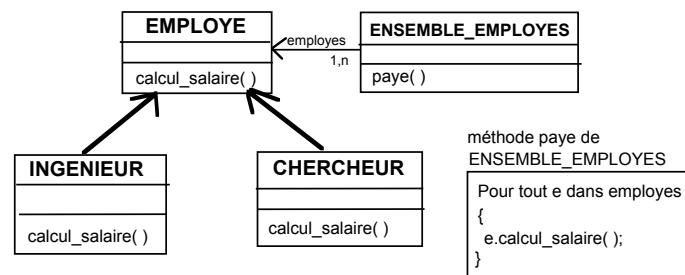
- à la compilation : édition statique (classique) des liens
- à l'exécution : édition retardée (dynamique) des liens

L'édition retardée permet le polymorphisme :

- suivant l'objet contenu dans une variable, une méthode (ou une autre) sera exécutée

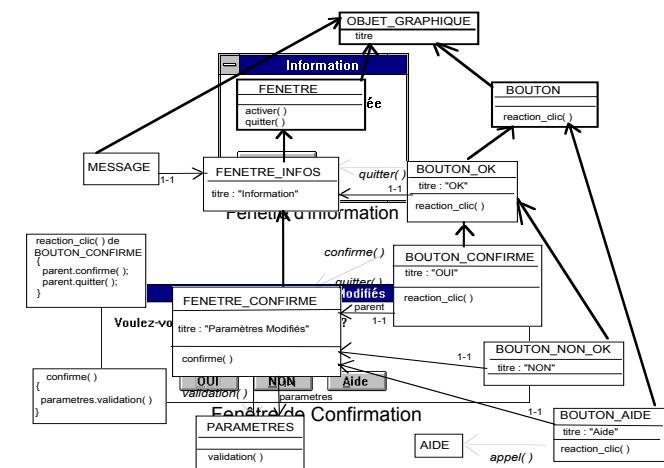
Polymorphisme

Le **polymorphisme** est réalisé grâce à l'édition retardée des liens de méthodes surchargées : à l'exécution, le programme appelle la fonction "**calcul_salaire()**" correspondant au type de l'objet contenu dans la variable e



Héritage fortuit

Exemple d'utilisation de l'héritage :



Héritage

L'héritage conceptuel correspond à la relation **est-un** :

- Un CONDUCTEUR est une PERSONNE
- Un CONDUCTEUR est une spécialisation de PERSONNE
- Une PERSONNE est une généralisation d'un CONDUCTEUR

L'héritage (conceptuel) doit être distingué de l'héritage fortuit

Héritage fortuit

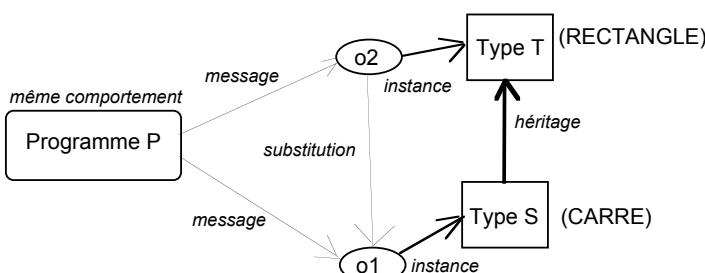
Pour l'héritage (conceptuel) permet la réutilisation de classes. Pour cela, on distingue :

- l'héritage avec ajout seul (la classe dérivée contient uniquement des attributs ou méthodes supplémentaires par rapport à sa super classe)
- l'héritage de classes abstraites, dont les méthodes ont une sémantique minimale ou "ne font rien"
- l'héritage combinant les deux avec utilisation de la surcharge

Héritage fortuit

Principe de substitution de B. LISKOV (Data abstraction hierarchy in SIGPLAN Notices 23. May 1988) :

Si pour tout objet o_1 de type S , il existe un objet o_2 de type T tel que pour tous les P définis en termes de T , le comportement de P reste inchangé quand o_1 est remplacé par o_2 , alors S est un sous type de T



Héritage fortuit

L'**Héritage fortuit** est un héritage qui ne respecte pas la règle de substitution de Liskov

Il correspond à la relation **est-comme-un** :

- fondé sur une analogie (ex. une fenêtre de confirmation "est comme une" fenêtre d'information)
- dangereux (où commence et finit l'analogie ?)
- permet de réutiliser des "bouts" de code

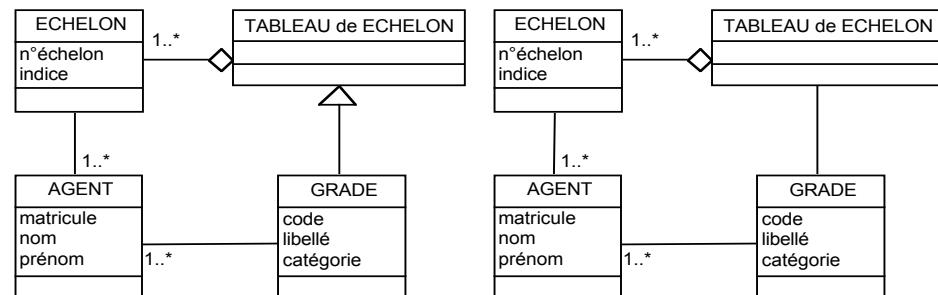
Héritage fortuit

La **délégation** est un mécanisme qui permet à un objet qui reçoit un message de le rediriger vers un autre objet (ou de déléguer le service demandé à un autre objet)

Plutôt qu'utiliser un héritage fortuit, il faut lui préférer l'utilisation d'un mécanisme de **délégation**

Héritage fortuit

Plutôt que d'utiliser un héritage entre deux objets A et B (relation A **est-comme-un** B), on établit une association entre A et B.

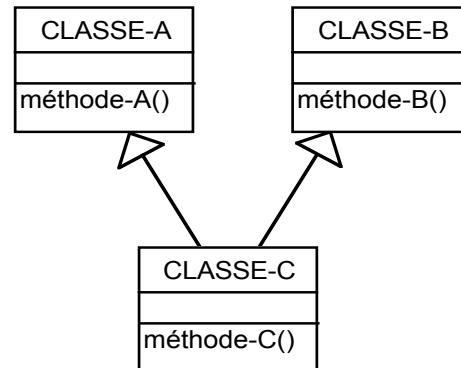


héritage fortuit

délégation

Héritage multiple

C'est la capacité, pour une classe dérivée, d'hériter de plusieurs classes de base

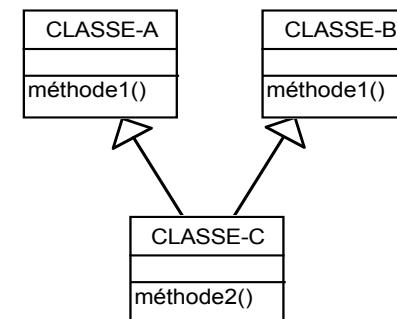


La classe CLASSE-C hérite des méthodes méthode-A() et méthode-B() des CLASSE-A et CLASSE-B

Héritage multiple

Il y a **conflict** de nom quand une classe dérivée hérite plusieurs fois d'un même nom

Problème : Désigner l'élément logiciel hérité dans la classe dérivée



Exemple : conflit sur la méthode1 héritée par CLASSE_C

Héritage multiple

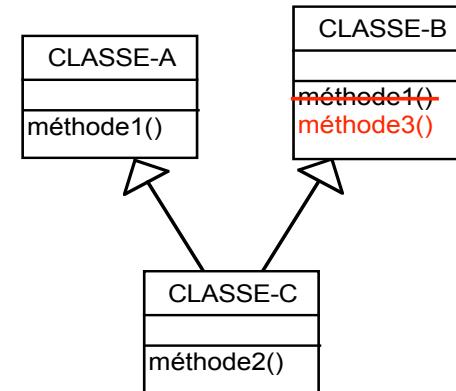
Les conflits doivent être résolus par le programmeur
(le compilateur doit savoir de quel objet on parle)

Il existe plusieurs techniques :

- résolution par renommage
- résolution par qualification
- résolution par surcharge
- résolution par recouvrement

Héritage multiple

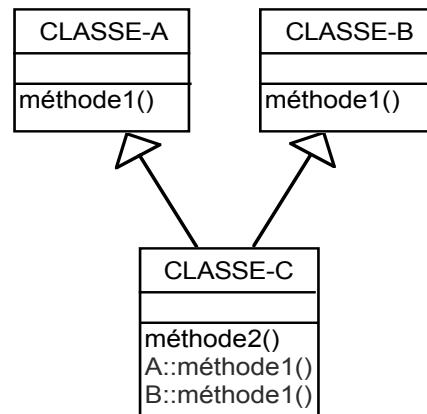
Résolution par renommage : multiplicité des noms, donc perte des avantages de la surcharge



Renommage de méthode1() en méthode3()

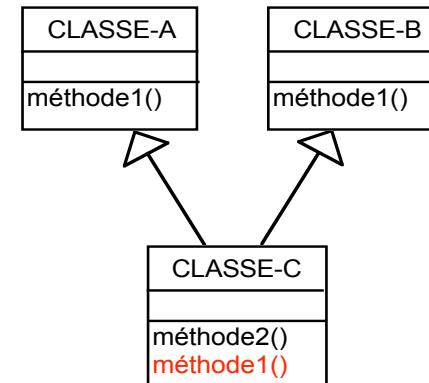
Héritage multiple

Résolution par qualification : l'ambiguïté est levée, mais revient à une multiplicité des noms



Héritage multiple

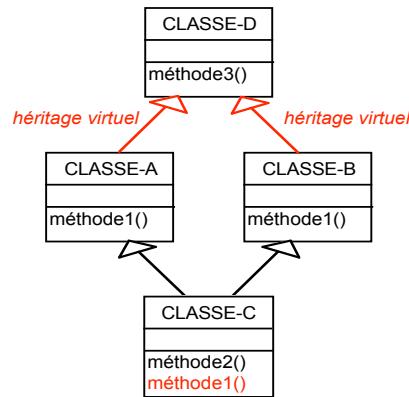
Résolution par surcharge : pas de multiplicité des noms, l'ambiguïté est levée



La méthode1() est redéfinie dans la CLASSE-C

Héritage multiple

Résolution par recouvrement : utilisable avec un type d'héritage particulier, uniquement dans le cas du problème du "losange"



La méthode1() est redéfinie dans la CLASSE-C.

© F.-Y. Villemin 2013

101

Héritage

Les descripteurs de classes sont de deux sortes :

- Les attributs de classes : décrivent une valeur commune à toutes les instances de la classe
- Les opérations de classes : c'est une opération portant sur la classe elle-même (comme la création d'une instance), ou une opération portant sur l'ensemble des instances (comme une requête)

© F.-Y. Villemin 2013

102

Héritage

Notation pour les descripteurs de classe : les attributs de classes ont un nom préfixé par le symbole "\$"

Fenetre
titre
\$ nbFenetresActives
open ()
créer ()
fermerToutes ()

© F.-Y. Villemin 2013

103

Contraintes

On distingue :

- Les contraintes sur un objet
- Les contraintes sur un lien
- Les contraintes entre liens

Les contraintes simples peuvent être définies dans le modèle objet

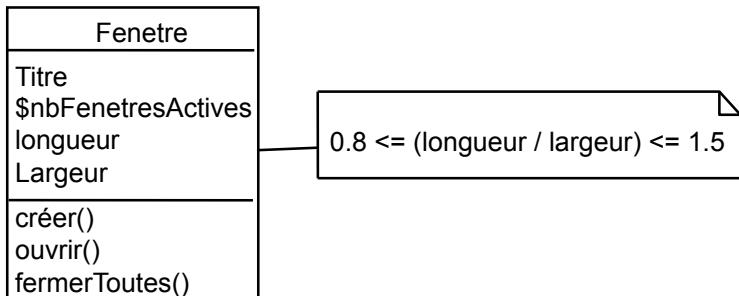
Les contraintes complexes doivent être définies dans le modèle fonctionnel

© F.-Y. Villemin 2013

104

Contraintes

Les **contraintes sur un objet** sont des contraintes qui portent sur des valeurs d'attributs d'une même instance, elles sont notées à l'aide d'une étiquette

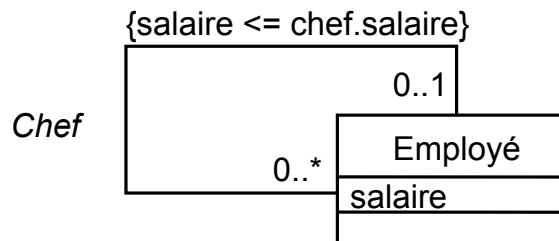


© F.-Y. Villemin 2013

105

Contraintes

Les **contraintes sur un lien** sont des contraintes qui portent sur des valeurs d'attributs d'instances différentes reliées par un lien



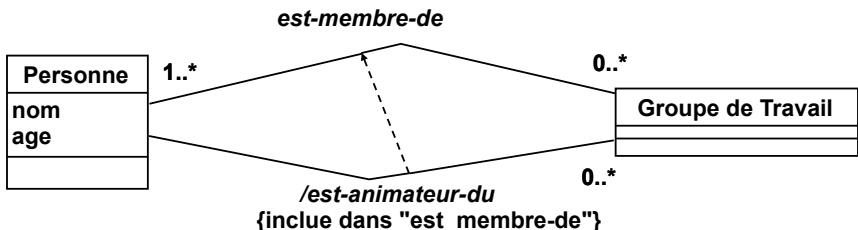
© F.-Y. Villemin 2013

106

Contraintes

Les **contraintes entre liens** s'expriment en langage naturel, ou par des équations

Une flèche pointillée indique une dépendance



© F.-Y. Villemin 2013

107

Association et attribut dérivés

Une **association dérivée** est un sous-ensemble de liens de l'association qui est dérivée

La notation pour une association dérivée est une barre oblique sur le trait

L'association dérivée doit être définie par la **contrainte**, l'équation ou la requête qui la détermine

© F.-Y. Villemin 2013

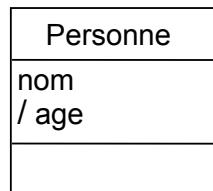
108

Association et attribut dérivés

Un **attribut dérivé** est le résultat d'une fonction ou d'un calcul

L'attribut dérivé est noté en utilisant le caractère '/' devant son nom

Exemple : l'attribut **age** d'une Personne

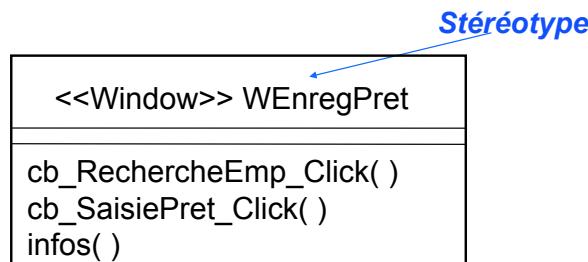


Stéréotype

Un **stéréotype** est une métaclassification des classes

Permet d'étendre la notation standard d'UML

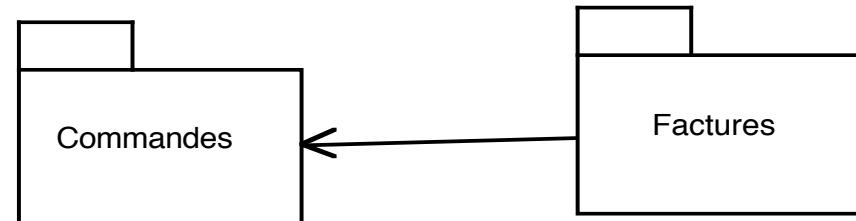
Une classe correspondant à un acteur du système d'information est du stéréotype <>Actor<>



Conteneurs

Les **conteneurs (packages)** servent à organiser un modèle objet en domaines et sous-domaines

Cette notion est purement organisationnelle. Elle ne possède aucune sémantique dans la définition du modèle



Factures utilise l'interface d'objets de **Commandes**

Interface

Une **interface** est une spécification des opérations visibles d'une classe ou de toutes autres éléments, tels que des paquetages

Une **interface** ne spécifie souvent qu'une partie limitée du comportement de la classe

