

# Conception Orientée Objet

## Introduction à l'UML

Valentin Montmirail

[valentin.montmirail@unice.fr](mailto:valentin.montmirail@unice.fr)

Université Côte d'Azur, CNRS, I3S, Nice, France



# Bibliographie UML

- ▶ “UML 2.0, guide de référence”
  - ▶ James Rumbaugh, Ivar Jacobson, Grady Booch
  - ▶ CampusPress, 2004
- ▶ “UML 2.0”
  - ▶ Benoît Charoux, Aomar Osmani, Yann Thierry-Mieg
- ▶ “UML 2.0 Superstructure” et “UML 2.0 Infrastructure”
  - ▶ OMG (Object Management Group)
  - ▶ [www.uml.org](http://www.uml.org) (2004).



# Bibliographie OCL

- ▶ “UML 2.0 Object Constraint Language (OCL) ”
  - ▶ **OMG (Object Management Group)**
  - ▶ [www.uml.org](http://www.uml.org) (2004)
- ▶ Cours de Jean-Marie Favre, IMAG
  - ▶ <http://megaplanet.org/jean-marie-favre>



# Matériel et logiciel

- ▶ Systèmes informatiques :
  - ▶ **80 % de logiciel** ;
  - ▶ 20 % de matériel.
- ▶ Depuis quelques années, la fabrication du matériel est assurée par quelques fabricants seulement.
  - ▶ Le matériel est relativement fiable.
  - ▶ Le marché est standardisé.

Les problèmes liés à l'informatique sont essentiellement des problèmes de logiciel.



# La “crise du logiciel”

- ▶ Étude sur 8 380 projets (Standish Group, 95) :
  - ▶ Succès : 16 % ;
  - ▶ Problématique : 53 % (budget ou délais non respectés, défaut de fonctionnalités) ;
  - ▶ Échec : 31 % (abandonné).

Le taux de succès décroît avec la taille des projets et la taille des entreprises.

- ▶ **Génie Logiciel (Software Engineering)** :
  - ▶ Comment faire des logiciels de qualité ?
  - ▶ Qu'attend-on d'un logiciel ? Quels sont les critères de qualité ?

# Critères de qualité d'un logiciel

## ► Utilité

- Adéquation entre le logiciel et les besoins des utilisateurs ;

## ► Utilisabilité

## ► Fiabilité

## ► Interopérabilité

- Interactions avec d'autres logiciels ;

## ► Performance

## ► Portabilité

## ► Ré-utilisabilité

## ► Facilité de maintenance

- Un logiciel ne s'use pas pourtant, la maintenance absorbe une très grosse partie des efforts de développement.

# Etapes du développement

- ▶ Étude de faisabilité
- ▶ Spécification
  - ▶ Déterminer les fonctionnalités du logiciel.
- ▶ Conception
  - ▶ Déterminer la façon dont le logiciel fournit les différentes fonctionnalités recherchées.
- ▶ Implantation
- ▶ Tests
  - ▶ Essayer le logiciel sur des données d'exemple pour s'assurer qu'il fonctionne correctement.
- ▶ Maintenance

Le coup de la maintenance absorbe une grande partie du coût de développement.

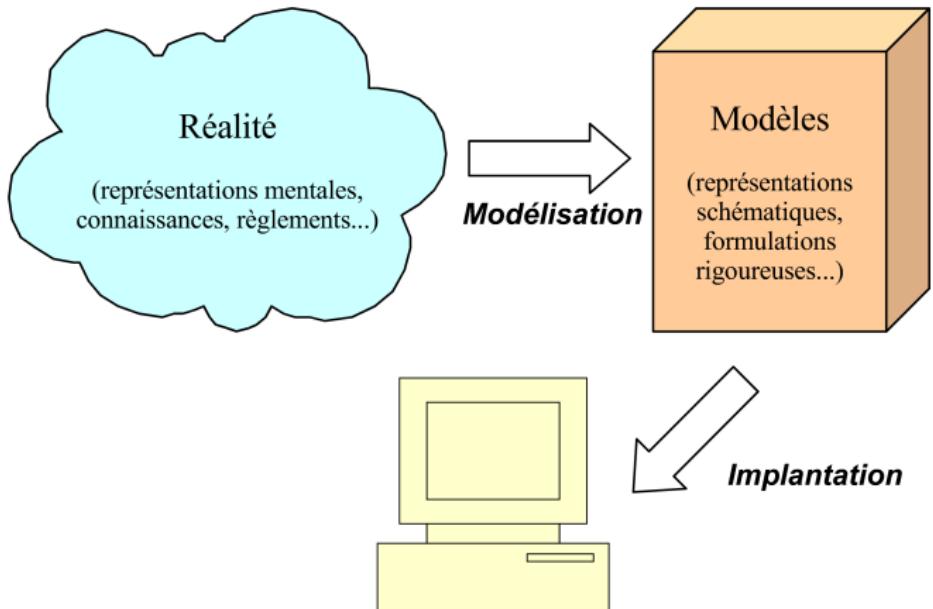
# Poids de la maintenance

	Répartition effort dév.	Origine des erreurs	Coût de la maintenance
<b>Spécification</b>	6%	56%	82%
<b>Conception</b>	5%	27%	13%
<b>Implantation</b>	7%	7%	1%
<b>Intégration - Tests</b>	15%	10%	4%
<b>Maintenance</b>	67%		

(Zeltovitz, De Marco)

La réduction du coup de maintenance se joue en amont.

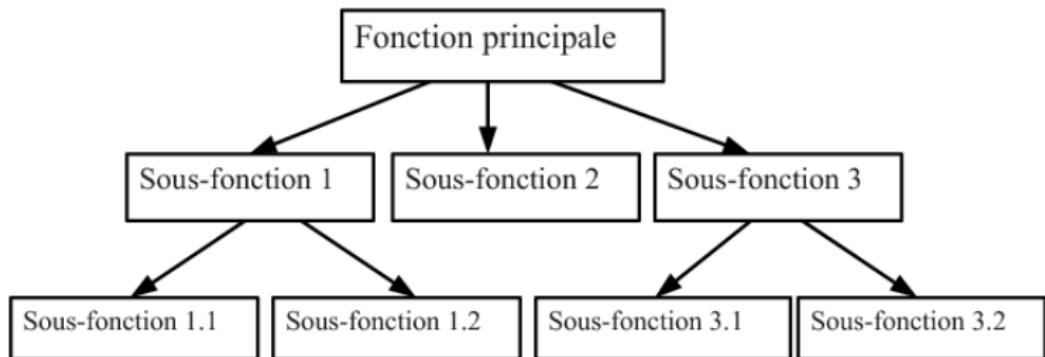
# Modélisation



► Approche **descendante** :

- Décomposer la fonction globale jusqu'à obtenir des fonctions simples à appréhender et donc à programmer.

C'est la **fonction** qui donne la **forme** du système.



- ▶ La **Conception Orientée Objet** (COO) est la méthode qui conduit à des architectures logicielles fondées sur les objets du système, plutôt que sur une décomposition fonctionnelle.

C'est la **structure** du système lui donne sa forme.

- ▶ On peut partir des objets du domaine (briques de base) et remonter vers le système global : **approche ascendante**.

Attention, l'approche objet n'est pas seulement ascendante.

# Unified Modeling Language

- ▶ Dans les 90, OOSE et OMT veulent créer un langage de modélisation uniifié avec pour objectifs :
  - ▶ Modéliser un système **des concepts à l'exécutable**, en utilisant les techniques orientée objet ;
  - ▶ **Réduire la complexité de la modélisation** ;
  - ▶ Utilisable par **l'homme comme la machine** :
    - ▶ Représentations graphiques mais disposant de qualités formelles suffisantes pour être **traduites automatiquement en code source** ;
    - ▶ Ces représentations ne disposent cependant pas de qualités formelles suffisantes pour justifier d'aussi bonnes propriétés qu'ont les langages de spécification formelle (Z, VDM...).

## UML v2.0 date de 2005

Il s'agit d'une **version majeure** apportant des innovations radicales et étendant largement le champ d'application d'UML.

## Conception objet élémentaire avec UML

Diagrammes de cas d'utilisation

Diagrammes de classes

Diagrammes d'objets

Diagrammes de séquences

## UML et méthodologie

## Modélisation avancée avec UML

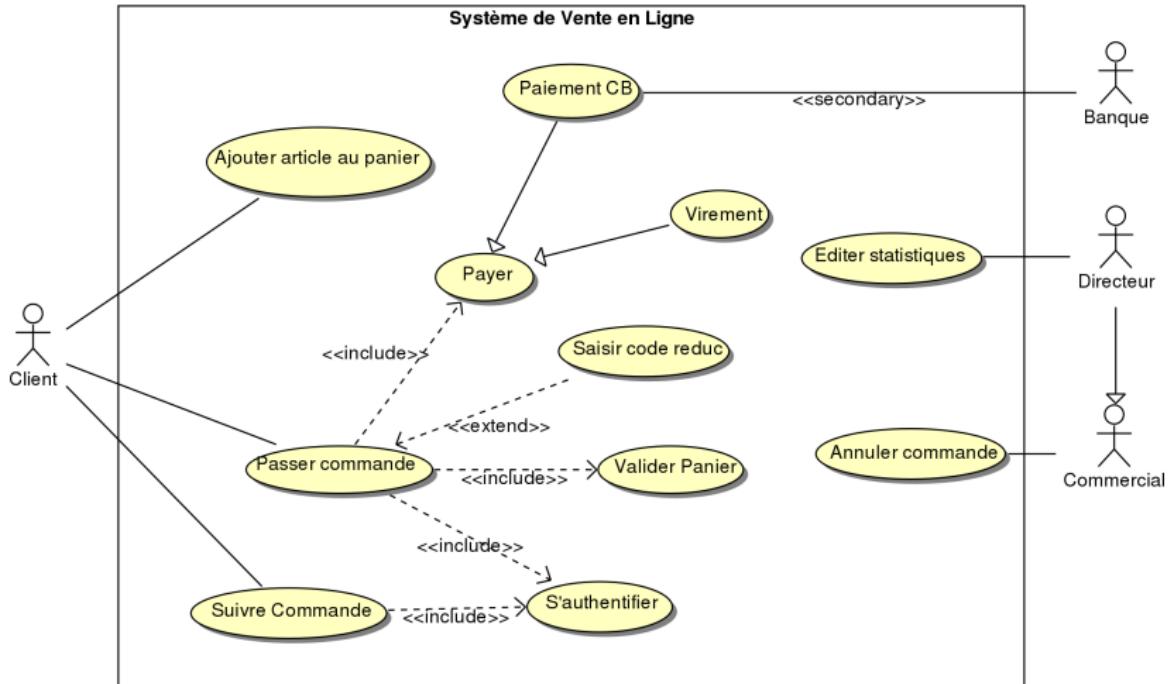
## Bonnes pratiques de la modélisation objet



Avant de développer un système, il faut savoir **précisément** à **QUOI** il devra servir, *cad* à quels besoins il devra répondre.

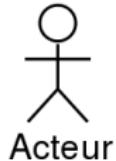
- ▶ **Modéliser les besoins** permet de :
  - ▶ Faire l'inventaire des fonctionnalités attendues ;
  - ▶ Organiser les besoins entre eux, de manière à faire apparaître des relations (réutilisations possibles, ...).
- ▶ Avec UML, on modélise les besoins au moyen de **diagrammes de cas d'utilisation**.

# Exemple de diagramme de cas d'utilisation



## Cas d'utilisation

- ▶ Un **acteur** est une entité extérieure au système modélisé, et qui interagit directement avec lui.

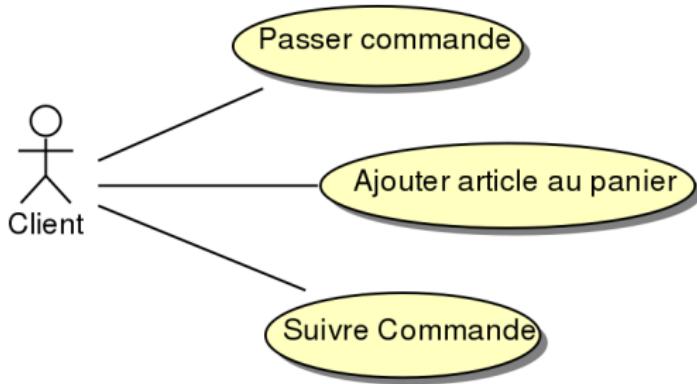


Acteur

- ▶ Un **cas d'utilisation** est un service rendu à un acteur, il nécessite une série d'actions plus élémentaires.

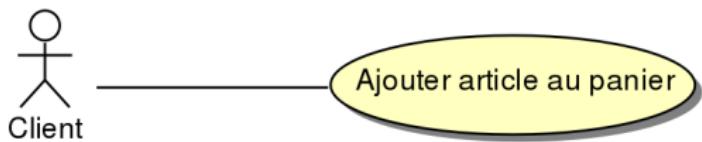
A yellow oval containing the text "Cas d'utilisation".

Un cas d'utilisation est l'expression d'un service réalisé de bout en bout, avec un déclenchement, un déroulement et une fin, pour l'acteur qui l'initie.



# Relations entre cas d'utilisation en acteurs

- ▶ Les acteurs impliqués dans un cas d'utilisation lui sont liés par une **association**.
- ▶ Un acteur peut utiliser plusieurs fois le même cas d'utilisation.

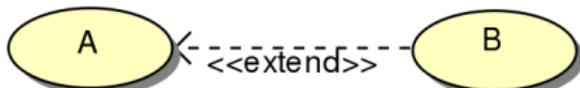


# Relations entre cas d'utilisation

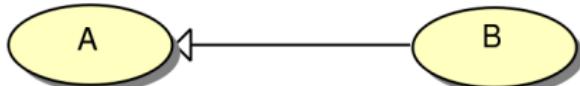
- ▶ **Inclusion** : le cas A inclut le cas B (B est une partie *obligatoire* de A).



- ▶ **Extension** : le cas B étend le cas A (B est une partie *optionnelle* de A).



- ▶ **Généralisation** : le cas A est une généralisation du cas du cas B (B est une sorte de A).

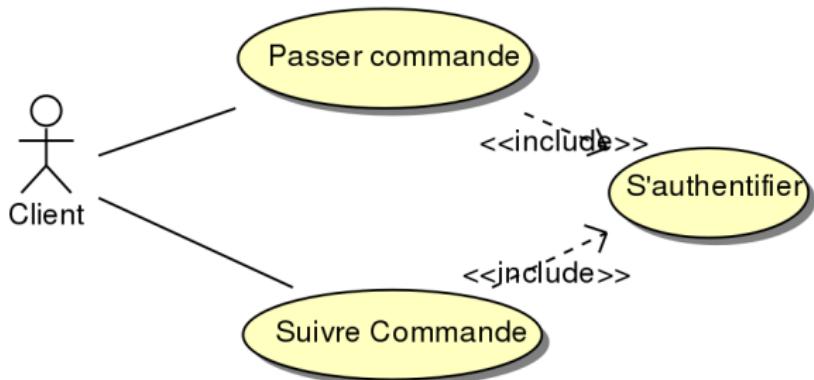


## Attention

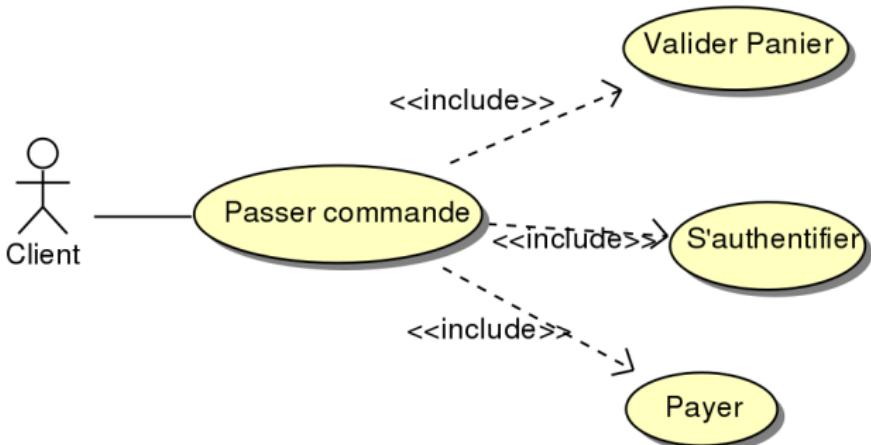
Le sens des flèches pointillées indique une dépendance, pas le sens de la relation d'inclusion.

- ▶ Les inclusions et les extensions sont toutes deux des **dépendances**.
  - ▶ Lorsqu'un cas B inclut un cas A, B dépend de A.
  - ▶ Lorsqu'un cas B étend un cas A, B dépend aussi de A.
  - ▶ On note toujours la dépendance par une flèche pointillée B → A qui se lit "B dépend de A".
- ▶ Lorsqu'un élément B dépend d'un élément A, toute modification de A sera susceptible d'avoir un impact sur B.
- ▶ Les "include" et les "extend" sont des **stéréotypes** (entre guillemets) des relations de dépendance.

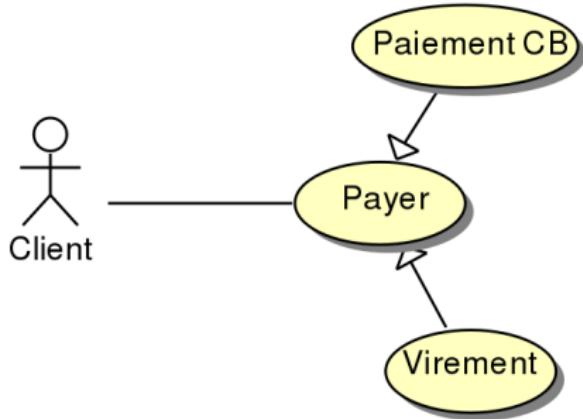
- ▶ Les relations entre cas permettent la **ré-utilisabilité** du cas “s’authentifier” : il sera inutile de développer plusieurs fois un module d’authentification.



- ▶ Quand un cas est trop complexe (faisant intervenir un trop grand nombre d'actions élémentaires), on peut procéder à sa **décomposition** en cas plus simples.



# Généralisation



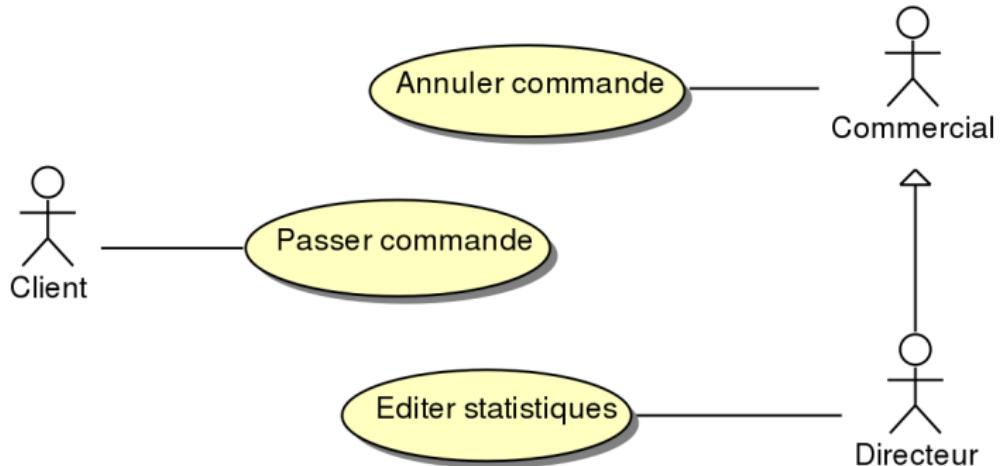
- ▶ Un virement est un cas particulier de paiement.

Un virement est une sorte de paiement.

- ▶ La flèche pointe vers l'élément général.
- ▶ Cette relation de généralisation/spécialisation est présente dans la plupart des diagrammes UML et se traduit par le concept d'**héritage** dans les langages orientés objet.

# Relations entre acteurs

- Une seule relation possible : la **généralisation**.



# Identification des acteurs

- ▶ Les principaux acteurs sont les utilisateurs du système.

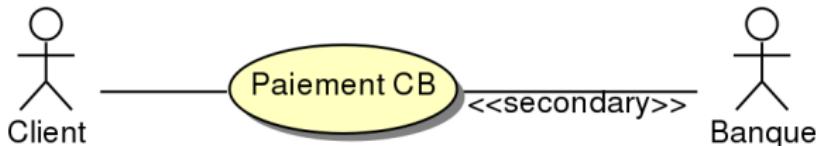
## Attention

Un acteur correspond à un **rôle**, pas à une personne physique.

- ▶ Une même personne physique peut être représentée par plusieurs acteurs si elle a plusieurs rôles.
- ▶ Si plusieurs personnes jouent le même rôle vis-à-vis du système, elles seront représentées par un seul acteur.
- ▶ En plus des utilisateurs, les acteurs peuvent être :
  - ▶ Des périphériques manipulés par le système (imprimantes...) ;
  - ▶ Des logiciels déjà disponibles à intégrer dans le projet ;
  - ▶ Des systèmes informatiques externes au système mais qui interagissent avec lui, etc.
- ▶ Pour faciliter la recherche des acteurs, on se fonde sur les **frontières** du système.

# Acteurs principaux et secondaires

- ▶ L'acteur est dit **principal** pour un cas d'utilisation lorsque l'acteur est à l'initiative des échanges nécessaires pour réaliser le cas d'utilisation.



- ▶ Les acteurs **secondaires** sont sollicités par le système alors que le plus souvent, les acteurs principaux ont l'initiative des interactions.
  - ▶ Le plus souvent, les acteurs secondaires sont d'autres systèmes informatiques avec lesquels le système développé est inter-connecté.

## Recenser les cas d'utilisation

- ▶ Il n'y a pas une manière mécanique et totalement objective de repérer les cas d'utilisation.
  - ▶ Il faut *se placer du point de vue de chaque acteur* et déterminer comment il se sert du système, dans quels cas il l'utilise, et à quelles fonctionnalités il doit avoir accès.
  - ▶ Il faut éviter *les redondances* et *limiter le nombre de cas* en se situant au bon niveau d'abstraction (par exemple, ne pas réduire un cas à une seule action).
  - ▶ Il ne faut pas faire apparaître les détails des cas d'utilisation, mais il faut rester au niveau des grandes fonctions du système.

Trouver le bon niveau de détail pour les cas d'utilisation est un problème difficile qui nécessite de l'expérience.



# Description des cas d'utilisation

- ▶ Le diagramme de cas d'utilisation décrit les grandes fonctions d'un système du point de vue des acteurs, mais n'expose pas de façon détaillée le dialogue entre les acteurs et les cas d'utilisation.
- ▶ **Un simple nom est tout à fait insuffisant pour décrire un cas d'utilisation.**

Chaque cas d'utilisation doit être documenté pour qu'il n'y ait aucune ambiguïté concernant son déroulement et ce qu'il recouvre précisément.

# Description textuelle d'un cas d'utilisation

## ► Identification :

- **Nom du cas** : Payer CB
- **Objectif** : Détails les étapes permettant à client de payer par carte bancaire
- **Acteurs** : Client, Banque (secondaire)
- **Date** : 18/09
- **Responsables** : Toto
- **Version** : 1.0



# Description textuelle d'un cas d'utilisation

- ▶ Commence lorsqu'un client demande le paiement par CB
- ▶ **Pré-conditions**
  - ▶ Le client a validé sa commande
- ▶ **Enchaînement nominal**
  1. Le client saisit les informations de sa carte bancaire
  2. Le système vérifie que le numéro de CB est correct
  3. Le système vérifie la carte auprès du système bancaire
  4. Le système demande au système bancaire de débiter le client
  5. Le système notifie le client du bon déroulement de la transaction
- ▶ **Enchaînements alternatifs**
  1. En (2) : si le numéro est incorrect, le client est averti de l'erreur, et invité à recommencer
  2. En (3) : si les informations sont erronées, elles sont re-demandées au client
- ▶ **Post-conditions**
  - ▶ La commande est validée
  - ▶ Le compte de l'entreprise est crédité



# Description textuelle d'un cas d'utilisation

## ► Rubriques optionnelles

### ► Contraintes non fonctionnelles :

- ▶ Fiabilité : les accès doivent être sécurisés
- ▶ Confidentialité : les informations concernant le client ne doivent pas être divulgués

### ► Contraintes liées à l'interface homme-machine :

- ▶ Toujours demander la validation des opérations bancaires

# Plan

## Conception objet élémentaire avec UML

Diagrammes de cas d'utilisation

Diagrammes de classes

Diagrammes d'objets

Diagrammes de séquences

## UML et méthodologie

## Modélisation avancée avec UML

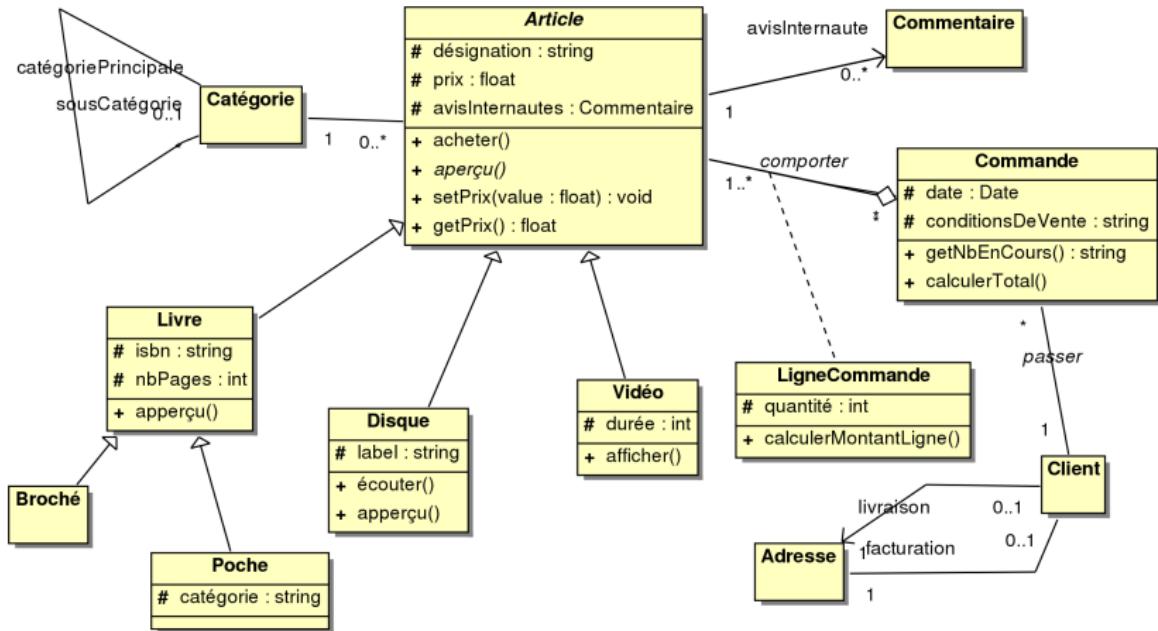
## Bonnes pratiques de la modélisation objet



- ▶ 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 la **STRUCTURE** et les liens entre les objets dont le système est composé.

Chaque nouveau diagramme montre un autre aspect du système, tous sont complémentaires et doivent être cohérents les uns avec les autres

# Exemple de diagramme de classes



# Concepts et instances

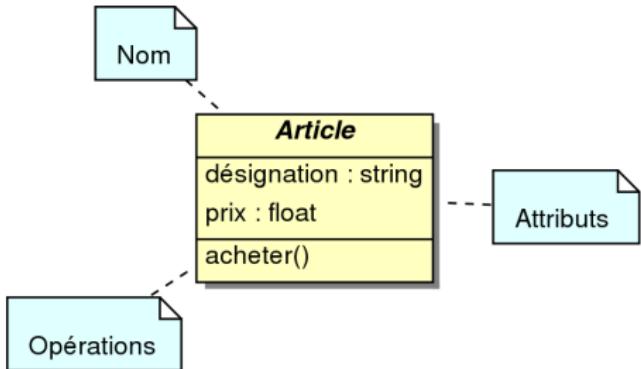
- ▶ Une **instance** est la concrétisation d'un **concept** abstrait.
  - ▶ Concept : Stylo
  - ▶ Instance : le stylo que vous utilisez à ce moment précis est une instance du concept stylo.
- ▶ Un **objet** est une instance d'une **classe**
  - ▶ Classe : Film
  - ▶ Objets : Il Padrino, 2001 Odyssée de l'Espace, etc.

Une classe décrit un *type* d'objets concrets.

- ▶ Une classe spécifie la manière dont tous les objets de même type seront décrits (désignation, label, auteur, etc).
- ▶ Un **lien** est une instance d'**association**.
  - ▶ Association : Concept "avis d'internaute" qui lie commentaire et article
  - ▶ Lien : instance [Jean avec son avis négatif], [Paul avec son avis positif]

# Classes et objets

- ▶ Une **classe** est la description d'un ensemble d'objets ayant une sémantique, des attributs, des méthodes et des relations en commun. Elle spécifie l'ensemble des caractéristiques qui composent des objets de même type.
- ▶ Une classe est composée d'un **nom**, d'**attributs** et d'**opérations**.
- ▶ Selon l'avancement de la modélisation, ces informations ne sont pas forcement toutes connues.
- ▶ D'autres compartiments peuvent être ajoutés : responsabilités, exceptions, etc.



# Propriétés : attributs et opérations

- ▶ Les attributs et les opérations sont les **propriétés** d'une classe.  
Leur nom commence par une minuscule.
- ▶ Un **attribut** décrit une donnée de la classe.
  - ▶ Les types des attributs et leurs initialisations ainsi que les modificateurs d'accès peuvent être précisés dans le modèle
  - ▶ Les attributs prennent des valeurs lorsque la classe est instanciée : ils sont en quelque sorte des "variables" attachées aux objets.
- ▶ Une **opération** est un service offert par la classe (un traitement que les objets correspondant peuvent effectuer).



# Compartiment des attributs

- ▶ Un attribut peut être initialisé et sa visibilité est définie lors de sa déclaration.
- ▶ **Syntaxe de la déclaration d'un attribut :**

```
modifAcces nomAtt:nomClasse [multi]=valeurI
```

<i>Article</i>
# désignation : string
# prix : float = -1
# avisInternautes : Commentaire [0..*]



# Compartiment des opérations

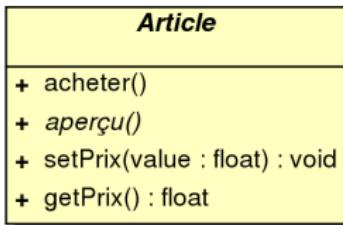
Une opération est définie par son nom ainsi que par les types de ses paramètres et le type de sa valeur de retour : **sa signature**

- ▶ La **syntaxe de la déclaration d'une opération** est la suivante :

```
modifAcces nomOpération(paramètres) : Classe
```

- ▶ La **syntaxe de la liste des paramètres** est la suivante :

```
nomClasse1 nomParam1 , . . . , nomClasseN nom
```



# Méthodes et Opérations

- Une **opération** est la signature d'une **méthode** indépendamment de son implantation.
- Exemples d'implémentations pour l'opération  
`fact(n:int):int`

```
{ // implementation iterative
    int resultat = 1 ;
    for (int i = n~; i>0~; i--)
        resultat*=i ;
    return resultat ;
}

{ // implementation recursive
    if (n==0)
        return 1 ;
    return (n * fact(n-1)) ;
}
```

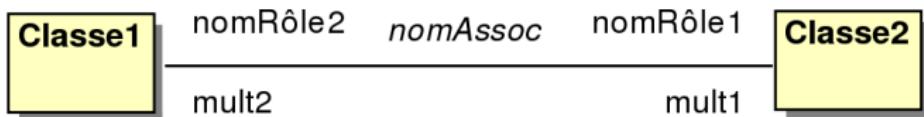


## Relations entre classes

- ▶ 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.

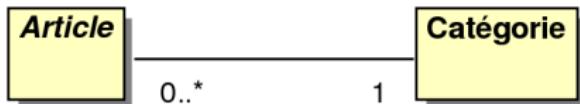
# Association

- ▶ Une **association** est une relation structurelle entre objets.
  - ▶ Une association est souvent utilisée pour représenter les liens possibles entre objets de classes données.
  - ▶ Elle est représentée par un trait entre classes.



# Multiplicités des associations

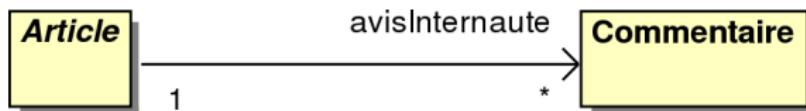
- ▶ La notion de **multiplicité** permet de contraindre le nombre d'objets intervenant dans les instanciations des associations.
- ▶ **Exemple :** un article n'appartient qu'à une seule catégorie (1) ; une catégorie concerne plus de 0 articles, sans maximum (0..\*).



- ▶ La syntaxe est MultMin..MultMax.
- ▶ "\*" à la place de MultMax signifie "plusieurs" sans préciser de nombre.
- ▶ "n..n" se note aussi "n", et "0..\*" se note "\*".

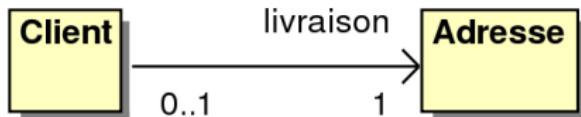
# Navigabilité d'une association

- ▶ 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.
- ▶ On peut aussi représenter les associations navigables dans un seul sens par des attributs.
  - ▶ Exemple : En ajoutant un attribut “avisInternaute” de classe “Commentaire” à *la place de l'association*.

# Association unidirectionnelle de 1 vers 1

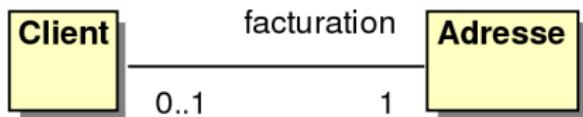


```
public class Adresse{...}

public class Client{
    private Adresse livraison;
    public void setAdresse(Adresse adresse){
        this.livraison = adresse;
    }
    public Adresse getAdresse(){
        return livraison;
    }
}
```

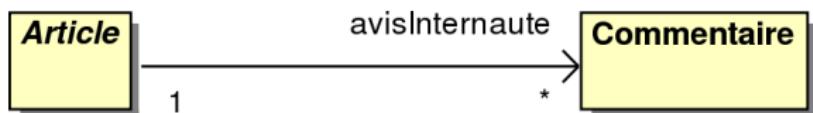


# Association bidirectionnelle de 1 vers 1



```
public class Client{  
    Adresse facturation;  
    public void setAdresse(Adresse uneAdresse){  
        if(uneAdresse!=null){  
            this.facturation = uneAdresse;  
            facturation.client = this; // correspondance  
        }  
    }  
}  
  
public class Adresse{  
    Client client;  
    public void setClient(Client unClient){  
        this.client = client;  
        client.facturation = this; // correspondance  
    }  
}
```

# Association unidirectionnelle de 1 vers \*

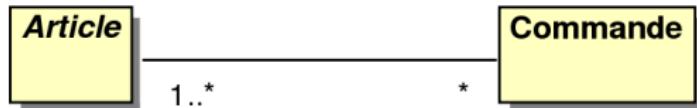


```
public class Commentaire{...}

public class Article{
    private Vector avisInternaute = new Vector();
    public void addCommentaire(Commentaire c){
        avisInternaute.addElement(c);
    }
    public void removeCommentaire(Commentaire c){
        avisInternaute.removeElement(c);
    }
}
```



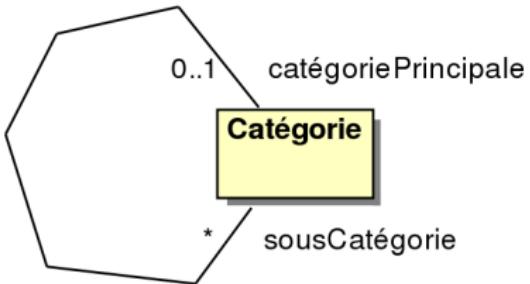
# Implantation d'une association bidirectionnelle de \* vers \*



Plus difficile : gérer à la fois la cohérence et les collections

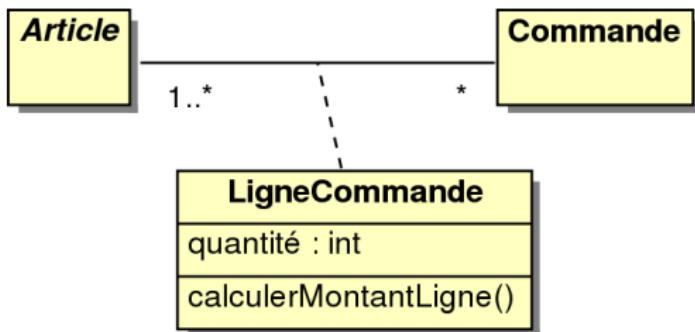
# Associations réflexives

- ▶ 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”.



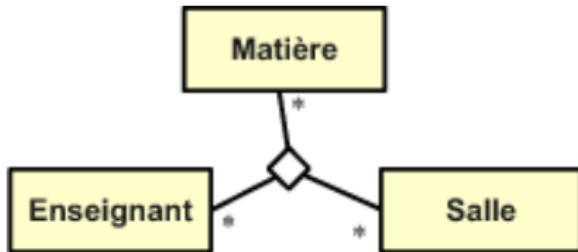
# Classe-association

- ▶ Une association peut être raffinée et avoir ses propres attributs, qui ne sont disponibles dans aucune des classes qu'elle lie.
- ▶ Comme, dans le modèle objet, seules les classes peuvent avoir des attributs, cette association devient alors une classe appelée "classe-association".



## Associations n-aires

- ▶ Une **association n-aire** lie plus de deux classes.
  - ▶ Notation avec un losange central pouvant éventuellement accueillir une classe-association.
  - ▶ La multiplicité de chaque classe s'applique à une instance du losange.

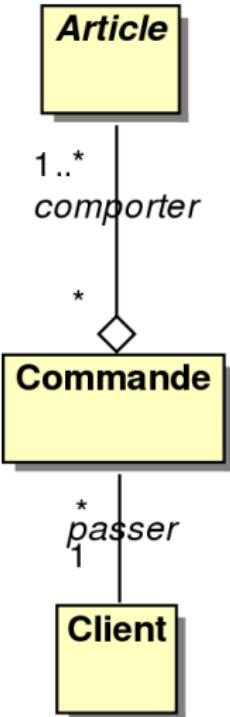


Les associations n-aires sont peu fréquentes et concernent surtout les cas où les multiplicités sont toutes “\*”. Dans la plupart des cas, on utilisera plus avantageusement des classes-association ou plusieurs relations binaires.

# Association de type agrégation

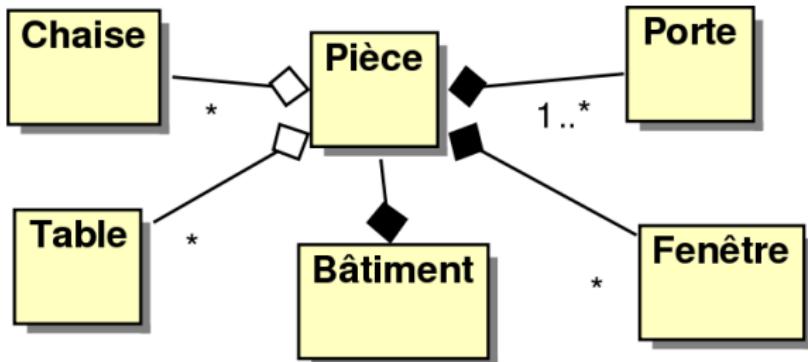
- ▶ Une **agrégation** est une forme particulière d'association. Elle représente la relation d'**inclusion** d'un élément dans un ensemble.
- ▶ On représente l'agrégation par l'ajout d'un losange vide du côté de l'agrégat.

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



# Association de type composition

- ▶ La relation de **composition** décrit une **contenance** structurelle entre instances. On utilise un losange plein.
- ▶ La **destruction** et la **copie** de l'objet composite (l'ensemble) impliquent respectivement la destruction ou la copie de ses composants (les parties).
- ▶ Une instance de la partie n'appartient jamais à plus d'une instance de l'élément composite.



# Composition et agrégation

- ▶ Dès lors que l'on a une relation du tout à sa partie, on a une relation d'agrégation ou de composition.

La composition est aussi dite “agrégation forte”.

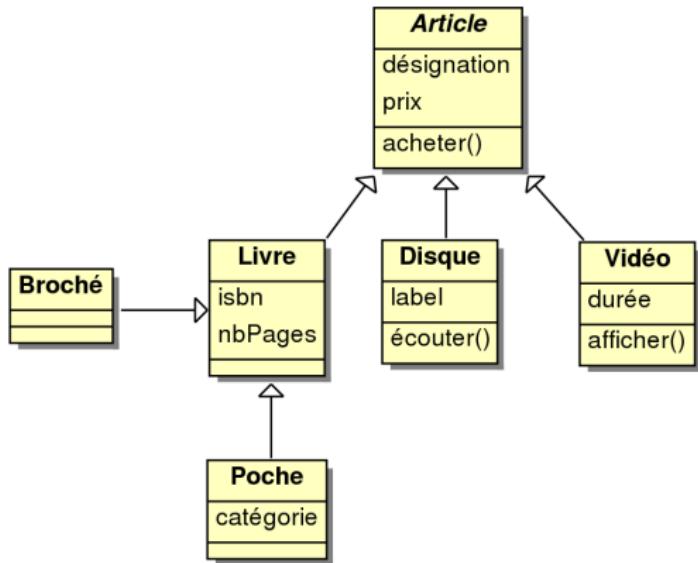
- ▶ Pour décider de mettre une composition plutôt qu'une agrégation, on doit se poser les questions suivantes :
  - ▶ Est-ce que la destruction de l'objet composite (du tout) implique nécessairement la destruction des objets composants (les parties) ? C'est le cas si les composants n'ont pas d'autonomie vis-à-vis des composites.
  - ▶ Lorsque l'on copie le composite, doit-on aussi copier les composants, ou est-ce qu'on peut les “réutiliser”, auquel cas un composant peut faire partie de plusieurs composites ?

Si on répond par l'affirmative à ces deux questions, on doit utiliser une composition.



# Relation d'héritage

- ▶ L'héritage une relation de spécialisation/généralisation.
- ▶ Les éléments spécialisés héritent de la structure et du comportement des éléments plus généraux (attributs, opérations, associations et nouveaux 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



# Implantation de l'héritage en Java

```
class Article {  
    ...  
    void acheter() {  
        ...  
    }  
}  
class Livre extends Article {  
    ...  
}
```

## Attention

Les “extends” Java n'a rien à voir avec le “extend” UML vu pour les cas d'utilisation



- ▶ L'**encapsulation** est un principe de conception consistant à protéger le cœur d'un système des accès intempestifs venant de l'extérieur.
- ▶ En UML, utilisation de **modificateurs d'accès** sur les attributs ou les classes :
  - ▶ **Public** ou “+” : propriété ou classe visible partout
  - ▶ **Protected** ou “#”. propriété ou classe visible dans la classe et par tous ses descendants.
  - ▶ **Private** ou “-” : propriété ou classe visible uniquement dans la classe
  - ▶ **Package**, ou “~” : propriété ou classe visible uniquement dans le paquetage
- ▶ Il n'y a pas de visibilité “par défaut”.

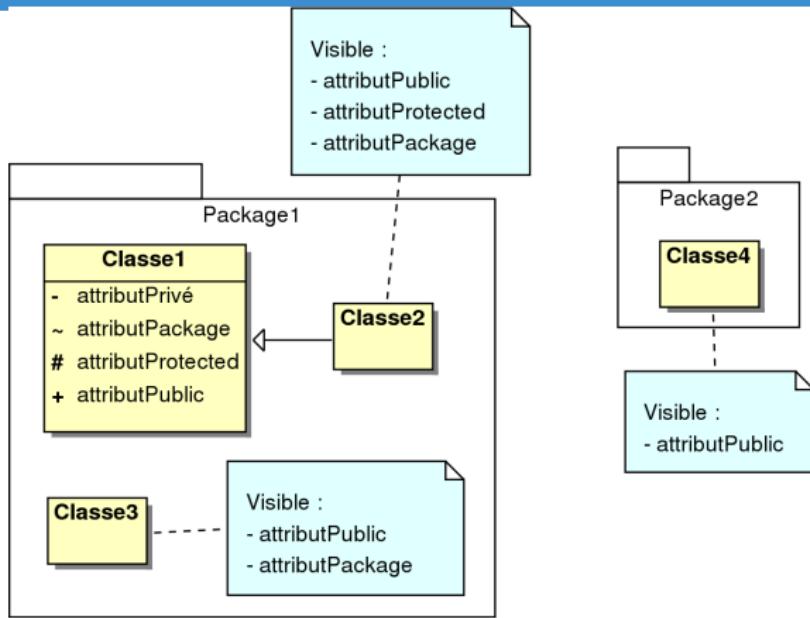
# Encapsulation

## Package (paquetage)

Les packages contiennent des éléments de modèle de haut niveau, comme des classes, des diagrammes de cas d'utilisation ou d'autres packages. On organise les éléments modélisés en packages et sous-packages.



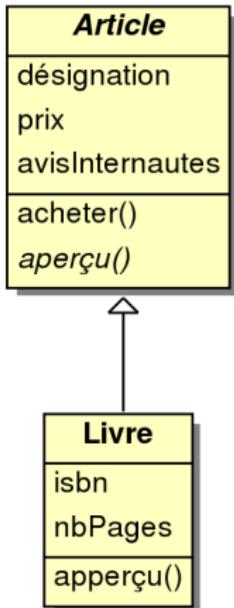
# Exemple d'encapsulation



Les modificateurs d'accès sont également applicables aux opérations.

# Relation d'héritage et propriétés

- ▶ La classe enfant 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)
- ▶ Toutefois, elle n'a pas accès aux propriétés privées.



# Terminologie de l'héritage

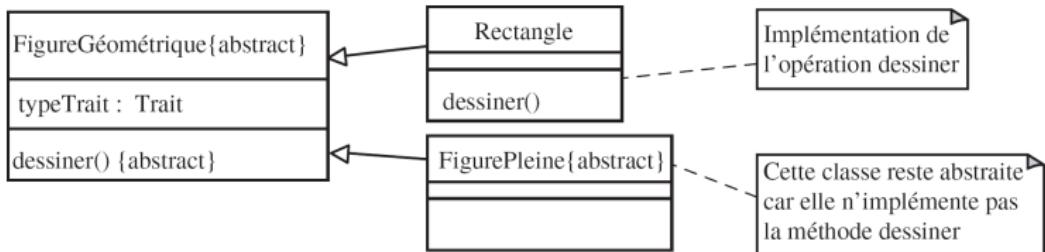
- ▶ Une classe enfant peut **redéfinir** (même signature) une ou plusieurs méthodes de la classe parent.
  - ▶ Sauf indications contraires, un objet utilise les opérations les plus spécialisées dans la hiérarchie des classes.
  - ▶ La **surcharge d'opérations** (même nom, mais signatures des opérations différentes) est possible dans toutes les classes.
- ▶ Toutes les associations de la classe parent s'appliquent, par défaut, aux classes **dérivées** (classes enfant).
- ▶ **Principe de 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).

# Classes abstraites

- Une opération est dite **abstraite** lorsqu'on connaît sa signature mais pas la manière dont elle peut être réalisée.

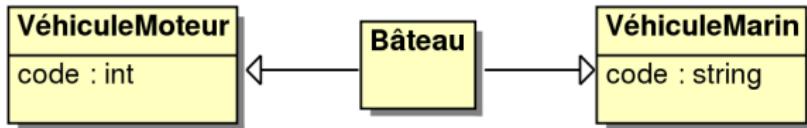
Il appartient aux classes enfant de définir les méthodes abstraites.

- 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.



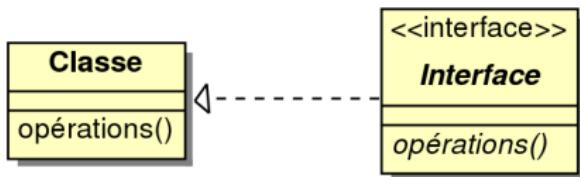
# Héritage multiple

- ▶ 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 ruser en employant des interfaces.



# Interface

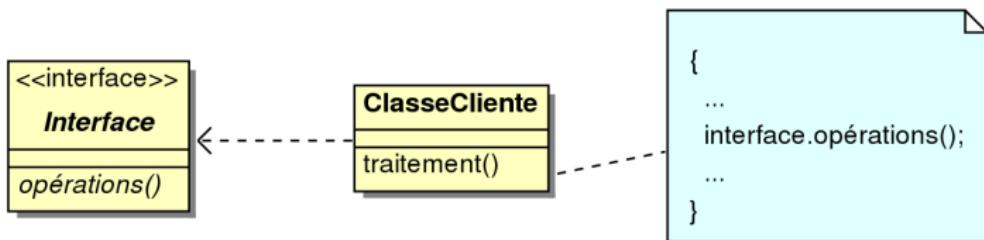
- ▶ Le rôle d'une **interface** est de regrouper un ensemble d'opérations assurant un service cohérent offert par une classe.
- ▶ Une interface est définie comme une classe, avec les mêmes compartiments. On ajoute le stéréotype "interface" avant le nom de l'interface.
- ▶ On utilise une relation de type **réalisation** entre une interface et une classe qui l'implémente.



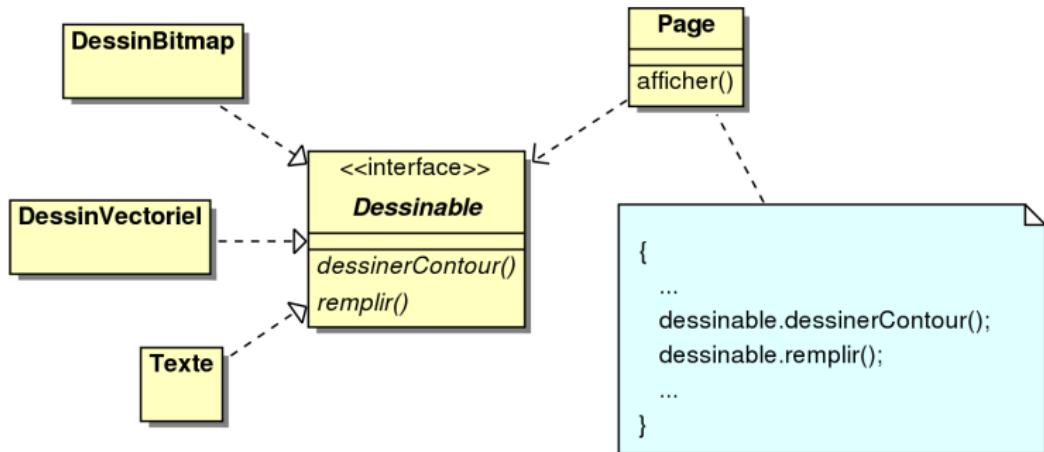
- ▶ Les classes implémentant une interface doivent implémenter toutes les opérations décrites dans l'interface

# Classe cliente d'une interface

- ▶ Quand une classe dépend d'une interface (**interface requise**) 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.



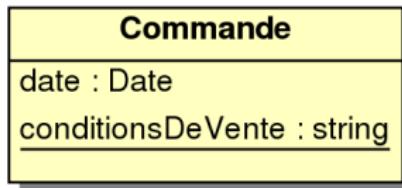
# Exemple d'interface



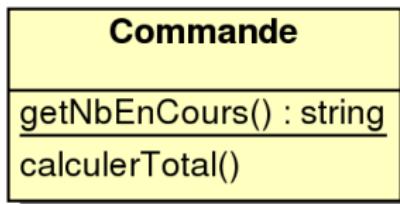
- ▶ Les **attributs dérivés** peuvent être calculés à partir d'autres attributs et des formules de calcul.
  - ▶ Les attributs dérivés sont symbolisés par l'ajout d'un “ / ” devant leur nom.
  - ▶ Lors de la conception, un attribut dérivé peut être utilisé comme marqueur jusqu'à ce que vous puissiez déterminer les règles à lui appliquer.
- ▶ Une **association dérivée** est conditionnée ou peut être déduite à partir d'autres autres associations. On utilise également le symbole “ / ”.

# Attributs de classe

- ▶ Par défaut, les valeurs des attributs définis dans une classe diffèrent d'un objet à un autre. Parfois, il est nécessaire de définir un **attribut de classe** qui garde une valeur unique et partagée par toutes les instances.
- ▶ Graphiquement, un attribut de classe est souligné

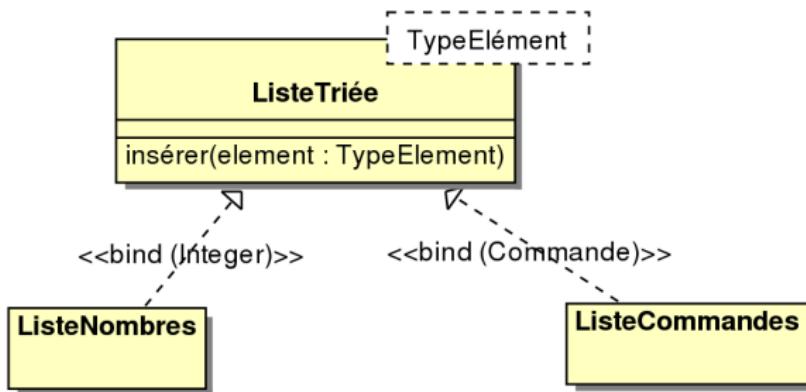


- ▶ Semblable aux attributs de classe
  - ▶ Une **opération de classe** est une propriété de la classe, et non de ses instances.
  - ▶ Elle n'a pas accès aux attributs des objets de la classe.



# 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.



- ▶ Java5 : généricité
- ▶ C++ : templates

- ▶ On peut utiliser les diagrammes de classes pour représenter un système à différents niveaux d'abstraction :
  - ▶ Le point de vue **spécification** met l'accent sur les interfaces des classes plutôt que sur leurs contenus.
  - ▶ Le point de vue **conceptuel** capture les concepts du domaine et les liens qui les lient. Il s'intéresse peu ou prou à la manière éventuelle d'implémenter ces concepts et relations et aux langages d'implantation.
  - ▶ Le point de vue **implantation**, le plus courant, détaille le contenu et l'implantation de chaque classe.
- ▶ Les diagrammes de classes s'étoffent à mesure qu'on va de hauts niveaux à de bas niveaux d'abstraction (de la spécification vers l'implantation)

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.

## Conception objet élémentaire avec UML

Diagrammes de cas d'utilisation

Diagrammes de classes

Diagrammes d'objets

Diagrammes de séquences

## UML et méthodologie

## Modélisation avancée avec UML

## Bonnes pratiques de la modélisation objet



- ▶ **Le diagramme d'objets** représente les objets d'un système à un instant donné. Il permet de :
  - ▶ Illustrer le modèle de classes (en montrant un exemple qui explique le modèle) ;
  - ▶ Préciser certains aspects du système (en mettant en évidence des détails imperceptibles dans le diagramme de classes) ;
  - ▶ Exprimer une exception (en modélisant des cas particuliers, des connaissances non généralisables...).

Le diagramme de classes modélise des *règles* et  
le diagramme d'objets modélise des *faits*.

# Représentation des objets

- ▶ Comme les classes, on utilise des **cadres compartimentés**.
- ▶ En revanche, **les noms des objets sont soulignés** et on peut rajouter son identifiant devant le nom de sa classe.
- ▶ Les valeurs (a) ou l'état (f) d'un objet peuvent être spécifiées.
- ▶ Les instances peuvent être **anonymes** (a,c,d), **nommées** (b,f), **orphelines** (e), **multiples** (d) ou **stéréotypées** (g).

:Personne  
nom:String="toto"  
age :integer="inconnu"

(a)

albert:Adhérent

(b)

:Adhérent ::Asso

(c)

:Adhérent

(d)

albert:

(e)

albert:Adhérent  
[grand]

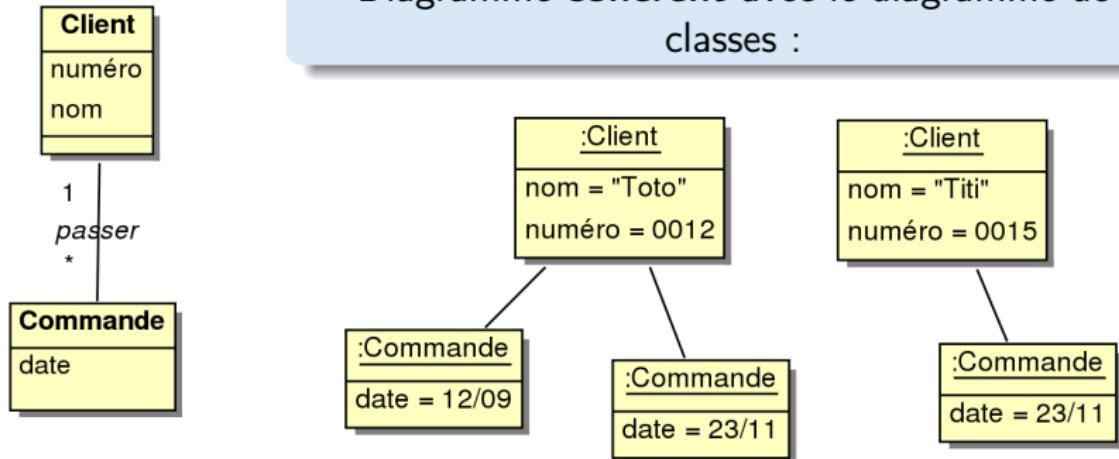
(f)

« exception »  
e.IOException

(g)

- Le diagramme de classes **contraint** la structure et les liens entre les objets.

Diagramme **cohérent** avec le diagramme de classes :



# Diagramme de classes et diagramme d'objets

- Le diagramme de classes **contraint** la structure et les liens entre les objets.

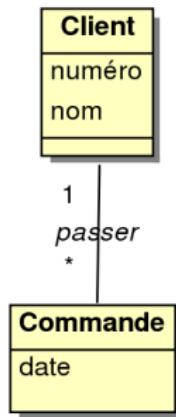
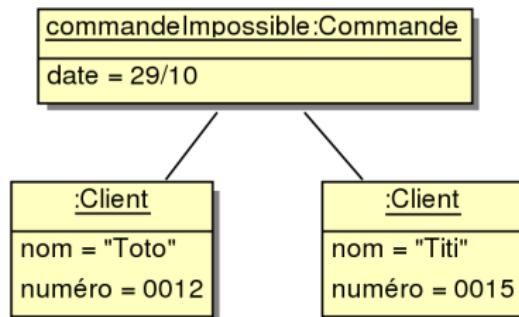


Diagramme incohérent avec le diagramme de classes :



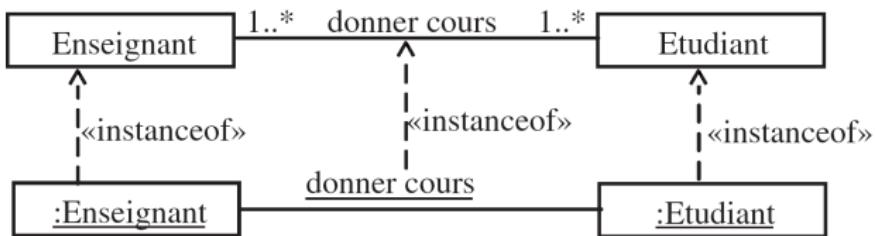
- ▶ Un **lien** est une instance d'une association.
- ▶ Un lien se représente comme une association mais s'il a un nom, il est souligné.

### Attention

Naturellement, on ne représente pas les multiplicités qui n'ont aucun sens au niveau des objets.

# Relation de dépendance d'instanciation

- ▶ La relation de **dépendance d'instanciation** (stéréotypée) décrit la relation entre un classeur et ses instances.
- ▶ Elle relie, en particulier, les associations aux liens et les classes aux objets.



## Conception objet élémentaire avec UML

Diagrammes de cas d'utilisation

Diagrammes de classes

Diagrammes d'objets

Diagrammes de séquences

## UML et méthodologie

## Modélisation avancée avec UML

## Bonnes pratiques de la modélisation objet



# Objectif des diagrammes de séquence

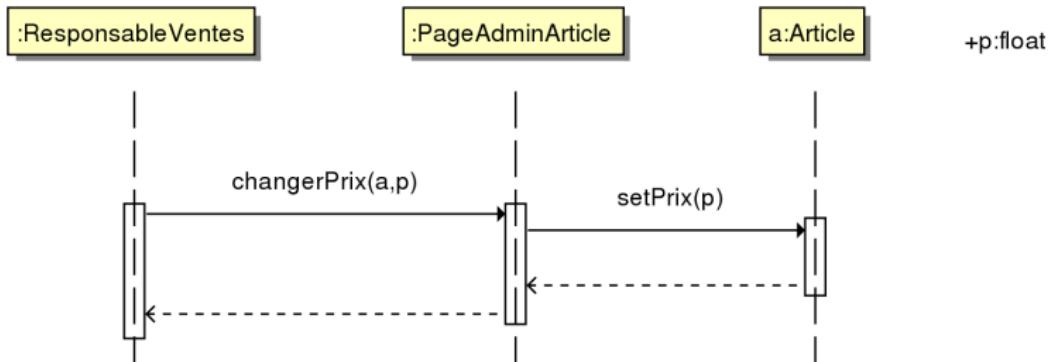
- ▶ Les **diagrammes de cas d'utilisation** modélisent à **QUOI** sert le système, en organisant les interactions possibles avec les acteurs.
- ▶ Les **diagrammes de classes** permettent de spécifier la structure et les liens entre les objets dont le système est composé : ils spécifie **QUI** sera à l'oeuvre dans le système pour réaliser les fonctionnalités décrites par les diagrammes de cas d'utilisation.
- ▶ Les **diagrammes de séquences** permettent de décrire **COMMENT** les éléments du système interagissent entre eux et avec les acteurs.
  - ▶ Les objets au coeur d'un système interagissent en s'échangent des messages.
  - ▶ Les acteurs interagissent avec le système au moyen d'IHM (Interfaces Homme-Machine).

# Exemple d'interaction

- ▶ Cas d'utilisation :

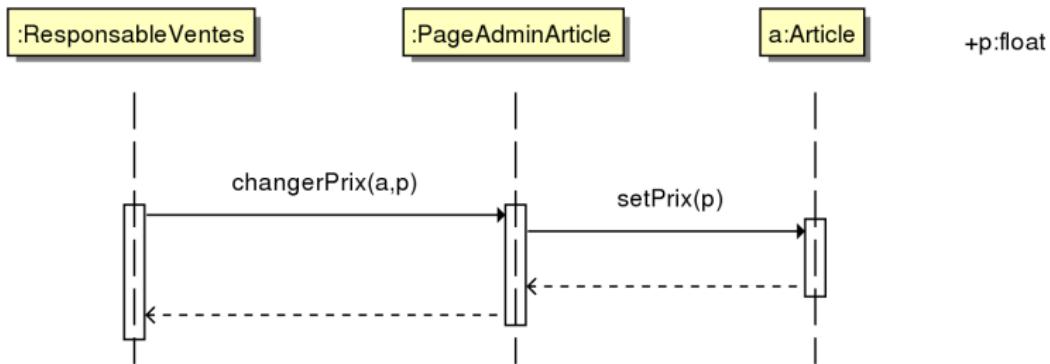


- ▶ Diagramme de séquences correspondant :

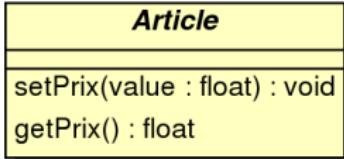


# Exemple d'interaction

- ▶ Diagramme de séquences correspondant :



- ▶ Opérations nécessaires dans le diagramme de classes :

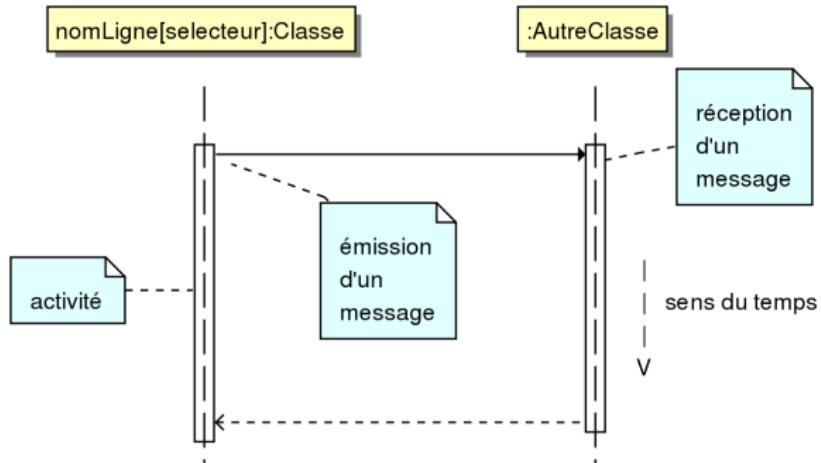


# Ligne de vie

- Une ligne de vie représente un participant à une interaction (objet ou acteur).

`nomLigneDeVie [selecteur] : nomClasseOuActeur`

- Dans le cas d'une collection de participants, un sélecteur permet de choisir un objet parmi n (par exemple `objets[2]`).



# Messages

- ▶ Les principales informations contenues dans un diagramme de séquence sont les **messages** échangés entre les lignes de vie, présentés dans un ordre chronologique.
  - ▶ Un message définit une communication particulière entre des lignes de vie (objets ou acteurs).
  - ▶ Plusieurs types de messages existent, dont les plus courants :
    - ▶ l'envoi d'un signal ;
    - ▶ l'invocation d'une opération (appel de méthode) ;
    - ▶ la création ou la destruction d'un objet.
- ▶ La réception des messages provoque une **période d'activité** (rectangle vertical sur la ligne de vie) marquant le traitement du message (spécification d'exécution dans le cas d'un appel de méthode).

# Principaux types de messages

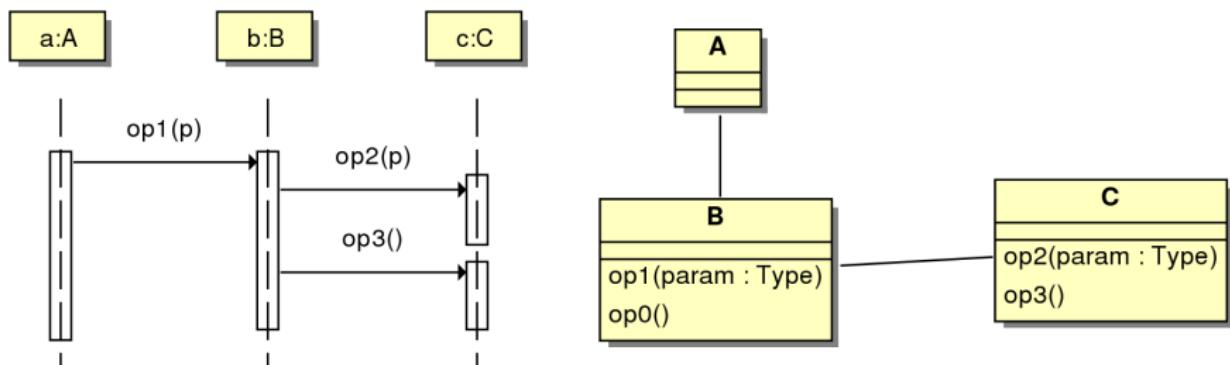
- ▶ Un message **synchrone** bloque l'expéditeur jusqu'à la réponse du destinataire. Le flux de contrôle passe de l'émetteur au récepteur.
  - ▶ Typiquement : appel de méthode
    - ▶ Si un objet A invoque une méthode d'un objet B, A reste bloqué tant que B n'a pas terminé.
- ▶ On peut associer aux messages d'appel de méthode un message de retour (en pointillés) marquant la reprise du contrôle par l'objet émetteur du message synchrone.
- ▶ Un message **asynchrone** n'est pas bloquant pour l'expéditeur. Le message envoyé peut être pris en compte par le récepteur à tout moment ou ignoré.
  - ▶ Typiquement : envoi de signal (voir stéréotype de classe "signal").



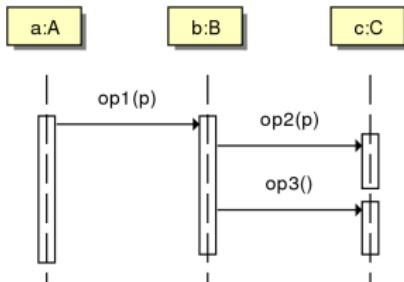
# Correspondance messages / opérations

- ▶ Les **messages synchrones** correspondent à des **opérations** dans le diagramme de classes.

Envoyer un message et attendre la réponse pour poursuivre son activité revient à invoquer une méthode et attendre le retour pour poursuivre ses traitements.



# implantation des messages synchrones



```

class B {
    C c;
    op1(p:Type){
        c.op2(p);
        c.op3();
    }
}
  
```

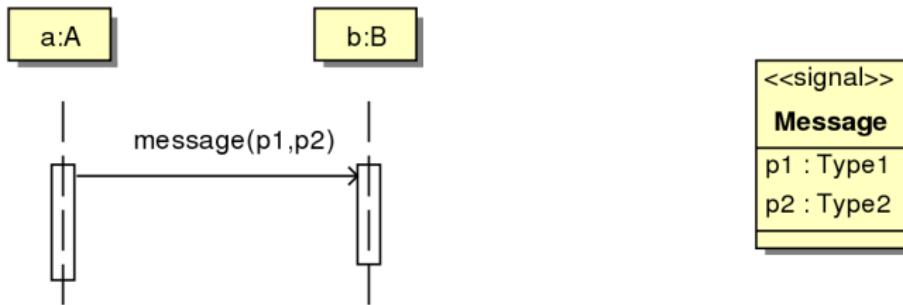
```

class C {
    op2(p:Type){
        ...
    }
    op3(){
        ...
    }
}
  
```

# Correspondance messages / signaux

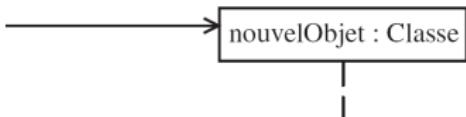
- ▶ Les **messages asynchrones** correspondent à des **signaux** dans le diagramme de classes.

Les signaux sont des objets dont la classe est stéréotypée “signal” et dont les attributs (porteurs d'information) correspondent aux paramètres du message.

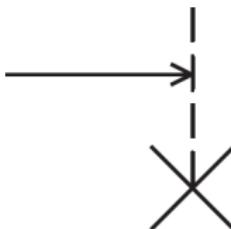


# Création et destruction de lignes de vie

- ▶ La **création** d'un objet est matérialisée par une flèche qui pointe sur le sommet d'une ligne de vie.
- ▶ On peut aussi utiliser un message asynchrone ordinaire portant le nom "create".



- ▶ La **destruction** d'un objet est matérialisée par une croix qui marque la fin de la ligne de vie de l'objet.



# Messages complets, perdus et trouvés

- ▶ **Un message complet** est tel que les événements d'envoi et de réception sont connus.
  - ▶ Un message complet est représenté par une flèche partant d'une ligne de vie et arrivant à une autre ligne de vie.
- ▶ **Un message perdu** est tel que l'événement d'envoi est connu, mais pas l'événement de réception.



- ▶ La flèche part d'une ligne de vie mais arrive sur un cercle indépendant marquant la méconnaissance du destinataire.
- ▶ Exemple : broadcast.
- ▶ **Un message trouvé** est tel que l'événement de réception est connu, mais pas l'événement d'émission.



# Syntaxe des messages

- ▶ La **syntaxe des messages** est :

```
nomSignalOuOperation(parametres)
```

- ▶ La **syntaxe des arguments** est la suivante :

```
nomParametre=valeurParametre
```

- ▶ Pour un argument modifiable :

```
nomParametre:valeurParametre
```

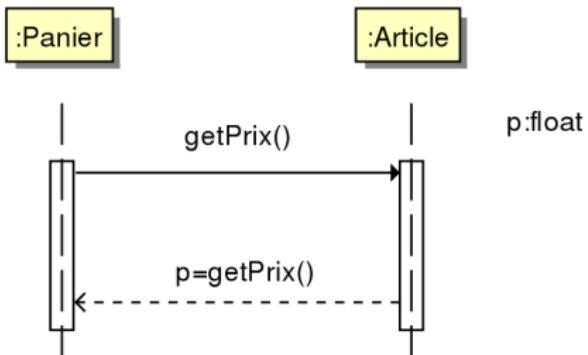
- ▶ Exemples :

- ▶ appeler("Capitaine Haddock", 54214110)
- ▶ afficher(x,y)
- ▶ initialiser(x=100)
- ▶ f(x:12)



# Messages de retour

- ▶ Le récepteur d'un message **synchrone** rend la main à l'émetteur du message en lui envoyant un **message de retour**
- ▶ Les messages de retour sont optionnels : la fin de la période d'activité marque également la fin de l'exécution d'une méthode.
- ▶ Ils sont utilisés pour spécifier le résultat de la méthode invoquée.



Le retour des messages asynchrones s'effectue par l'envoi de nouveaux messages asynchrones.

- ▶ La **syntaxe des messages de retour** est :

```
attributCible=nomOperation(params):valeurR
```

- ▶ La **syntaxe des paramètres** est :

```
nomParam=valeurParam
```

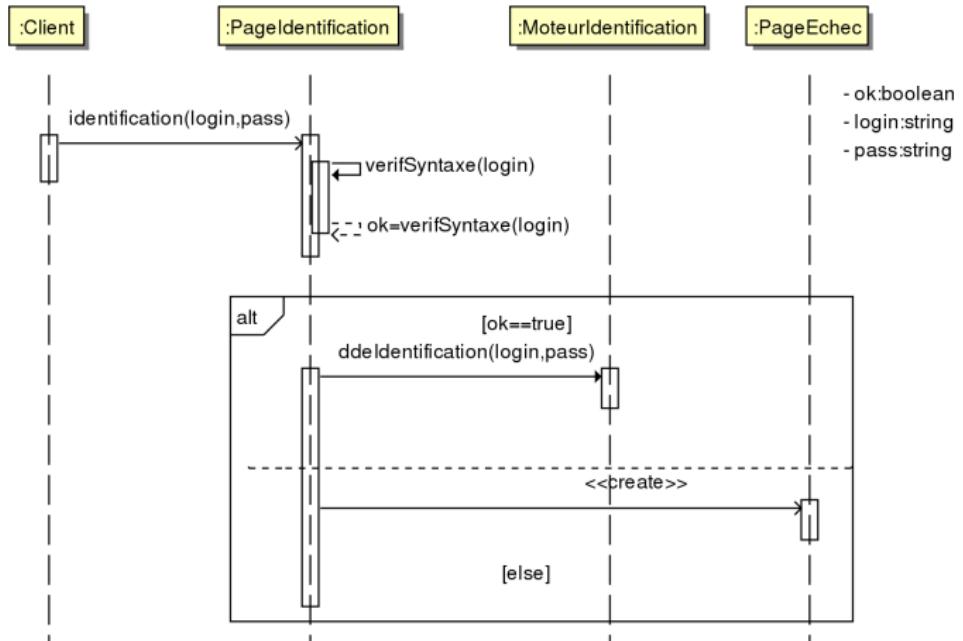
ou

```
nomParam:valeurParam
```

- ▶ Les messages de retour sont représentés en pointillés.

- ▶ Un **fragment combiné** permet de décomposer une interaction complexe en fragments suffisamment simples pour être compris.
  - ▶ Recombiner les fragments restitue la complexité.
  - ▶ Syntaxe complète avec UML 2 : représentation complète de processus avec un langage simple (ex : processus parallèles).
- ▶ Un fragment combiné se représente de la même façon qu'une interaction. Il est représenté un rectangle dont le coin supérieur gauche contient un pentagone.
  - ▶ Dans le pentagone figure le type de la combinaison (appelé "**opérateur d'interaction**").

# Exemple de fragment avec l'opérateur conditionnel



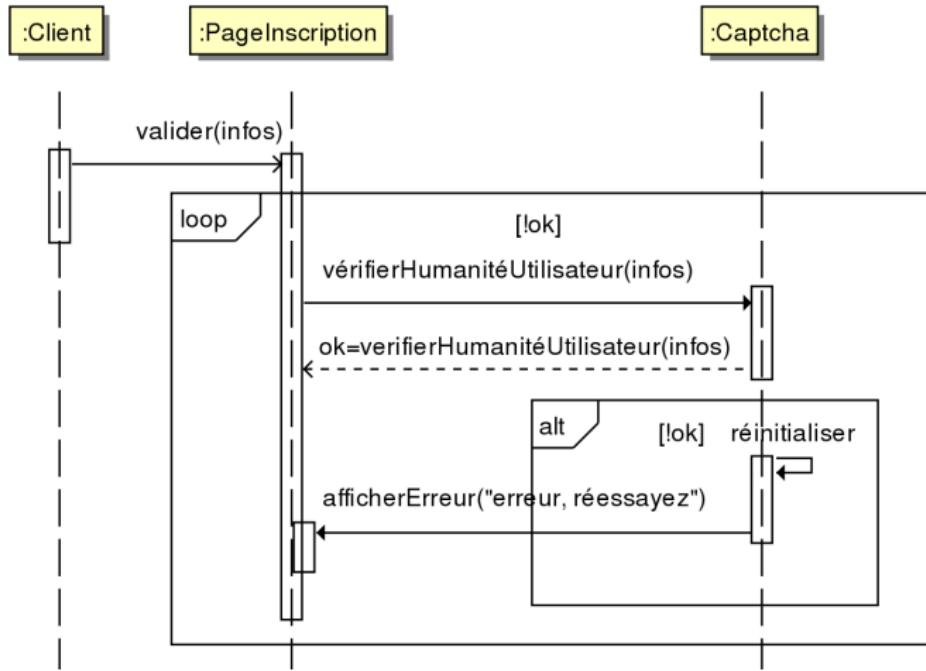
# Type d'opérateurs d'interaction

- ▶ Opérateurs de branchement (**choix et boucles**) :
  - ▶ alternative, option, break et loop ;
- ▶ Opérateurs contrôlant l'envoi en **parallèle** de messages :
  - ▶ parallel et critical region ;
- ▶ Opérateurs contrôlant l'**envoi de messages** :
  - ▶ ignore, consider, assertion et negative ;
- ▶ Opérateurs fixant l'**ordre** d'envoi des messages :
  - ▶ weak sequencing et strict sequencing.



# Opérateur de boucle

97



► Syntaxe d'une boucle :

```
loop(minNbIterations, maxNbIterations)
```

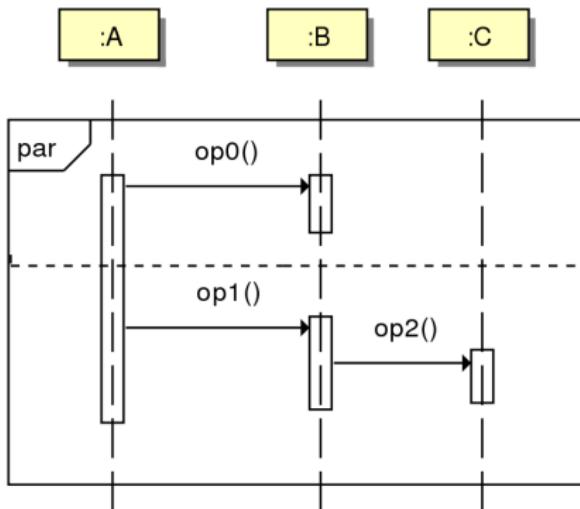
- ▶ La boucle est répétée au moins minNbIterations fois avant qu'une éventuelle condition booléenne ne soit testée (la condition est placée entre crochets dans le fragment)
- ▶ Tant que la condition est vraie, la boucle continue, au plus maxNbIterations fois.

► Notations :

- ▶ loop(valeur) est équivalent à loop(valeur, valeur).
- ▶ loop est équivalent à loop(0, \*), où \* signifie "illimité".

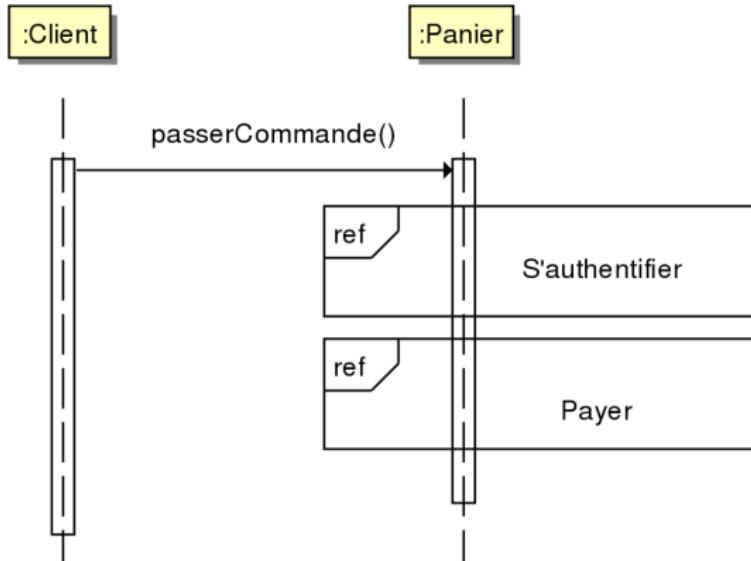
# Opérateur parallèle

- ▶ L'opérateur **par** permet d'envoyer des messages en parallèle.
- ▶ Ce qui se passe de part et d'autre de la ligne pointillée est indépendant.



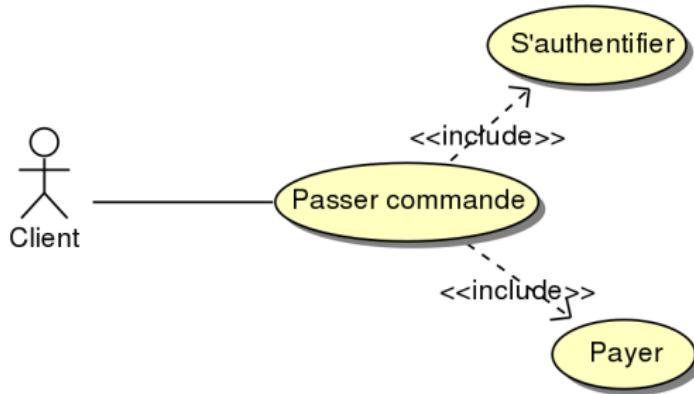
# Réutilisation d'une interaction

- ▶ Réutiliser une interaction consiste à placer un fragment portant la référence “ref” là où l'interaction est utile.



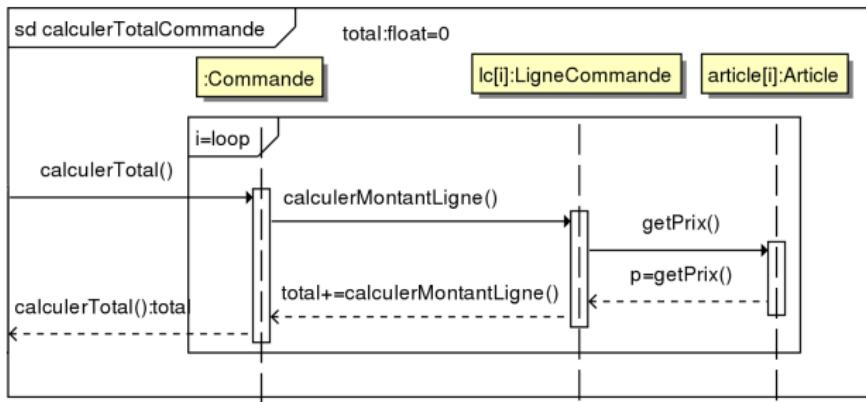
- ▶ On spécifie le nom de l'interaction dans le fragment.

- ▶ Chaque cas d'utilisation donne lieu à un diagramme de séquences



- ▶ Les inclusions et les extensions sont des cas typiques d'utilisation de la réutilisation par référencement

- ▶ Une interaction peut être identifiée par un fragment sd (pour pour "sequence diagram") précisant son nom
- ▶ Un message peut partir du bord de l'interaction, spécifiant le comportement du système après réception du message, quel que soit l'expéditeur



## Conception objet élémentaire avec UML

### UML et méthodologie

Des besoins au code avec UML : une méthode minimale

Rational Unified Process

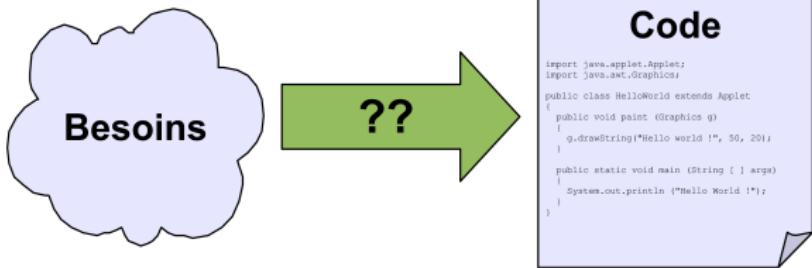
eXtreme programming

## Modélisation avancée avec UML

## Bonnes pratiques de la modélisation objet



# Pourquoi une méthode ?



Ensemble d'étapes partiellement ordonnées, qui concourent à l'obtention d'un système logiciel ou à l'évolution d'un système existant.

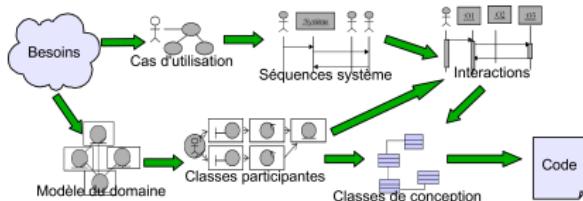
- ▶ **Objectif :** produire des logiciels
  - ▶ De qualité (qui répondent aux besoins de leurs utilisateurs)
  - ▶ Dans des temps et des coûts prévisibles
- ▶ A chaque étape, on produit :
  - ▶ Des modèles ;
  - ▶ De la documentation ;
  - ▶ Du code.

- ▶ La méthode MERISE fournit :
  - ▶ Un langage de modélisation graphique (MCD, MPD, MOT, MCT...)
  - ▶ ET Une démarche à adopter pour développer un logiciel.
- ▶ UML n'est qu'un langage :
  - ▶ Il spécifie comment décrire des cas d'utilisation, des classes, des interactions
  - ▶ Mais ne préjuge pas de la démarche employée.
- ▶ Méthodes s'appuyant sur UML :
  - ▶ RUP (Rational Unified Process) - par les auteurs d'UML ;
  - ▶ XP (eXtreme Programming) - pouvant s'appuyer sur UML.

## Objectif

Résoudre 80% des problèmes avec 20% d'UML.

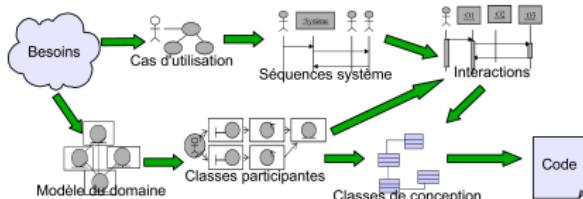
- ▶ Proposition d'une méthode **archi-minimale** :
  - ▶ Très nettement moins complexe que RUP ;
  - ▶ Adaptée pour à projets modestes ;
  - ▶ Minimum vital pour qui prétend utiliser *un peu* UML.



## Objectif

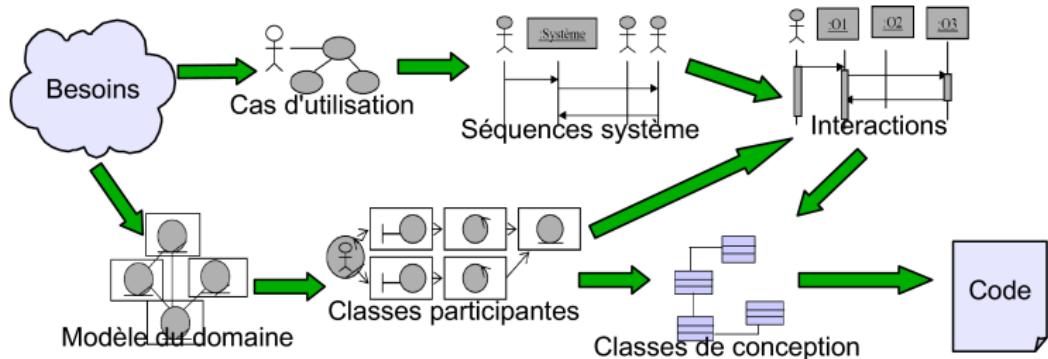
Résoudre 80% des problèmes avec 20% d'UML.

- ▶ Inspirée de
  - ▶ "UML 2 - Modéliser une application web"
  - ▶ **Pascal Roques**
  - ▶ Editions Eyrolles (2006)



## Objectif

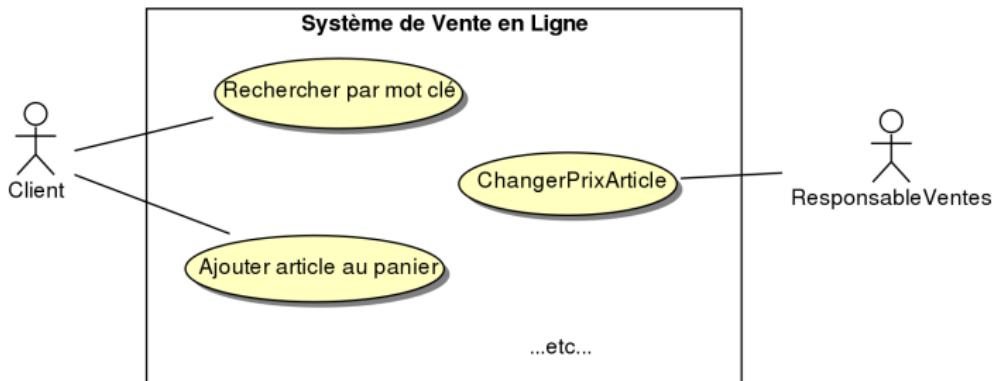
Résoudre 80% des problèmes avec 20% d'UML.



# Cas d'utilisation

## ► Comment définir les besoins ?

1. Identifier les limites du système ;
2. Identifier les acteurs ;
3. Identifier les cas d'utilisation ;
4. Structurer les cas d'utilisation en packages ;
5. Ajouter les relations entre cas d'utilisation ;
6. Classer les cas d'utilisation par ordre d'importance.



## Exemple de classement

Cas d'utilisation	Priorité	Risque
Ajouter article au panier	Haute	Moyen
Changer prix Article	Moyen	Moyen
Rechercher par mots-clés	Bas	Moyen
... etc ...	... etc ...	... etc ...

- ▶ Un tel classement permet de déterminer les cas d'utilisation centraux en fonction :
  - ▶ De leur priorité fonctionnelle ;
  - ▶ Du risque qu'il font courrir au projet dans son ensemble.
- ▶ Les fonctionnalités des cas les plus centraux seront développées prioritairement.

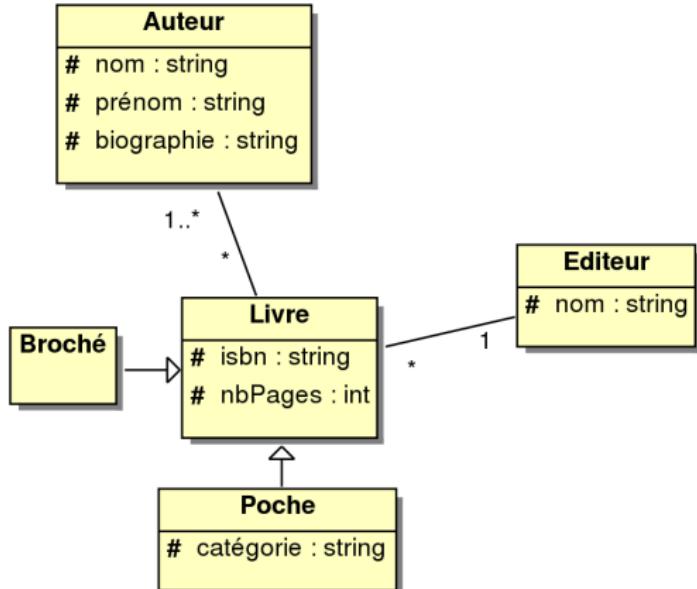
# Modèle du domaine

Le modèle du domaine est constitué d'un ensemble de classes dans lesquelles aucune opération n'est définie.

- ▶ Le **modèle du domaine** décrit les **concept invariants** du domaine d'application.
  - ▶ **Exemple :** Pour un logiciel de gestions de factures, on aura des classes comme Produit, Client, Facture...
    - ▶ Peu importe que le logiciel soit en ligne ou non.
    - ▶ Peu importent les technologies employées.
- ▶ **Etapes de la démarche :**
  1. Identifier les concepts du domaine ;
  2. Ajouter les associations et les attributs ;
  3. Généraliser les concepts ;
  4. Structurer en packages : structuration selon les principes de cohérence et d'indépendance.
- ▶ Les concepts du domaine peuvent être identifiés directement à partir de la connaissance du domaine ou par interview des experts métier.

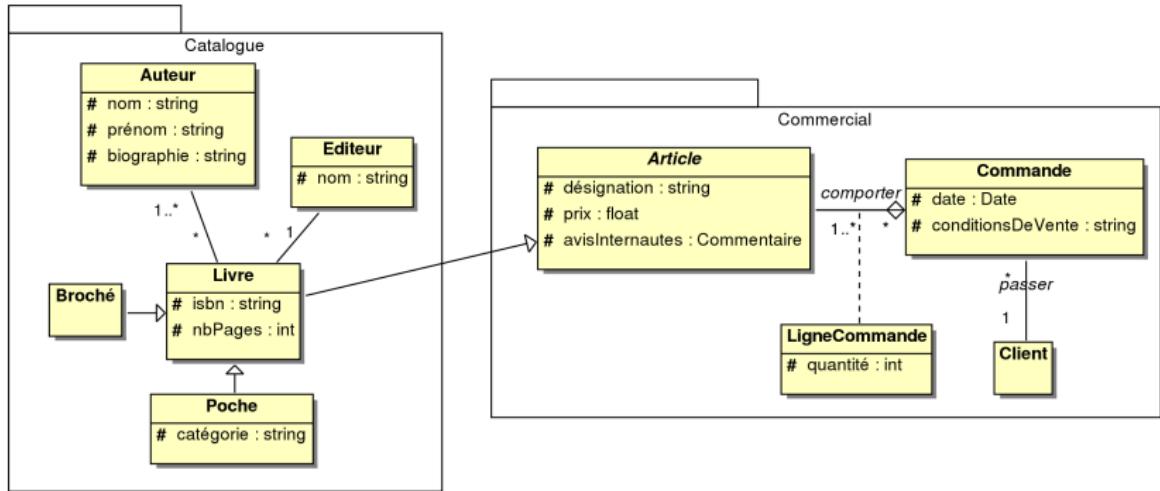


# Exemple de modèle du domaine



# Structuration en packages

112



## Séquences système

Les séquences système formalisent les descriptions textuelles des cas d'utilisation, en utilisant des diagrammes de séquence.

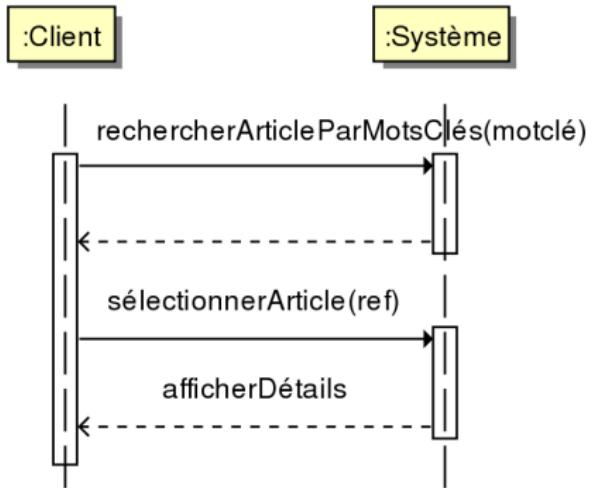
- ▶ Construire les **Diagrammes de Séquences Système** implique souvent la mise à jour des cas d'utilisation à la lumière des réflexions que nous inspirent la production des DSS.
- ▶ Les DSS permettent de spécifier les **opérations système**.
- ▶ **Le système est considéré comme un tout :**
  - ▶ On s'intéresse à ses interactions avec les acteurs ;
  - ▶ On utilise une classe "Système" qui – à part les acteurs – donnera lieu à la seule ligne de vie des DSS.

Un nouveau DSS est produit pour chacun des cas d'utilisation.

- ▶ Les DSS sont parfois *très simples* mais ils seront enrichis par la suite.

# Exemple de diagramme de séquence système

114



Système
rechercherArticleParMotsClés()
rechercherArticleParCritères()
sélectionnerArticle()
passerCommande()
changerPrixArticle()
ajouterArticle()
changerQuantité()

...etc...

- ▶ Les opérations système sont des opérations qui devront être réalisées par l'une ou l'autre des classes du système.
- ▶ Elles correspondent à tous les messages qui viennent des acteurs vers le système dans les différents DSS.

# Classes participantes

- ▶ Pour chaque cas d'utilisation, on définit les classes d'analyse mises en oeuvre pour sa réalisation effective.
- ▶ Typologie des classes d'analyse :
  - ▶ Les **classes métier** (ou entités) représentent les objets métier. Elles correspondent aux classes du modèle du domaine.
  - ▶ Les **classes de dialogue** sont celles qui permettent les interactions entre les acteurs et l'application.
  - ▶ Les **classes de contrôle** permettent d'abstraire les fonctionnalités du système :
    - ▶ Elles font le lien entre les classes dialogue et les classes métier.
    - ▶ Elles permettent de contrôler la cinématique de l'application, cad l'ordre dans lequel les choses doivent se dérouler.

# Diagramme de classes participantes

Le **Diagramme des Classes Participantes** est un diagramme de classes décrivant toutes les classes d'analyse.

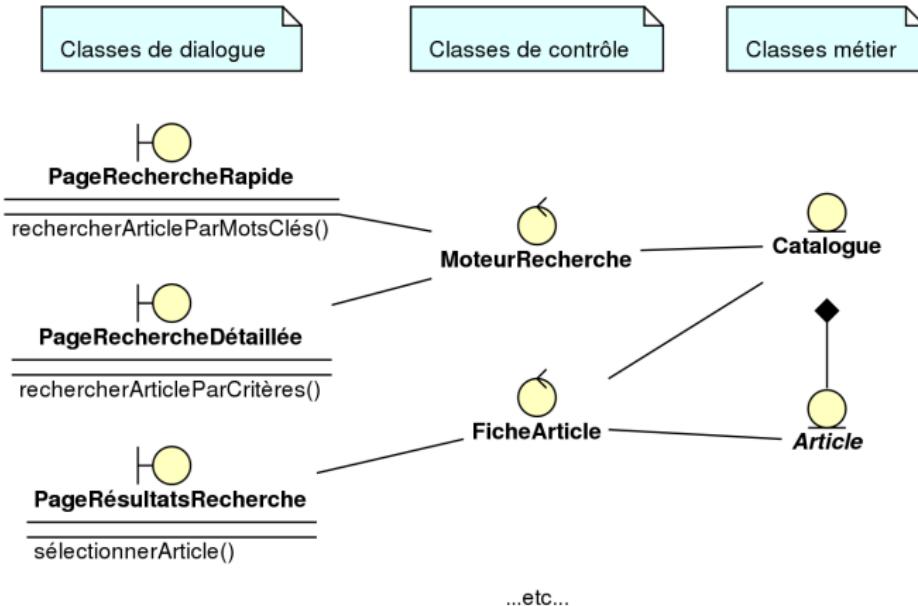
Le DCP est une version enrichie du modèle du domaine, auquel on adjoint les classes d'interaction et de contrôle.

- ▶ A ce point du développement, seules les classes de dialogue ont des **opérations**, qui correspondent aux **opérations système**, c'est à dire aux messages échangés avec les acteurs, que seules les classes de dialogues sont habilitées à intercepter ou à émettre.
- ▶ **Architecture en couches** :
  - ▶ Les dialogues ne peuvent être reliés qu'aux contrôles ou à d'autres dialogues (en général, associations unidirectionnelles).
  - ▶ Les classes métier ne peuvent être reliées qu'aux contrôles ou à d'autres classes métier.
  - ▶ Les contrôles peuvent être associés à tous les types de classes.



# Exemple de diagramme de classes participantes

118



# Diagrammes d'interaction

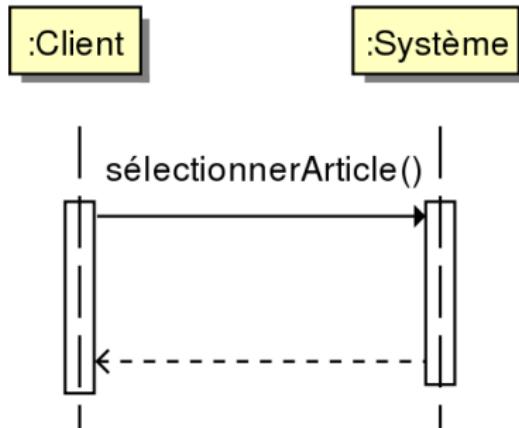
- ▶ Dans les diagrammes de séquence système, le système était vu comme une boîte noire (ligne de vie "Système").
  - ▶ On sait maintenant de quels objets est composé le système (diag. de classes participantes).
  - ▶ Le système n'est plus une boîte noire.

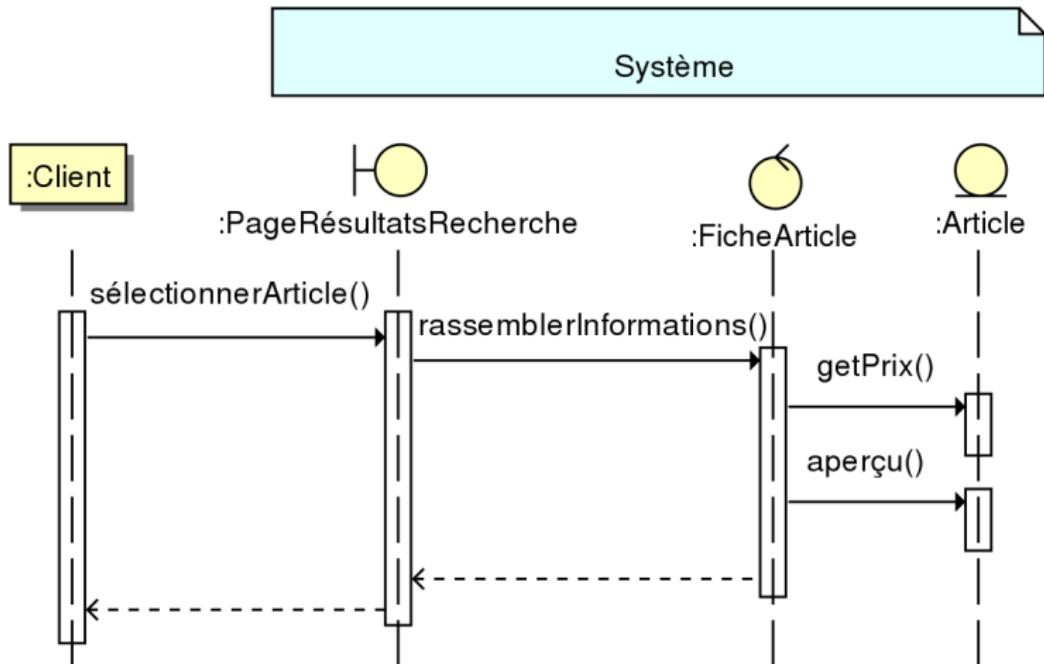
Chaque diagramme de séquence système donne lieu à un diagramme d'interaction. Il y en a donc autant que de cas d'utilisation.

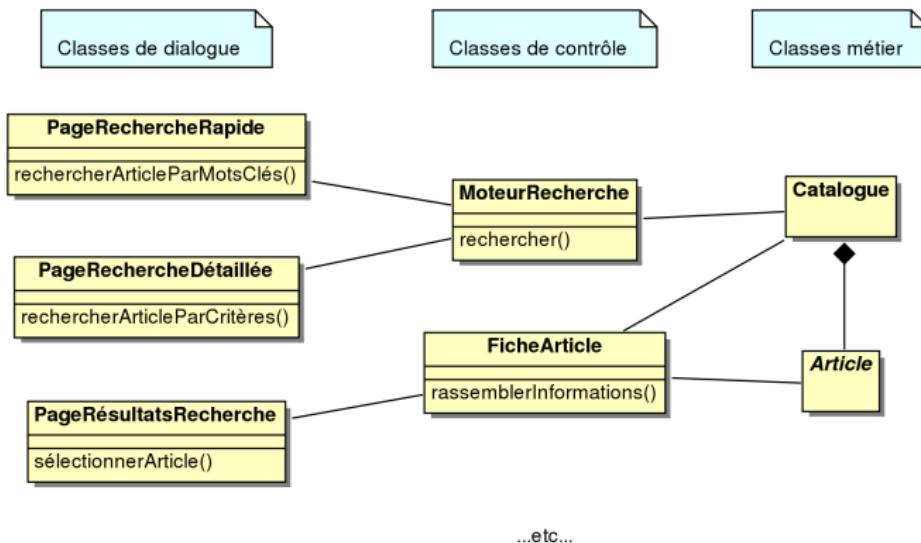
En plus des interactions du système avec l'extérieur, les **Diagrammes d'Interaction** montrent les interactions internes provoquées.

- ▶ Les DSS sont repris mais l'objet "Système" est éclaté pour donner le détail des classes d'analyse :
  - ▶ **Les lignes de vie correspondent aux classes participantes.**









- ▶ Les diagrammes d'interaction permettent de définir les opérations (nécessaires et suffisantes) des classes métier et de contrôle (messages synchrones).

**Le Diagramme des Classes de Conception** reprend le diagramme de classes participantes en y adjoignant toutes les opérations nécessaires.

- ▶ Le DCC est en outre enrichi pour :
  - ▶ Prendre en compte l'architecture logicielle hôte ;
  - ▶ Modéliser les opérations privées des différentes classes ;
  - ▶ Finaliser le modèle des classes avant l'Implantation.

## Conception objet élémentaire avec UML

### UML et méthodologie

Des besoins au code avec UML : une méthode minimale

Rational Unified Process

eXtreme programming

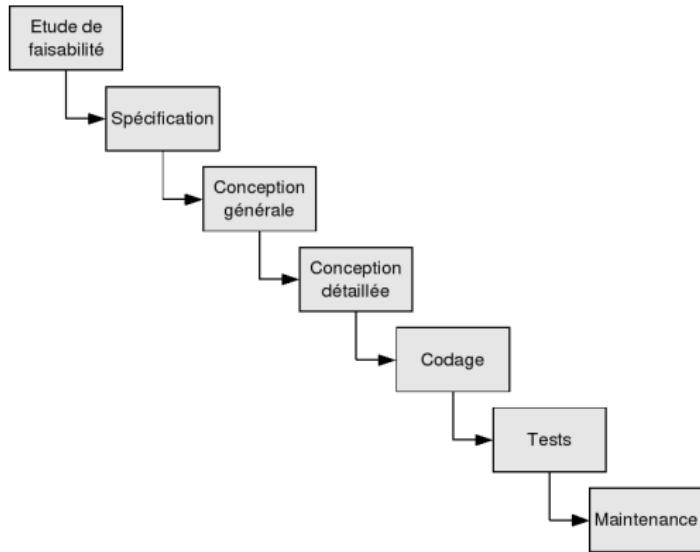
## Modélisation avancée avec UML

## Bonnes pratiques de la modélisation objet



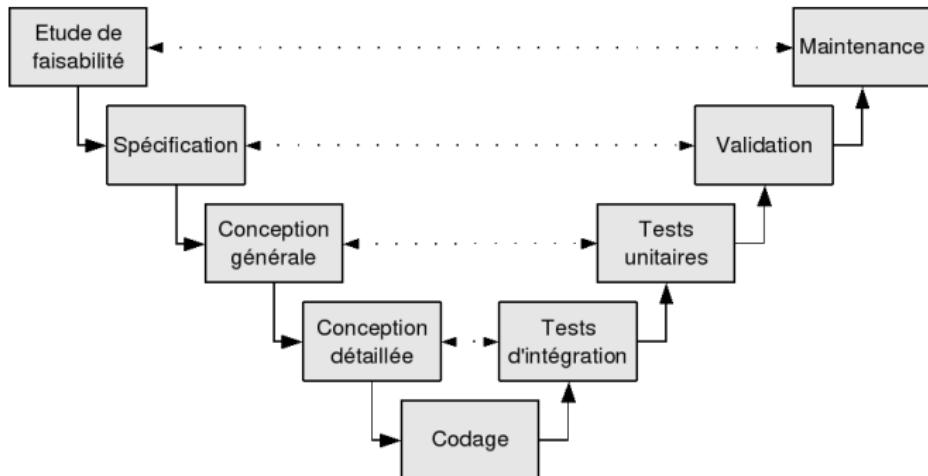
# Modèles de cycles de vie linéaire

- ▶ Les phases du développement se suivent dans l'ordre et sans retour en arrière.



# Modèles de cycles de vie linéaire

- ▶ Les phases du développement se suivent dans l'ordre et sans retour en arrière.



- ▶ Risques élevés et non contrôlés :
  - ▶ Identification **tardive** des problèmes ;
  - ▶ Preuve **tardive** de bon fonctionnement ;
  - ▶ “**Effet tunnel**.”
- ▶ Améliorations : construction **itérative** du système :
  - ▶ Chaque itération produit un nouvel **incrément** ;
  - ▶ Chaque nouvel incrément a pour objectif la maîtrise d'une partie des risques et apporte une preuve tangible de faisabilité ou d'adéquation :
    - ▶ Enrichissement d'une série de prototypes ;
    - ▶ Les versions livrées correspondent à des étapes de la chaîne des prototypes.

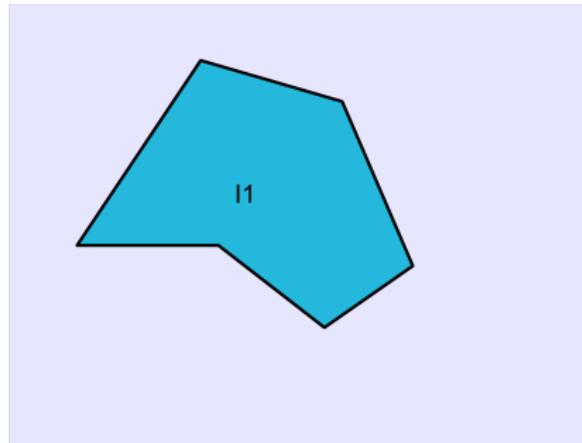
## ► Itérations 0



- A chaque itération, on refait :
  1. Spécification ;
  2. Conception ;
  3. Implémentation ;
  4. Tests.

# Production itérative d'incrément

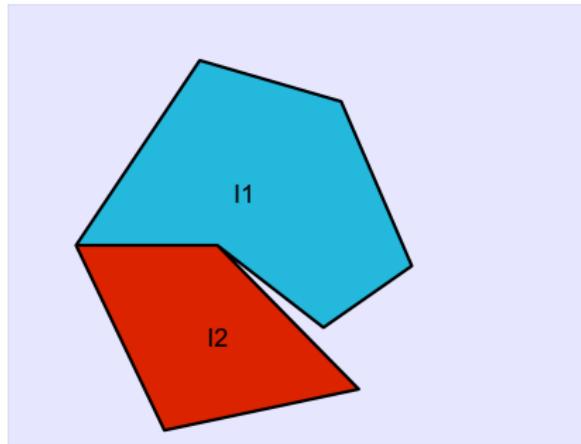
## ► Itérations 1



- A chaque itération, on refait :
  1. Spécification ;
  2. Conception ;
  3. Implémentation ;
  4. Tests.

# Production itérative d'incrément

## ► Itérations 2

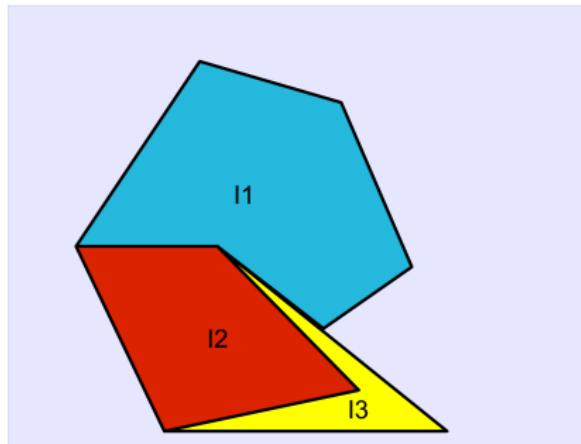


- A chaque itération, on refait :
  1. Spécification ;
  2. Conception ;
  3. Implémentation ;
  4. Tests.

# Production itérative d'incrément

## ► Itérations

3

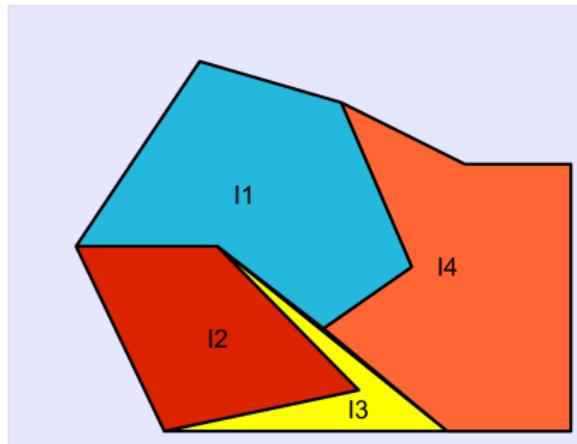


- A chaque itération, on refait :
  1. Spécification ;
  2. Conception ;
  3. Implémentation ;
  4. Tests.

# Production itérative d'incrément

## ► Itérations

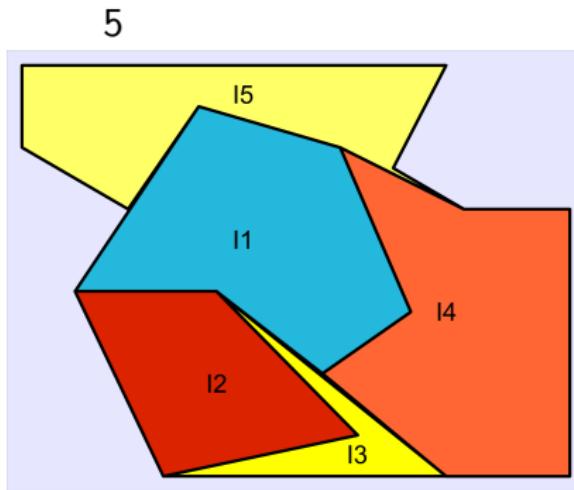
4



- A chaque itération, on refait :
  1. Spécification ;
  2. Conception ;
  3. Implémentation ;
  4. Tests.

# Production itérative d'incrément

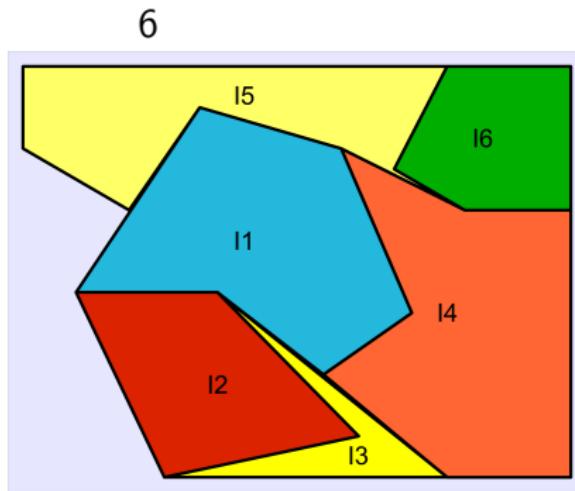
## ► Itérations



- A chaque itération, on refait :
  1. Spécification ;
  2. Conception ;
  3. Implémentation ;
  4. Tests.

# Production itérative d'incrément

## ► Itérations

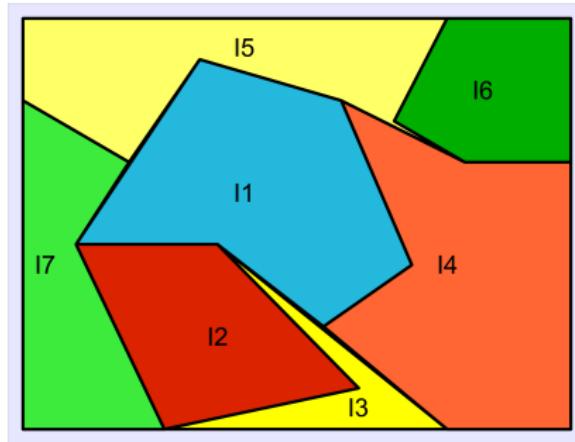


- A chaque itération, on refait :
  1. Spécification ;
  2. Conception ;
  3. Implémentation ;
  4. Tests.

# Production itérative d'incrément

## ► Itérations

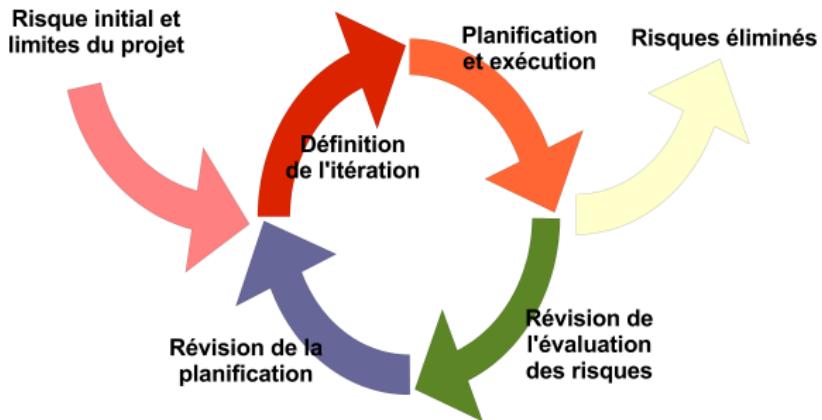
7



- A chaque itération, on refait :
  1. Spécification ;
  2. Conception ;
  3. Implémentation ;
  4. Tests.

# Elimination des risques à chaque itération

On peut voir le développement d'un logiciel comme un processus graduel d'élimination de risques.



- ▶ C'est pendant “**Planification et exécution**” qu'on répète Spécification → Conception → Implémentation → Tests.

RUP est une démarche de développement qui est souvent utilisé conjointement au langage UML.

► Rational Unified Process est

- Piloté par les cas d'utilisation ;
- Centré sur l'architecture ;
- Itératif et incrémental.

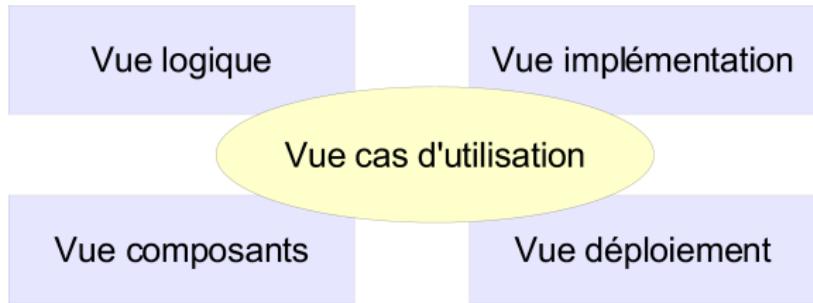
## RUP est itératif et incrémental

- ▶ Chaque itération prend en compte un certain nombre de cas d'utilisation.
- ▶ Les risques majeurs sont traités en priorité.
- ▶ Chaque itération donne lieu à un incrément et produit une nouvelle version exécutable.



- ▶ La principale qualité d'un logiciel est son **utilité** :
  - ▶ Adéquation du service rendu par le logiciel avec les besoins des utilisateurs.
- ▶ Le développement d'un logiciel doit être centré sur l'utilisateur.
- ▶ Les cas d'utilisation permettent d'exprimer ces besoins :
  - ▶ Détection et description des besoins fonctionnels ;
  - ▶ Organisation des besoins fonctionnels.

- ▶ Modélisation de différentes perspectives indépendantes et complémentaires.
- ▶ **Architecture en couches et vues de Krutchen.**



# Vues du système

## ► Vue cas d'utilisation :

- Description du système comme un ensemble de transactions du point de vue de l'utilisateur.

## ► Vue logique :

- Créeée lors de la phase d'élaboration et raffinéee lors de la phase de construction ;
- Utilisation de diagrammes de classes, de séquences...

## ► Vue composants :

- Description de l'architecture logicielle.

## ► Vue déploiement :

- Description de l'architecture matérielle du système.

## ► Vue implémentation :

- Description des algorithmes, code source.

- ▶ **Initialisation :**
  - ▶ Définition du problème.
- ▶ **Elaboration :**
  - ▶ Planification des activités, affectation des ressources, analyse.
- ▶ **Construction :**
  - ▶ Développement du logiciel par incrémentations successifs.
- ▶ **Transition :**
  - ▶ Recettage et déploiement.

Les phases du développement sont les grandes étapes du développement du logiciel

- ▶ Le projet commence en phase d'initialisation et termine en phase de transition

- ▶ Définition du cadre du projet, son concept, et inventaire du contenu ;
- ▶ Elaboration des cas d'utilisation critiques ayant le plus d'influence sur l'architecture et la conception ;
- ▶ Réalisation d'un ou de plusieurs prototypes démontrant les fonctionnalités décrites par les cas d'utilisation principaux ;
- ▶ Estimation détaillée de la charge de travail, du coût et du planning général ainsi que de la phase suivante d'élaboration  
Estimation des risques.

- ▶ Formulation du cadre du projet, des besoins, des contraintes et des critères d'acceptation ;
- ▶ Planification et préparation de la justification économique du projet et évaluation des alternatives en termes de gestion des risques, ressources, planification ;
- ▶ Synthèse des architectures candidates, évaluation des coûts.

- ▶ Un document de vision présentant les besoins de base, les contraintes et fonctionnalités principales ;
- ▶ Une première version du modèle de cas d'utilisation ;
- ▶ Un glossaire de projet ;
- ▶ Un document de justification économique incluant le contexte général de réalisation, les facteurs de succès et la prévision financière ;
- ▶ Une évaluation des risques ;
- ▶ Un plan de projet présentant phases et itérations ;
- ▶ Un ou plusieurs prototypes.

- ▶ Un consensus sur :
  - ▶ La planification ;
  - ▶ L'estimation des coûts ;
  - ▶ La définition de l'ensemble des projets des parties concernées.
- ▶ La compréhension commune des besoins.

- ▶ Définir, valider et arrêter l'architecture ;
- ▶ Démontrer l'efficacité de cette architecture à répondre à notre besoin ;
- ▶ Planifier la phase de construction.

- ▶ Elaboration de la vision générale du système, les cas d'utilisation principaux sont compris et validés ;
- ▶ Le processus de projet, l'infrastructure, les outils et l'environnement de développement sont établis et mis en place ;
- ▶ Elaboration de l'architecture et sélection des composants.

- ▶ Le modèle de cas d'utilisation est produit au moins à 80 % ;
- ▶ La liste des exigences et contraintes non fonctionnelles identifiées ;
- ▶ Une description de l'architecture ;
- ▶ Un exécutable permettant de valider l'architecture du logiciel à travers certaines fonctionnalités complexes ;
- ▶ La liste des risques revue et la mise à jour de la justification économique du projet ;
- ▶ Le plan de réalisation, y compris un plan de développement présentant les phases, les itérations et les critères d'évaluation ;

- ▶ La stabilité de la vision du produit final ;
- ▶ La stabilité de l'architecture ;
- ▶ La prise en charge des risques principaux est adressée par le(s) prototype(s) ;
- ▶ La définition et le détail du plan de projet pour la phase de construction ;
- ▶ Un consensus, par toutes les parties prenantes, sur la réactualisation de la planification, des coûts et de la définition de projet.

- ▶ La minimisation des coûts de développement par :
  - ▶ L'optimisation des ressources ;
  - ▶ La minimisation des travaux non nécessaires.
- ▶ Le maintien de la qualité ;
- ▶ Réalisation des versions exécutables.

- ▶ Gestion et le contrôle des ressources et l'optimisation du processus de projet ;
- ▶ Evaluation des versions produites en regard des critères d'acceptation définis.

- ▶ Les versions exécutables du logiciel correspondant à l'enrichissement itération par itération des fonctionnalités ;
- ▶ Les manuels d'utilisation réalisés en parallèle à la livraison incrémentale des exécutables ;
- ▶ Une description des versions produites.

- ▶ La stabilité et la qualité des exécutables ;
- ▶ La préparation des parties prenantes ;
- ▶ La situation financière du projet en regard du budget initial.

- ▶ Le déploiement du logiciel dans l'environnement d'exploitation des utilisateurs ;
- ▶ La prise en charge des problèmes liés à la transition ;
- ▶ Atteindre un niveau de stabilité tel que l'utilisateur est indépendant ;
- ▶ Atteindre un niveau de stabilité et qualité tel que les parties prenantes considèrent le projet comme terminé.

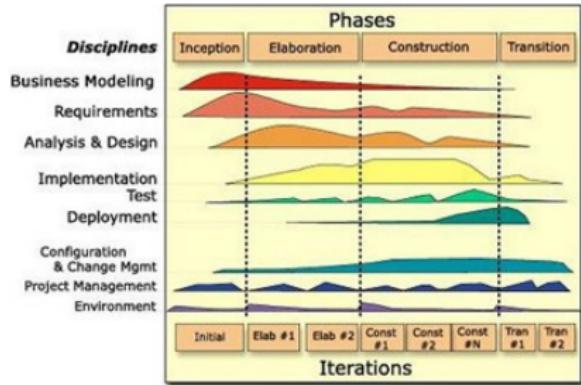
- ▶ Activités de “packaging” du logiciel pour le mettre à disposition des utilisateurs et de l'équipe d'exploitation ;
- ▶ Correction des erreurs résiduelles et amélioration de la performance et du champ d'utilisation ;
- ▶ Evaluation du produit final en regard des critères d'acceptation définis.

- ▶ La version finale du logiciel ;
- ▶ Les manuels d'utilisation.

- ▶ La satisfaction des utilisateurs ;
- ▶ La situation financière du projet en regard du budget initial.

# Organisation en activités de développement

- ▶ Chaque phase comprend plusieurs itérations
- ▶ Pour chacune des itérations, on se livre à plusieurs activités :
  - ▶ Modélisation métier ;
  - ▶ **Expression des besoins** ;
  - ▶ Analyse ;
  - ▶ Conception ;
  - ▶ Implémentation ;
  - ▶ Test ;
  - ▶ Déploiement.



- ▶ Les **activités** sont des étapes dans le développement d'un logiciel, mais à un niveau de granularité beaucoup plus fin que les phases.
- ▶ Chaque activité est répétée autant de fois qu'il y a d'itérations.



- ▶ **Objectif :** Mieux comprendre la structure et la dynamique de l'organisation :
  - ▶ Proposer la meilleure solution dans le contexte de l'organisation cliente ;
  - ▶ Réalisation d'un glossaire des termes métiers ;
  - ▶ Cartographie des processus métier de l'organisation cliente.
- ▶ Activité coûteuse mais qui permet d'accélérer la compréhension d'un problème.

- ▶ **Objectif :** Cibler les besoins des utilisateurs et du clients grâce à une série d'interviews.
  - ▶ L'ensemble des parties prenantes du projet, maîtrise d'oeuvre et maîtrise d'ouvrage, est acteur de cette activité.
- ▶ L'activité de recueil et d'expression des besoins débouche sur ce que doit faire le système (question “**QUOI ?**”)

- ▶ Utilisation des **cas d'utilisation** pour :
  - ▶ Schématiser les besoins ;
  - ▶ Structurer les documents de spécifications fonctionnelles.
- ▶ Les cas d'utilisation sont décomposés en scénarios d'usage du système, dans lesquels l'utilisateur "raconte" ce qu'il fait grâce au système et ses interactions avec le système.
- ▶ Un maquettage est réalisable pour mieux "immerger" l'utilisateur dans le futur système.
- ▶ Une fois posées les limites fonctionnelles, le projet est planifié et une prévision des coûts est réalisée.

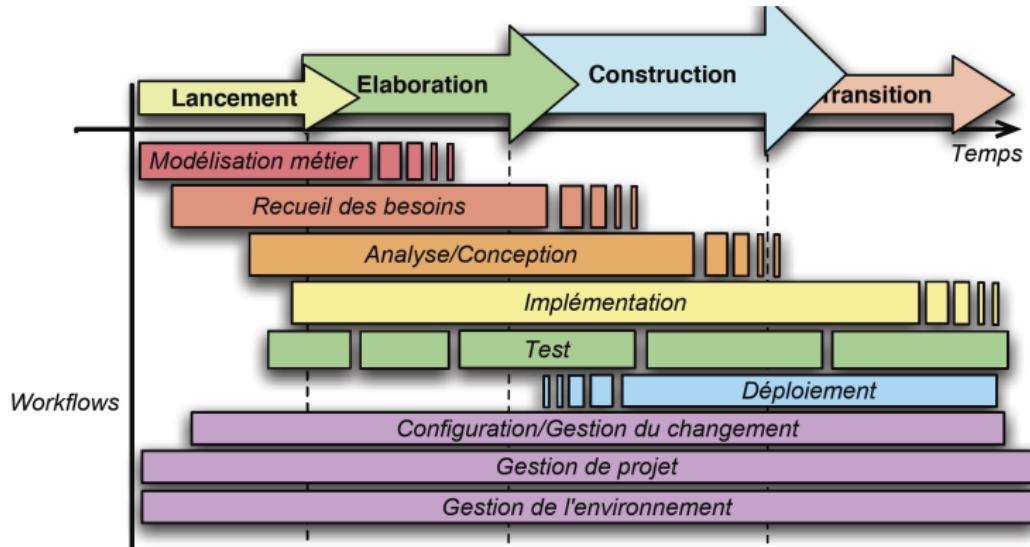
- ▶ **Objectif** : Transformer les besoins utilisateurs en modèles UML.
  - ▶ Analyse objet servant de base à une réflexion sur les mécanismes internes du système.
- ▶ Principaux livrables :
  - ▶ Modèles d'analyse, neutre vis à vis d'une technologie ;
  - ▶ Livre une spécification plus précise des besoins.
- ▶ Peut envisagé comme une première ébauche du modèle de conception.

- ▶ **Objectif :** Modéliser **comment** le système va fonctionner :
  - ▶ Exigences non fonctionnelles ;
  - ▶ Choix technologiques.
- ▶ Le système est analysé et on produit :
  - ▶ Une proposition d'architecture ;
  - ▶ Un découpage en composants.

- ▶ **Objectif :** Implémenter le système par composants.
  - ▶ Le système est développé par morceaux dépendant les uns des autres.
  - ▶ Optimisation de l'utilisation des ressources selon leurs expertises.
- ▶ Les découpages fonctionnel et en couches sont indispensable pour cette activité.
- ▶ Il est tout à fait envisageable de retoucher les modèles d'analyse et de conception à ce stade.

- ▶ **Objectif** : Vérifier des résultats de l'implémentation en testant la construction :
  - ▶ **Tests unitaires** : tests composants par composants ;
  - ▶ **Tests d'intégration** : tests de l'interaction de composants préalablement testés individuellement.
- ▶ Méthode :
  - ▶ Planification pour chaque itération ;
  - ▶ Implémentation des tests en créant des cas de tests ;
  - ▶ Exécuter les tests ;
  - ▶ Prendre en compte le résultat de chacun.

- ▶ **Objectif :** Déployer les développements une fois réalisés.
  - ▶ Peut être réalisé très tôt dans le processus dans une sousactivité de prototypage dont l'objectif est de valider :
    - ▶ L'architecture physique ;
    - ▶ Les choix technologiques.



- ▶ Expression des besoins et modélisation métier :
  - ▶ Modèles métier, domaine, cas d'utilisation ;
  - ▶ Diagramme de séquences ;
  - ▶ Diagramme d'activité.
- ▶ Analyse
  - ▶ Modèles métier, cas d'utilisation ;
  - ▶ Diagramme des classes, de séquences et de déploiement.
- ▶ Conception
  - ▶ Diagramme des classes, de séquences ;
  - ▶ Diagramme état/transition ;
  - ▶ Diagramme d'activité ;
  - ▶ Diagramme de déploiement et de composant.

- ▶ **2TUP**, avec un processus de développement en “Y”, développé par Valtech.
  - ▶ “UML 2.0, en action : De l’analyse des besoins à la conception J2EE”
    - ▶ **Pascal Roques, Franck Vallée**
    - ▶ Editions Eyrolles (2004)

## Conception objet élémentaire avec UML

### UML et méthodologie

Des besoins au code avec UML : une méthode minimale

Rational Unified Process

eXtreme programming

## Modélisation avancée avec UML

## Bonnes pratiques de la modélisation objet



“Quelles activités pouvons nous abandonner tout en produisant des logiciels de qualité ?”

“Comment mieux travailler avec le client pour nous focaliser sur ses besoins les plus prioritaires et être aussi réactifs que possible ?”

- ▶ Filiation avec le RAD.
- ▶ Exemples de méthodes agiles :
  - ▶ **XP** (eXtreme Programming), **DSDM** (Dynamic Software Development Method), **ASD** (Adaptative Software Development), **CCM** (Crystal Clear Methodologies), **SCRUM**, **FDD** (Feature Driven Development).

- ▶ Priorité aux personnes et aux interactions sur les procédures de les outils ;
- ▶ Priorité aux applications fonctionnelles sur une documentation pléthorique ;
- ▶ Priorité à la collaboration avec le client sur la négociation de contrat ;
- ▶ Priorité à l'acceptation du changement sur la planification.

eXtreme Programming, une méthode “basée sur des pratiques qui sont autant de boutons poussés au maximum”.

- ▶ Méthode qui peut sembler naturelle mais **concrètement difficile à appliquer et à maîtriser** :
  - ▶ Réclame beaucoup de discipline et de communication (contrairement à la première impression qui peut faire penser à une ébullition de cerveaux individuels).
  - ▶ Aller vite mais sans perdre de vue la rigueur du codage et les fonctions finales de l'application.
- ▶ **Force de XP** : sa simplicité et le fait qu'on va droit à l'essentiel, selon un rythme qui doit rester constant.

# Valeurs d'XP

## ► Communication :

- ▶ XP favorise la communication directe, plutôt que le cloisonnement des activités et les échanges de documents formels.
- ▶ Les développeurs travaillent directement avec la maîtrise d'ouvrage.

## ► Feedback :

- ▶ Les pratiques XP sont conçues pour donner un maximum de feedback sur le déroulement du projet afin de corriger la trajectoire au plus tôt.

## ► SimPLICITÉ :

- ▶ Du processus ;
- ▶ Du code.

## ► Courage :

- ▶ D'honorer les autres valeurs ;
- ▶ De maintenir une communication franche et ouverte ;
- ▶ D'accepter et de traiter de front les mauvaises nouvelles.

- ▶ XP est fondé sur des valeurs, mais surtout sur **13 pratiques** réparties en 3 catégories :
  - ▶ Gestion de projets ;
  - ▶ Programmation ;
  - ▶ Collaboration.

# Pratiques de gestion de projets

## ► Livraisons fréquentes :

- ▶ L'équipe vise la mise en production rapide d'une version minimale du logiciel, puis elle fournit ensuite régulièrement de nouvelles livraisons en tenant compte des retours du client.

## ► Planification itérative :

- ▶ Un plan de développement est préparé au début du projet, puis il est revu et remanié tout au long du développement pour tenir compte de l'expérience acquise par le client et l'équipe de développement.

## ► Client sur site :

- ▶ Le client est intégré à l'équipe de développement pour répondre aux questions des développeurs et définir les tests fonctionnels.

## ► Rythme durable :

- ▶ L'équipe adopte un rythme de travail qui lui permet de fournir un travail de qualité tout au long du projet.
- ▶ Jamais plus de 40h de travail par semaine (un développeur fatigué développe mal).

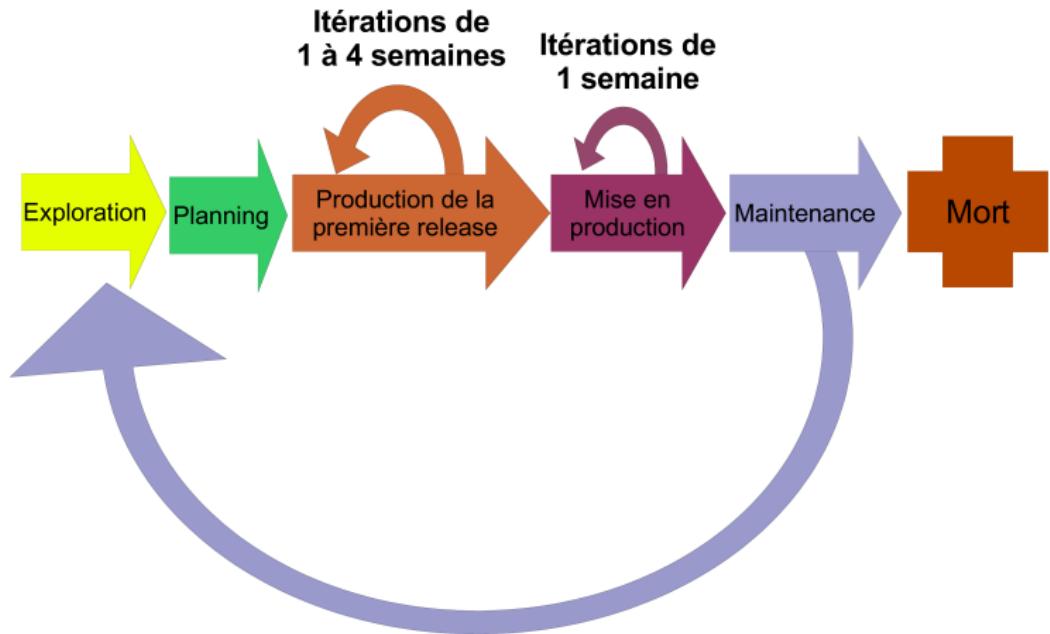


- ▶ **Conception simple :**
  - ▶ On ne développe rien qui ne soit utile tout de suite.
- ▶ **Remaniement :**
  - ▶ Le code est en permanence réorganisé pour rester aussi clair et simple que possible.
- ▶ **Tests unitaires :**
  - ▶ Les développeurs mettent en place une batterie de tests de nonrégression qui leur permettent de faire des modifications sans crainte.
- ▶ **Tests de recette :**
  - ▶ Les testeurs mettent en place des tests automatiques qui vérifient que le logiciel répond aux exigences du client.
  - ▶ Ces tests permettent des recettes automatiques du logiciel.

# Pratiques de collaboration

- ▶ Responsabilité collective du code :
  - ▶ Chaque développeur est susceptible de travailler sur n'importe quelle partie de l'application.
- ▶ Programmation en binômes :
  - ▶ Les développeurs travaillent toujours en binômes, ces binômes étant renouvelés fréquemment.
- ▶ Règles de codage :
  - ▶ Les développeurs se plient à des règles de codage strictes définies par l'équipe elle-même.
- ▶ Métaphore :
  - ▶ Les développeurs s'appuient sur une description commune du design.
- ▶ Intégration continue :
  - ▶ L'intégration des nouveaux développements est faite chaque jour.





- ▶ Les développeurs se penchent sur des **questions techniques** :
  - ▶ Explorer les différentes possibilités d'architecture pour le système :
  - ▶ Etudier par exemple les limites au niveau des performances présentées par chacune des solutions possibles.
- ▶ Le client s'habitue à **exprimer ses besoins** sous forme de user stories (proximes de diagrammes de cas illustrés par des diagrammes de séquences).
  - ▶ Les développeurs estiment les temps de développement.

- ▶ Planning de la première release :
  - ▶ Uniquement les fonctionnalités essentielles ;
  - ▶ Première release à enrichir par la suite.
- ▶ Durée du planning : 1 ou 2 jours.
- ▶ Première version (release) au bout de 2 à 6 mois.

- ▶ Développement de la **première version de l'application**.
- ▶ Itérations de une à quatre semaines :
  - ▶ Chaque itération produit un sous ensemble des fonctionnalités principales ;
  - ▶ Le produit de chaque itération subit des tests fonctionnels ;
  - ▶ Itérations courtes pour identifier très tôt des déviation par rapport au planning.
- ▶ Brèves réunions quotidiennes réunissant toute l'équipe, pour mettre chacun au courant de l'avancement du projet.

- ▶ La mise en production produit un logiciel :
  - ▶ Offrant toutes les fonctionnalités indispensables ;
  - ▶ Parfaitement fonctionnel ;
  - ▶ Mis à disposition des utilisateurs.
- ▶ Itérations très courtes ;
- ▶ Tests constants en parallèle du développement ;
- ▶ Les développeurs procèdent à des **réglages affinés** pour améliorer les performances du logiciel.

- ▶ Continuer à **faire fonctionner le système** ;
- ▶ Adjonction de **nouvelles fonctionnalités secondaires**.
  - ▶ Pour les fonctionnalités secondaires, on recommence par une rapide exploration.
  - ▶ L'ajout de fonctionnalités secondaires donne lieu à de nouvelles releases.

- ▶ Quand le client ne parvient plus à spécifier de nouveaux besoins, le projet est dit “mort”
  - ▶ Soit que tous les besoins possibles sont remplis ;
  - ▶ Soit que le système ne supporte plus de nouvelles modifications en restant rentable.

- ▶ Pour un travail en équipe, on distingue 6 rôles principaux au sein d'une équipe XP :
  - ▶ Développeur ;
  - ▶ Client ;
  - ▶ Testeur ;
  - ▶ Tracker ;
  - ▶ Manager ;
  - ▶ Coach.

- ▶ Conception et programmation, même combat !
- ▶ Participe aux séances de planification, évalue les tâches et leur difficulté ;
- ▶ Définition des test unitaires ;
- ▶ Implémentation des fonctionnalités et des tests unitaires.

- ▶ Écrit, explique et maîtrise les scénarios ;
- ▶ Spécifie les tests fonctionnels de recette ;
- ▶ Définit les priorités.



- ▶ Écriture des tests de recette automatiques pour valider les scénarios clients ;
- ▶ Peut influer sur les choix du clients en fonction de la "testabilité" des scénarios.

- ▶ Suivre le planning pour chaque itération ;
- ▶ Comprendre les estimations produites par les développeurs concernant leur charges ;
- ▶ Interagir avec les développeurs pour le respect du planning de l'itération courante ;
- ▶ Détection des éventuels retards et rectifications si besoin.

- ▶ Supérieur hiérarchique des développeurs :
  - ▶ Responsable du projet.
- ▶ Vérification de la satisfaction du client ;
- ▶ Contrôle le planning ;
- ▶ Gestion des ressources.

► Garant du processus XP :

- Organise et anime les séances de planifications ;
- Favorise la créativité du groupe, n'impose pas ses solutions techniques ;
- Coup de gueules... .

- ▶ Pas de documents d'analyse ou de spécifications détaillées.
- ▶ **Les tests de recette remplacent les spécifications.**
- ▶ Emergence de l'architecture au fur et à mesure du développement.

► Inconvénients de XP :

- Focalisation sur l'aspect individuel du développement, au détriment d'une vue globale et des pratiques de management ou de formalisation ;
- Manquer de contrôle et de structuration, risques de dérive.

► Inconvénients de RUP :

- Fait tout, mais lourd, "usine à gaz";
- Parfois difficile à mettre en oeuvre de façon spécifique.

- XP pour les petits projets en équipe de 12 max ;
- RUP pour les gros projets qui génèrent beaucoup de documentation.

Conception objet élémentaire avec UML

UML et méthodologie

Modélisation avancée avec UML

Expression de contraintes avec OCL

Diagrammes d'états-transitions

Diagrammes d'activités

Diagrammes de communication

Diagrammes de composants et de déploiement

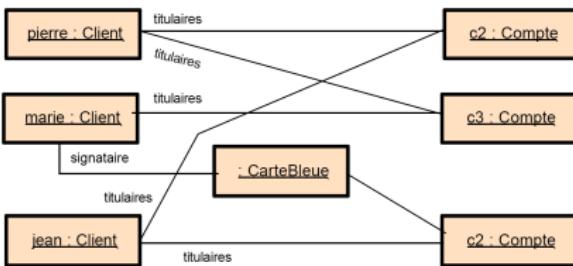
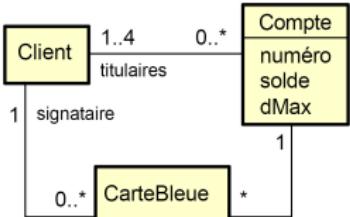
Bonnes pratiques de la modélisation objet



- ▶ Les différents diagrammes d'UML expriment en fait des **contraintes**
  - ▶ Graphiquement
    - ▶ Contraintes structurelles (un attribut dans une classe)
    - ▶ Contraintes de types (sous-typage)
    - ▶ Contraintes diverses (composition, cardinalité, etc.)
  - ▶ Via des propriétés prédéfinies
    - ▶ Sur des classes (`{abstract}`)
    - ▶ Sur des rôles (`{ordered}`)
- ▶ C'est toutefois insuffisant

- Certaines contraintes "évidentes" sont difficilement exprimables avec UML seul.

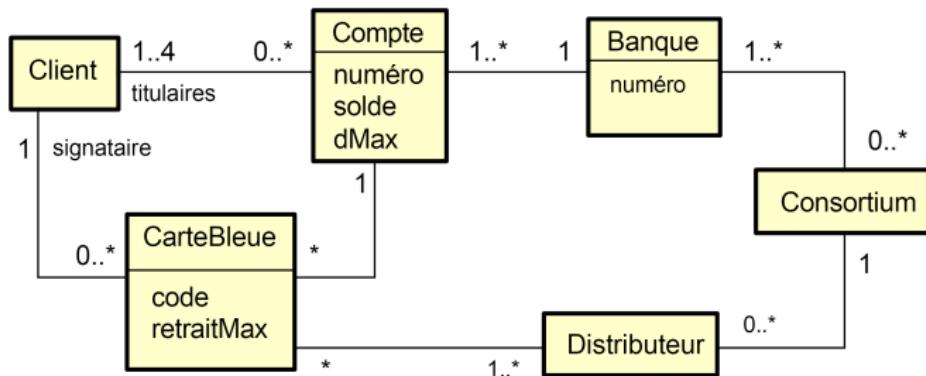
"Le signataire d'une carte bleue est titulaire du compte correspondant"



- ▶ **Simple** à mettre en oeuvre :
  - ▶ Utilisation des notes en UML + texte libre,
  - ▶ Compréhensible par tous.
- ▶ **Indispensable** :
  - ▶ Documenter les contraintes est essentiel,
  - ▶ Déetecter les problèmes le plus tôt possible.
- ▶ **Probématique**
  - ▶ **Ambigu**, imprécis,
  - ▶ Difficile d'exprimer clairement des contraintes complexes,
  - ▶ Difficile de lier le texte aux éléments du modèle.

# Exemples de contraintes exprimables en OCL

- ▶ Le solde d'un compte ne doit pas être inférieur au découvert maximum.
- ▶ Le signataire d'une carte bleue associée à un compte en est le titulaire.
- ▶ Une carte bleue est acceptée dans tous les distributeurs des consortiums de la banque.
- ▶ ...



- ▶ Si l'attribut dMax de la classe Compte est un réel et que le découvert est exprimé par une valeur négative :

```
context c~: Compte  
inv: soldé >= dMax
```

- ▶ Si le découvert est exprimé par un nombre positif (par ex : 300 euros si on ne doit pas descendre en dessous de 300 euros)

```
context c~: Compte  
inv: soldé >= -dMax
```

- ▶ ... est **le** titulaire ...

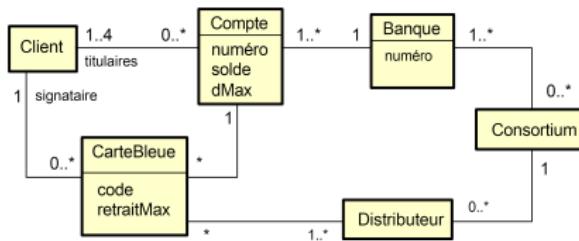
```
context CarteBleue
    inv: signataire = compte.titulaires
```

- ▶ ... est **un des** titulaires ...

```
context CarteBleue
    inv: compte.titulaires -> includes(signa
```

# “Une carte bleue est acceptée dans tous les distributeurs des consortiums de la banque”

192



```
context CarteBleue
inv: distributeur
= compte.banque.consortium.distributeur
```

- ▶ OCL : Object Constraint Language
  - ▶ Langage de requêtes et de contraintes
  - ▶ Relativement simple à écrire et à comprendre
    - ▶ syntaxe purement textuelle sans symboles étranges
- ▶ Parfaitement intégré à UML
  - ▶ Sémantique d'UML écrite en OCL : tous les schémas UML produits ont une traduction en OCL
- ▶ En pleine expansion
  - ▶ Nouvelle version **majeure** avec UML2.0
  - ▶ Essentiel pour avoir des modèles suffisamment précis
  - ▶ De plus en plus d'outils
    - ▶ édition, vérification, génération (partielle) de code...

- ▶ Langage d'expressions (**fonctionnel**)
  - ▶ Valeurs, expressions, types
  - ▶ Fortement typé
  - ▶ Pas d'effets de bords
- ▶ Langage de spécification, pas de programmation
  - ▶ Haut niveau d'abstraction
  - ▶ Pas forcément exécutable (seul un sous-ensemble l'est)
  - ▶ Permet de trouver des erreurs beaucoup plus tôt dans le cycle de vie

## ► Points faibles

- ▶ Pas aussi rigoureux que des langages de spécification formelle comme VDM, Z ou B
  - ▶ Pas de preuves formelles possibles dans tous les cas
- ▶ Puissance d'expression limitée

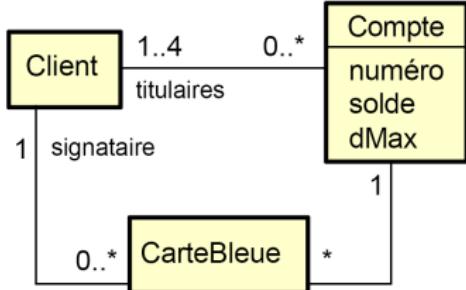
## ► Points forts

- ▶ Relativement simple à écrire et à comprendre
- ▶ Très bien intégré à UML
- ▶ Bon compromis entre simplicité et puissance d'expression
- ▶ Passage vers différentes plateformes technologiques

# Contexte d'une contrainte

- ▶ Une contrainte est toujours associée à un élément de modèle : le **contexte** de la contrainte.
- ▶ Deux techniques pour spécifier le contexte :
  - ▶ En écrivant la contrainte entre accolades {...} dans une **note**. L'élément pointé par la note est alors le contexte de la contrainte
  - ▶ En utilisant le mot clé "context" dans un document quelconque.

```
context CarteBleue
inv: compte.titulaires -> includes(signataire)
inv: code>0 and code<=9999
inv: retraitMax>10
```



- ▶ OCL peut être utilisé pour décrire trois types de prédictats avec les mots clé :
  - ▶ **inv:** invariants de classes

```
inv: solde < dMax
```

- ▶ **pre:** pré-conditions d'opérations

```
pre: montantARetirer > 0
```

- ▶ **post:** post-conditions d'opérations

```
post: solde > solde@pre
```



- ▶ OCL peut également être utilisé pour décrire des **expressions** avec les mots clés :

- ▶ **def:** déclarer des attributs ou des opérations

```
def : nbEnfants : Integer
```

- ▶ **init:** spécifier la valeur initiale des attributs

```
init : enfants -> size()
```

- ▶ **body:** exprimer le corps de méthodes {query}

```
body : enfants -> select(age < a)
```

- ▶ **derive:** définir des éléments dérivés (/)

```
derive : age < 18
```

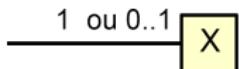
- ▶ Accès à un attribut : `objet.attribut`
  - ▶ Ex : `self.dateDeNaissance`
- ▶ Accès à une méthode `objet.operation(param1, ... )`
  - ▶ Ex : `self.impots(1998)`

- ▶ Accéder à un ensemble d'objets liés à un objet donné

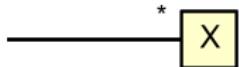
`<objet>. <nomderole>`

- ▶ Le résultat est soit une valeur soit un ensemble
- ▶ Le type du résultat dépend de la multiplicité cible et de la présence de la contrainte {ordered}

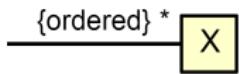
- ▶ X



- ▶ Set(X)



- ▶ OrderedSet(X)



# Notes sur la navigation via les associations

- ▶ Un élément est converti en singleton lorsqu'une opération sur une collection est appliquée

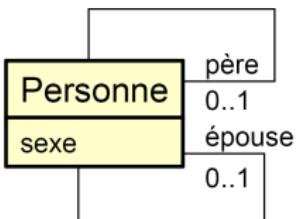
```
pere->size()=1
```

- ▶ La navigation permet de tester si la valeur est définie (l'ensemble vide représente la valeur indéfinie)

```
pere->isEmpty()
```

```
epouse->notEmpty()
```

```
implies self.epouse.sex = sexe::feminin
```

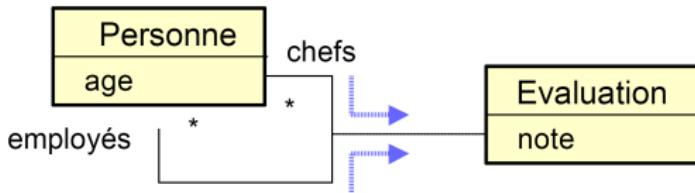


- ▶ Si une association n'a pas de nom de rôle alors on peut utiliser le nom de la classe destination



- ▶ Si l'association est réflexive, pour éviter les ambiguïtés, il faut indiquer avec un rôle entre crochets [...] comment est parcourue l'association

```
objet.nomAssociation[nomDeRole]
```



```
p.evaluation[chefs]  
p.evaluation[employes]  
p.evaluation[chefs].note -> sum()
```

- ▶ Accès qualifié

```
lien.nomderole[valeur1, valeur2, ... ]
```

- ▶ ou ensemble d'objets liés

```
lien.nomderole
```



```
b.compte[4029]
```

```
b.compte
```

```
compte
```

# Invariant (inv)

- ▶ Prédicat associé à une classe ou une association
  - ▶ Doit être vérifié à tout instant
- ▶ Le contexte est défini par un objet
  - ▶ Cet objet peut être référencé par self
- ▶ L'invariant peut être nommé

```
context Personne
    inv pasTropVieux~: age < 110
    inv~: self.age >= 0
```

```
context Personne
inv~: age>0 and self.age<110
inv mariageLegal~: marie implies age > 16
inv enfantsOk~: enfants->size() < 20
inv~: not enfants->includes(self)
inv~: enfants->includesAll(filles)
inv~: enfants->forall(e | self.age-e.age<14)
```



- ▶ Prédicats associés à une opération
  - ▶ Les **pré-conditions** doivent être vérifiées avant l'exécution
  - ▶ Les **post-conditions** sont vraies après l'exécution
- ▶ self désigne l'objet sur lequel l'opération à lieu
- ▶ Dans une post-condition :
  - ▶ @pre permet de faire référence à la valeur avant que opération ne soit effectuée
  - ▶ result désigne le résultat de l'opération
  - ▶ ocsIsNew() indique si un objet n'existait pas dans l'état précédent

```
context Classe::operation(...):ClasseVa
    pre nom1~: param1 < ...
    post~: ... result > ...
```

```
context Personne::retirer(montant:Integer)
  pre~: montant > 0
  post~: solde < solde@pre - montant

context Personne::salaire():integer
  post~: result >= Legislation::salaireMinimum

context Compagnie::embaucheEmploye(p:Personne):Companie
  pre pasPresent~: not employes->includes(p)
  post~: employes=employes@pre->including(p)
  post~: result.ocliIsNew()
  post~: result.compagnie=self and result.employe=p
```

# Définition additionnelle (def)

- ▶ Il est possible en OCL de définir dans une classe existante:
  - ▶ de **nouveaux attributs**
  - ▶ de **nouvelles opérations**

```
context Classe
  def~: nomAtt~: type = expr
  def~: nomOp...~: type = expr
```

- ▶ Utile pour décomposer des requêtes ou contraintes

```
context Personne
  def: ancestres()~:
    Set(Personne) = parents
    ->union(parents.ancestres())
    ->asSet()
  inv: not ancestres() -> includes(self)
```



- ▶ Description d'une méthode sans effet de bord (`{isQuery}`)
- ▶ Équivalent à une requête

```
context Personne:acf(p:Personne):OrderedSet(Personne)
body~: self.ancestres()
      ->select(sexe=Sexe::Feminin)
      ->sortedBy(dateDeNaissance)

context Personne
def: ancestres~: Set(Personne) = parents
      ->union(parents.ancestres->asSet())
```

- ▶ OCL est un langage simple d'expressions
  - ▶ constantes
  - ▶ identificateur
  - ▶ self
  - ▶ expr op expr
  - ▶ exprojet.propobjet
  - ▶ exprojet.propobjet ( parametres )
  - ▶ exprcollection -> propcollection(parametres)
  - ▶ package::package::element
  - ▶ if cond then expr else expr endif
  - ▶ let var : type = expr in expr

- ▶ “.” permet d'accéder à une **propriété d'un objet**
- ▶ “->” permet d'accéder à une **propriété d'une collection**
- ▶ “ ::” permet d'accéder à un **élément** d'un paquetage, d'une classe, ...
- ▶ Des règles permettent de mixer collections et objets

```
self.impots(1998) / self.enfants->size()
self.salaire() -100
self.enfants->select(sexe=Sexe::masculin)
self.enfants->isEmpty()
self.enfants->forall(age>20)
self.enfants->collect(salaire)->sum()
self.enfants.salaire->sum()
self.enfants->union(self.parents)->collect(age)
```

# Types entiers et réels

## ► Integer

- ▶ Valeurs : 1, -5, 34, 24343, ...
- ▶ Opérations : +, -, \*, div, mod, abs, max, min

## ► Real

- ▶ Valeurs : 1.5, 1.34, ...
- ▶ Opérations : +, -, \*, /, floor, round, max, min

► Le type Integer est “conforme” au type Real



# Type booléen

## ► Boolean

- Valeurs : true, false
- Opérations : not, and, or, xor, implies, if-then-else-endif

## ► L'évaluation des opérateurs or, and, if est partielle

- “true or x” est toujours vrai, même si x est indéfini
- “false and x” est toujours faux, même si x est indéfini

```
(age < 40 implies salaire > 1000)
and (age >= 40 implies salaire > 2000)
```

```
if age < 40 then salaire > 1000
else salaire > 2000 endif
```

```
salaire > (if age < 40 then 1000 else 2000 endif)
```

## ► String

- Valeurs : "une phrase"
- Opérations :
  - "="
  - s.size()
  - s1.concat(s2), s1.substring(i1,i2)
  - s.toUpperCase(), s.toLowerCase()

```
nom = nom.substring(1,1)
      .toUpperCase().concat(
          nom.substring(2,nom.size()))
```

- Les chaînes ne sont pas des séquences de caractères
  - String<>Sequence(character), le type character n'existe pas

# Utilisation des valeurs de types énumérés

- ▶ Jour::Mardi
  - ▶ noté "#Mardi" avant UML2.0
- ▶ Opérations
  - ▶ =, <>
  - ▶ Pas de relation d'ordre

<<enumeration>>
Jour
Lundi
Mardi
Mercredi
Jeudi
Vendredi
Samedi
Dimanche

```
epouse ->notEmpty()
implies epouse.sex=Femme::Feminin
```

- ▶ Dans tout langage typé il faut distinguer
  - ▶ un **élément e**
  - ▶ du **singleton** contenant cet élément  $\text{Set}\{e\}$
- ▶ Pour simplifier la navigation OCL, une **conversion implicite** est faite lorsqu'une opération sur une collection est appliquée à un élément isolé
  - ▶ `elem->prop` est équivalent à `Set\{elem\}->prop`
  - ▶ `self->size() = 1`

- ▶ Ensembles : Set(T)
  - ▶ Eléments unique non ordonnés
- ▶ Ensembles ordonnés : OrderedSet(T)
  - ▶ Eléments uniques et ordonnés
- ▶ Sac : Bag(T)
  - ▶ Eléments répétables sans ordre
- ▶ Sequence : Sequence(T)
  - ▶ Eléments répétables mais ordonnés
- ▶ Collection est le type de base Collection(T)

- ▶ Cardinalité : coll  $\rightarrow$  size()
- ▶ Vide : coll  $\rightarrow$  isEmpty()
- ▶ Non vide : coll  $\rightarrow$  nonEmpty()
- ▶ Nombre d'occurrences : coll  $\rightarrow$  count(elem)
- ▶ Appartenance : coll  $\rightarrow$  includes( elem )
- ▶ Non appartenance : coll  $\rightarrow$  excludes( elem )
- ▶ Inclusion : coll  $\rightarrow$  includesAll(coll)
- ▶ Exclusion: coll  $\rightarrow$  excludesAll(coll)
- ▶ Somme des éléments : coll  $\rightarrow$  sum()

- ▶ Union : `ens->union(ens)`
- ▶ Difference : `ens1-ens2`
- ▶ Ajout d'un élément : `ens->including(elem)`
- ▶ Suppression d'un élément : `ens->excluding(elem)`
- ▶ Conversion vers une liste : `ens->asSequence()`
- ▶ Conversion vers un sac : `ens->asBag()`
- ▶ Conv.vers un ens. ord. : `ens->asOrderedSet()`

- ▶ **Select** retient les éléments vérifiant la condition
    - ▶ coll -> select( cond )
  - ▶ **Reject** élimine ces éléments
    - ▶ coll -> reject( cond )
  - ▶ **Any** sélectionne l'un des éléments vérifiant la condition
    - ▶ coll -> any( cond )
    - ▶ opération non déterministe
    - ▶ utile lors de collection ne contenant qu'un élément
    - ▶ retourne la valeur indéfinie si l'ensemble est vide
- ```
self.enfants -> select(age > 10 and sexe = Sexe :: Mas
```

## Autres syntaxes pour le filtrage

- ▶ Il est également possible de
  - ▶ nommer la variable
  - ▶ d'expliciter son type

```
self.employe->select(age > 50)
self.employe->select(p | p.age>50 )
self.employe->select(p:Personne | p.age>50 )
self.employe->reject(p:Personne | p.age<=50 )
```

# Quantificateurs

- ▶ **ForAll** est le quantificateur **universel**

▶ coll -> forAll( cond )

- ▶ **Exists** est le quantificateur **existentiel**

▶ coll -> exists( cond )

```
self.enfants ->forall(age<10)
```

```
self.enfants ->exists(sexe=Sexe::Masculin)
```

- ▶ Il est possible

▶ de nommer la variable

▶ d'expliciter son type

▶ de parcourir plusieurs variables à la fois

```
self.enfants
```

```
->exists(e1,e2 | e1.age=e2.age and e1<)
```

- ▶ Retourne vrai si pour chaque valeur de la collection, l'expression retourne une valeur différente

- ▶ `coll->isUnique(expr)`
- ▶ Equivalence entre

```
    self.enfants ->isUnique(prenom)}  
  
    self.enfants  
    ->forall( e1,e2~: Personne |  
              e1 <> e2 implies  
              e1.prenom <> e2.prenom)
```

- ▶ Utile pour définir la notion de clé en BD

# T-uples

- ▶ Champs nommés, pas d'ordre entre les champs
  - ▶ Tuples (valeurs)

```

Tuple{x=-1.5,y=12.6}
Tuple{y=12.6,x=-1.5}
Tuple{y:Real=12.6,x:Real=-1.5}
Tuple{nom='dupont',prenom='leon',
     age=43}
  
```

- ▶ A partir d'UML 2.0
- ▶ Définition de types tuples

```

TupleType(x:Real,y:Real)
TupleType(y:Real,x:Real)
TupleType(nom:String,prenom:String,
         age:Integer)
  
```



# Collections imbriquées

## ► Collections imbriquées

- Jusqu'à UML 2.0 pas de collections de collections car peu utilisé et plus complexe à comprendre
- Mise à plat intuitive lors de la navigation

```
self.parents.parents
```

- Mise à plat par default lié à l'opération implicite collect
- A partir de UML 2.0 imbrications arbitraires des constructeurs de types

```
Set(Set(Personne))  
Sequence(OrderedSet(Jour))
```

- Très utile pour spécifier des structures non triviales

- ▶ L'opération **collect** met à plat les collections
  - ▶ utile pour naviger

```
enfants.enfants.prenom  
= enfants->collect(enfants)->collect  
= Bag\{'pierre','paul','marie','pau
```

- ▶ **CollectNested** permet de conserver l'imbrication

```
enfants->collectNested(enfants.prenom)  
= Bag{Bag{'pierre','paul'},  
      Bag\{'marie','paul'}}}
```

# Itérateur général

- ▶ L'itérateur le plus général est **iterate**
  - ▶ Les autres itérateurs (forall, exist, select, etc.) en sont des cas particulier
  - ▶ coll -> iterate(

```
coll->iterate(
    elem~: Type1~;
    accumulateur:Type2=<valInit>
    | <expr> )
```

- ▶ Exemple

```
enfants->iterate(
    e:Enfant;
    ac:Integer=0
    | acc+e.age )
```

- ▶ Collections ordonnées
  - ▶ **Sequence**
  - ▶ **OrderedSet**
- ▶ ... mais pas forcément triées
  - ▶ Séquence simple (l'ordre compte) :
    - ▶ Sequence { 12, 23, 5, 5, 12 }
  - ▶ Séquence triée :
    - ▶ Sequence { 5, 5, 12, 12, 23 }

# Tri d'une collection

- ▶ Pour trier une collection en fonction d'une expression
  - ▶ `coll -> sortedBy(expr)`
  - ▶ L'opération “`<`” doit être définie sur le type correspond à expr

```
enfants ->sortedBy(age)
let ages = enfants.age -> sortedBy(
    ages.last() -ages.first()
enfants ->sortedBy(enfants->size()) ->
```

- ▶ Le résultat est de type
  - ▶ `OrderedSet` si l'opération est appliquée sur un `Set`
  - ▶ `Sequence` si l'opération est appliquée sur un `Bag`



Conception objet élémentaire avec UML

UML et méthodologie

Modélisation avancée avec UML

Expression de contraintes avec OCL

Diagrammes d'états-transitions

Diagrammes d'activités

Diagrammes de communication

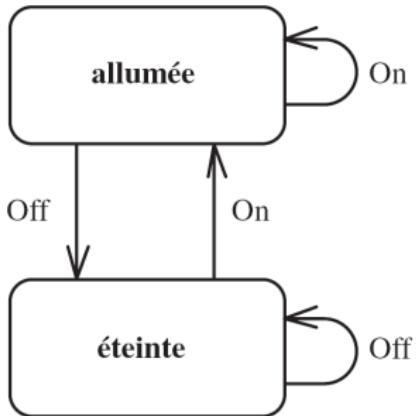
Diagrammes de composants et de déploiement

Bonnes pratiques de la modélisation objet

- ▶ Un automate à états finis est la spécification de la séquence d'états que subira un objet au cours de son cycle de vie.
- ▶ Un tel automate représente le comportement d'un classeur dont les sorties
  - ▶ ne dépendent pas seulement de ses entrées,
  - ▶ mais aussi d'un historique des sollicitations passées.
- ▶ Cet historique est caractérisé par un état.
- ▶ Les objets changent d'état en réponse à des événements extérieurs donnant lieu à des transitions entre états.
- ▶ Sauf cas particuliers, à chaque instant, chaque objet est dans un et un seul état.

# Etat et transition

- ▶ Les états sont représentés par des rectangles aux coins arrondis
- ▶ Les transitions sont représentées par des arcs orientés liant les états entre eux.
- ▶ Certains états, dits “**composites**”, peuvent contenir des sous-diagrammes.



Les diagrammes d'état-transition d'UML représentent en fait des automates à pile avec emboîtement et concurrence et pas seulement des automates à états finis comme dans les premières versions d'UML

# Diagramme d'état-transition

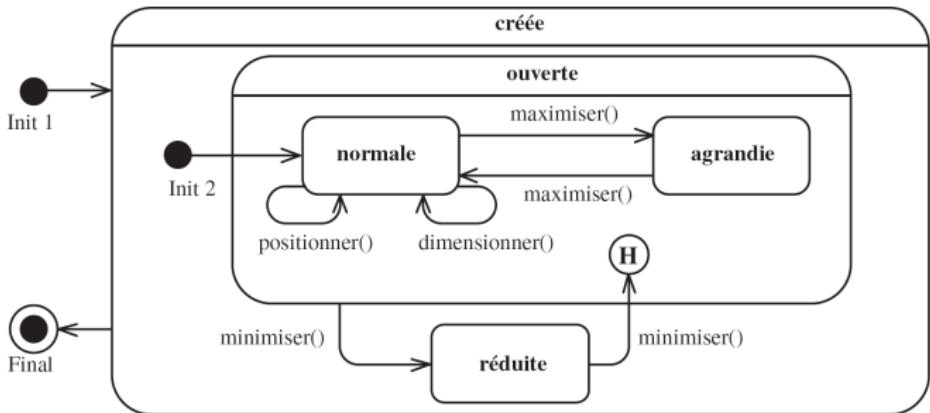
- ▶ L'organisation des états et des transitions pour un classeur donné est représentée dans un **diagramme d'états-transitions**.
- ▶ Le modèle dynamique comprend **plusieurs diagrammes d'états**.

## Attention

Chaque diagramme d'états ne concerne qu'une seule classe.

Chaque automate à états finis s'exécute concurremment et peut changer d'état de façon indépendante des autres.

# Exemple de diagramme d'états-transitions



## Etat initial et état final

- ▶ L'état initial est un pseudo-état qui définit le point de départ par défaut pour l'automate ou le sous-état.
  - ▶ Lorsqu'un objet est créé, il entre dans l'état initial.



- ▶ L'état final est un pseudo-état qui indique que l'exécution de l'automate ou du sous-état est terminée.



## Evénement déclencheur

- ▶ Les transitions d'un diagramme d'états-transitions sont déclenchées par des événements déclencheurs.
  - ▶ Un appel de méthode sur l'objet courant génère un événement de type **call**.
  - ▶ Le passage de faux à vrai de la valeur de vérité d'une condition booléenne génère implicitement un événement de type **change**.
  - ▶ La réception d'un signal asynchrone, explicitement émis par un autre objet, génère un événement de type **signal**.
  - ▶ L'écoulement d'une durée déterminée après un événement donné génère un événement de type **after**. Par défaut, le temps commence à s'écouler dès l'entrée dans l'état courant.

L'événement déclencheur est indiqué à côté de la flèche représentant la transition



## Evénements call et signal

- ▶ Un événement de type **call** ou **signal** est déclaré ainsi :

```
nomEvenement (params)
```

- ▶ Chaque paramètre a la forme :

```
param : ClasseParam
```

- ▶ Les événements de type **call** sont des méthodes déclarées au niveau du diagramme de classes.
- ▶ Les **signaux** sont déclarés par la définition d'une classe portant le stéréotype signal, ne fournissant pas d'opérations, et dont les attributs sont interprétés comme des arguments.

# Événements change et after

- ▶ Un événement de type **change** est introduit de la façon suivante :

```
when(conditionBooleenne)
```

- ▶ Il prend la forme d'un test continu et se déclenche potentiellement à chaque changement de valeurs des variables intervenant dans la condition.
- ▶ Un événement temporel de type **after** est spécifié par :

```
after(duree)
```

- ▶ Le paramètre s'évalue comme une durée, par défaut écoulée depuis l'entrée dans l'état courant.
- ▶ Par exemple : after(10 secondes).

# Transition simple

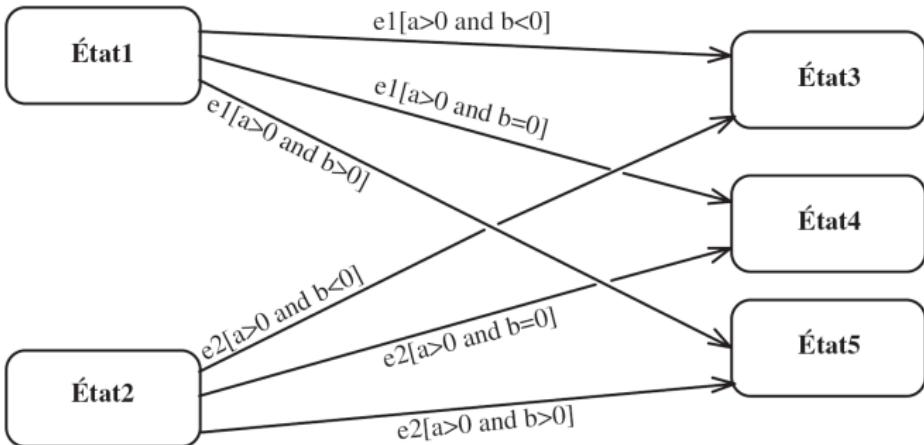
- ▶ Une **transition entre deux états** est représentée par un arc qui les lie l'un à l'autre.
  - ▶ Elle indique qu'une instance peut changer d'état et exécuter certaines activités, si un événement déclencheur se produit et que les conditions de garde sont vérifiées.
- ▶ Sa **syntaxe** est la suivante :

```
nomEvenement(params) [garde] / activite
```

- ▶ La garde désigne une condition qui doit être remplie pour pouvoir déclencher la transition,
- ▶ L'activité désigne des instructions à effectuer au moment du tir.

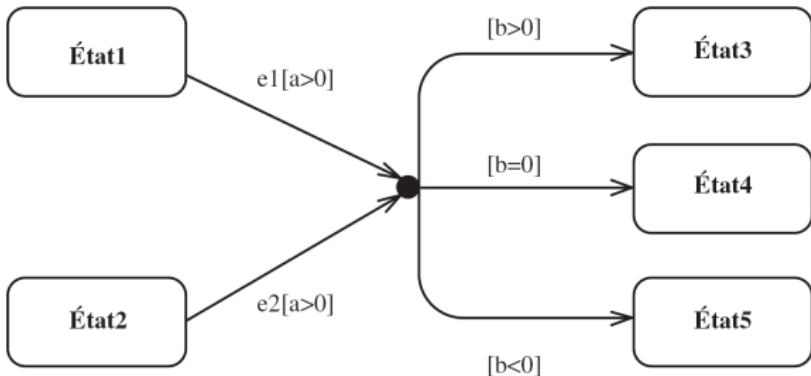
- ▶ On peut représenter des alternatives pour le franchissement d'une transition.
- ▶ On utilise pour cela des pseudo-états particuliers :
  - ▶ Les points de jonction (petit cercle plein) permettent de partager des segments de transition.
    - ▶ Ils ne sont qu'un raccourci d'écriture.
    - ▶ Ils permettent des représentations plus compactes.
  - ▶ Les points de choix (losange) sont plus que des raccourcis d'écriture.

# Simplification avec les points de jonction



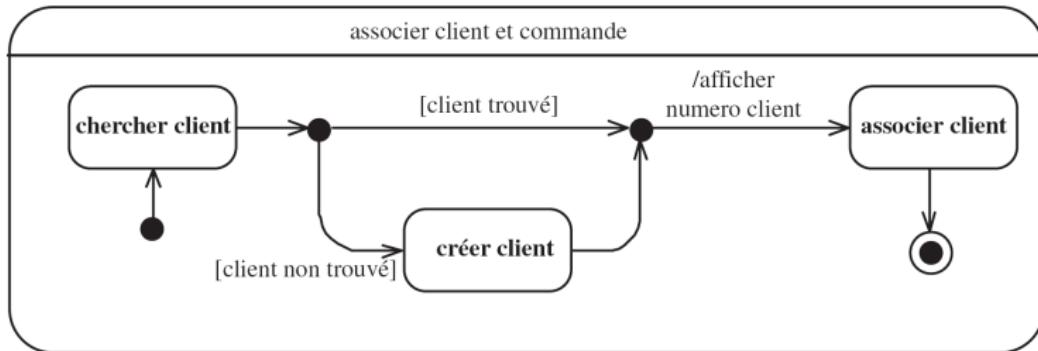
- ▶ Pour emprunter un chemin, toutes les gardes le long de ce chemin doivent s'évaluer à vrai dès le franchissement du premier segment.

# Simplification avec les points de jonction



# Représentation d'alternatives avec les points de jonction

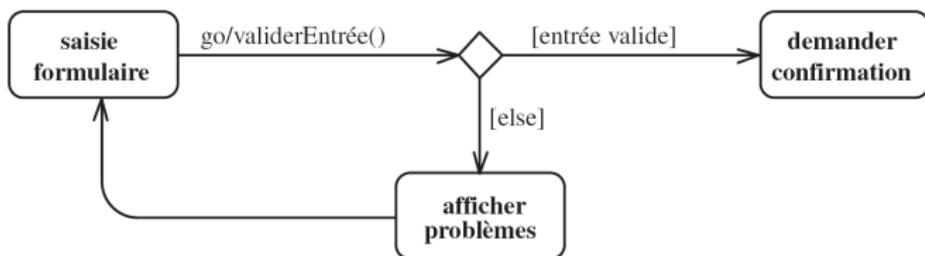
242



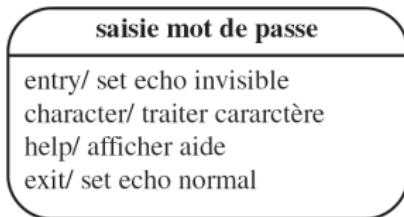
# Utilisation des points de choix

- ▶ Les gardes après le point de choix sont évaluées au moment où il est atteint.
- ▶ Cela permet de baser le choix sur des résultats obtenus en franchissant le segment avant le point de choix.
- ▶ Si, quand le point de choix est atteint, aucun segment en aval n'est franchissable, le modèle est **mal formé**.

Contrairement aux points de jonction, les points de choix ne sont pas de simples racourcis d'écriture.



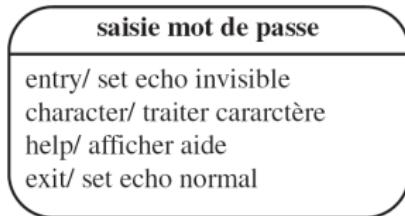
- ▶ Un objet reste dans un état durant une certaine durée et des transitions internes peuvent intervenir.
- ▶ Une **transition interne** ne modifie pas l'état courant, mais suit globalement les règles d'une transition simple entre deux états.
- ▶ Trois déclencheurs particuliers sont introduits permettant le tir de transitions internes : **entry/**, **do/**, et **exit/**.



- ▶ “entry” définit une activité à effectuer à chaque fois que l'on rentre dans l'état considéré.
- ▶ “exit” définit une activité à effectuer quand on quitte l'état.
- ▶ “do” définit une activité continue qui est réalisée tant que l'on se trouve dans l'état, ou jusqu'à ce que le calcul associé soit terminé.
  - ▶ On pourra dans ce dernier cas gérer l'événement correspondant à la fin de cette activité (completion event).

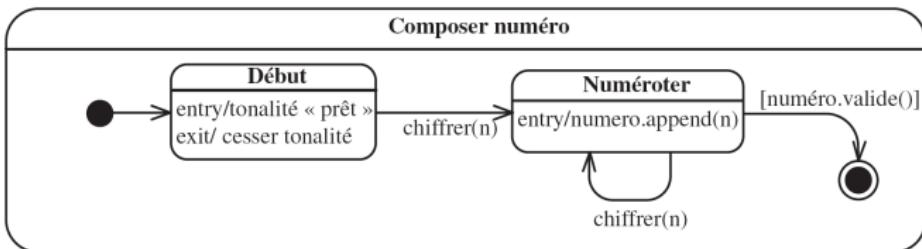
- ▶ Les **transitions internes** sont spécifiées dans le compartiment inférieur de l'état, sous le compartiment du nom.
- ▶ Chaque transition interne est décrite selon la syntaxe suivante :

```
nomEvenement(params) [garde] / activiteARE
```



# Etat composite

- ▶ Un état composite, par opposition à un état dit “simple”, est décomposé en deux ou plusieurs sous-états.
  - ▶ Tout état ou sous-état peut ainsi être décomposé en sous-états imbriqués sans limite a priori de profondeur.
  - ▶ Un état composite est représenté par les deux compartiments de nom et d’actions internes habituelles, et par un compartiment contenant le sous-diagramme.



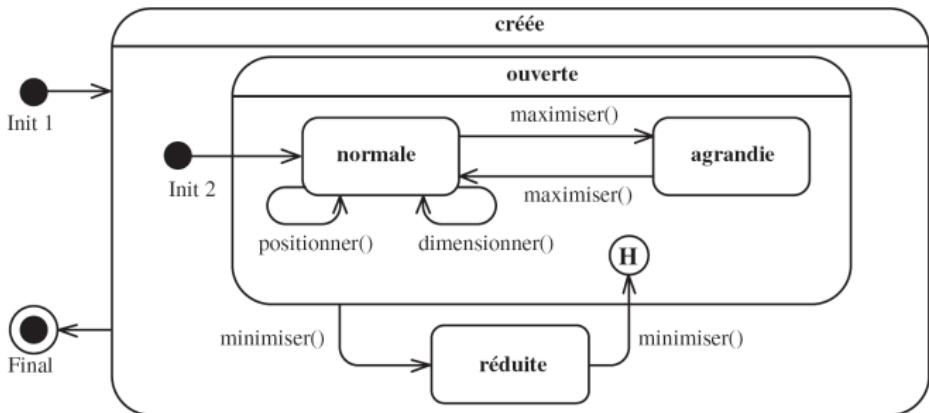
# Etat composite

- ▶ Un état **composite**, par opposition à un état dit “simple”, est décomposé en deux ou plusieurs sous-états.
  - ▶ Tout état ou sous-état peut ainsi être décomposé en sous-états imbriqués sans limite a priori de profondeur.
  - ▶ Un état composite est représenté par les deux compartiments de nom et d’actions internes habituelles, et par un compartiment contenant le sous-diagramme.

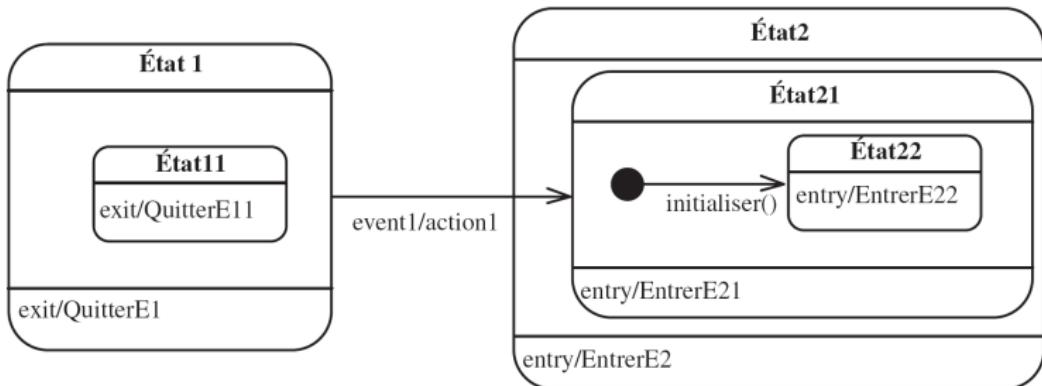


# Etats composites et états initiaux/finaux

- ▶ Les transitions peuvent avoir pour cible la frontière d'un état composite. Elle sont alors équivalentes à une transition ayant pour cible l'état initial de l'état composite.
- ▶ Une transition ayant pour source la frontière d'un état composite est équivalente à une transition qui s'applique à tout sous-état de l'état composite source.
  - ▶ Cette relation est transitive et peut "traverser" plusieurs niveaux d'imbrication.

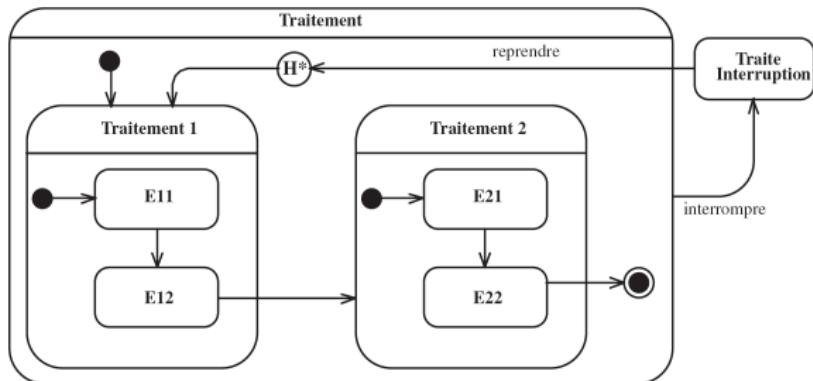


- ▶ Depuis Etat11, quand **event** survient
  - ▶ On produit la séquence d'activités QuitterE11, QuitterE1, action1, EntrerE2, Entrer21, initialiser, Entrer22
  - ▶ L'objet se trouve alors est dans Etat22.



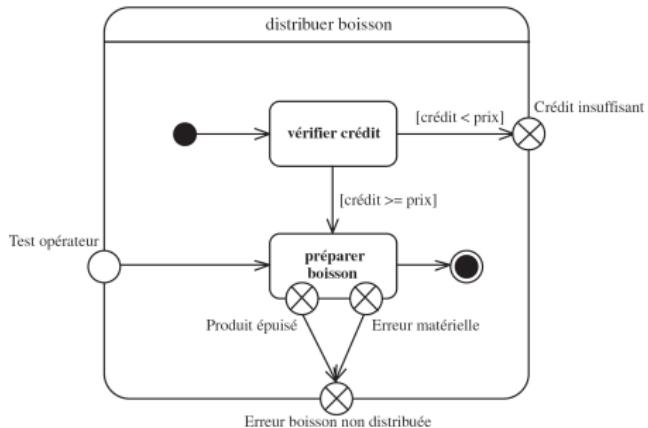
# Historique

- ▶ Un **pseudo-état historique** est noté par un H cerclé.
- ▶ Une transition ayant pour cible le pseudo-état historique est équivalente à une transition qui a pour cible le dernier état visité dans la région contenant le H.
- ▶ H\* désigne un **historique profond**, cad un historique valable pour tous les niveaux d'imbrication.



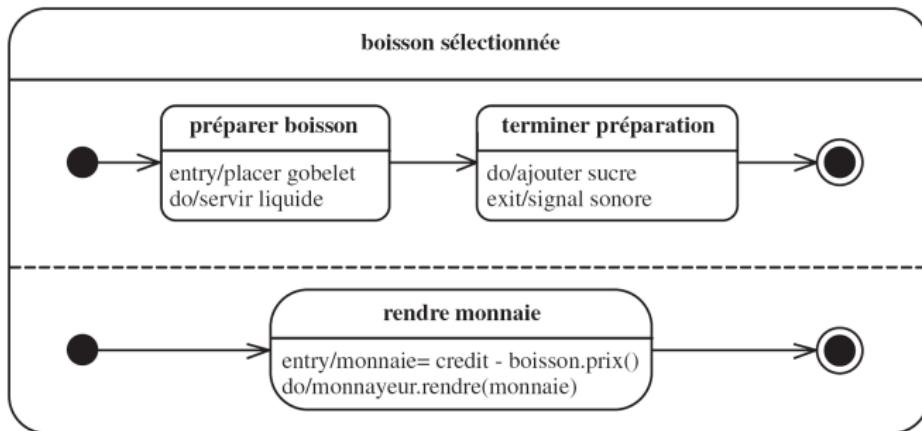
# Interface des états composites

- ▶ Pour pouvoir représenter un sous état indépendamment d'un macro-état, on a recours à des **points de connexion**.
  - ▶ Avec un X pour les points de sortie
  - ▶ Vides pour les points d'entrée
- ▶ Ces interfaces permettent d'abstraire les sous-états des macro-états (réutilisabilité).



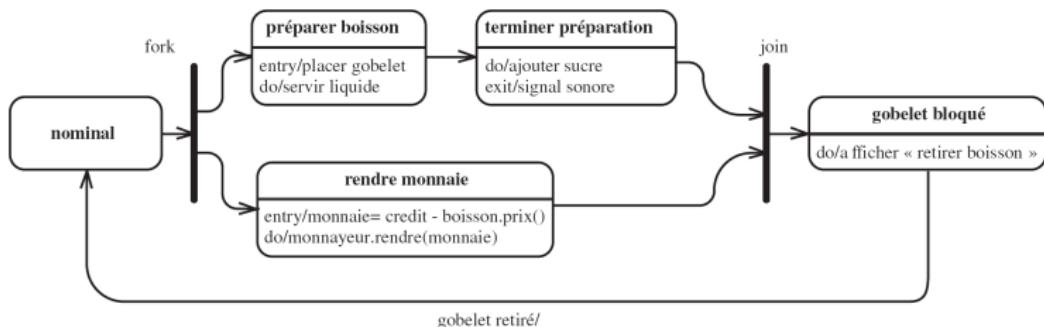
## Etat concurrent

- ▶ Avec un séparateur en pointillés, on peut représenter plusieurs automates s'exécutant indépendamment.
- ▶ Un objet peut alors être simultanément dans plusieurs états concurrents.



# Transition concurrente

- ▶ Une transition **fork** correspond à la création de deux états concurrentes.
- ▶ Une transition **join** correspond à une barrière de synchronisation qui supprime la concurrence.
  - ▶ Pour pouvoir continuer leur exécution, toutes les tâches concurrentes doivent préalablement être prêtes à franchir la transition.



## Conception objet élémentaire avec UML

### UML et méthodologie

## Modélisation avancée avec UML

Expression de contraintes avec OCL

Diagrammes d'états-transitions

Diagrammes d'activités

Diagrammes de communication

Diagrammes de composants et de déploiement

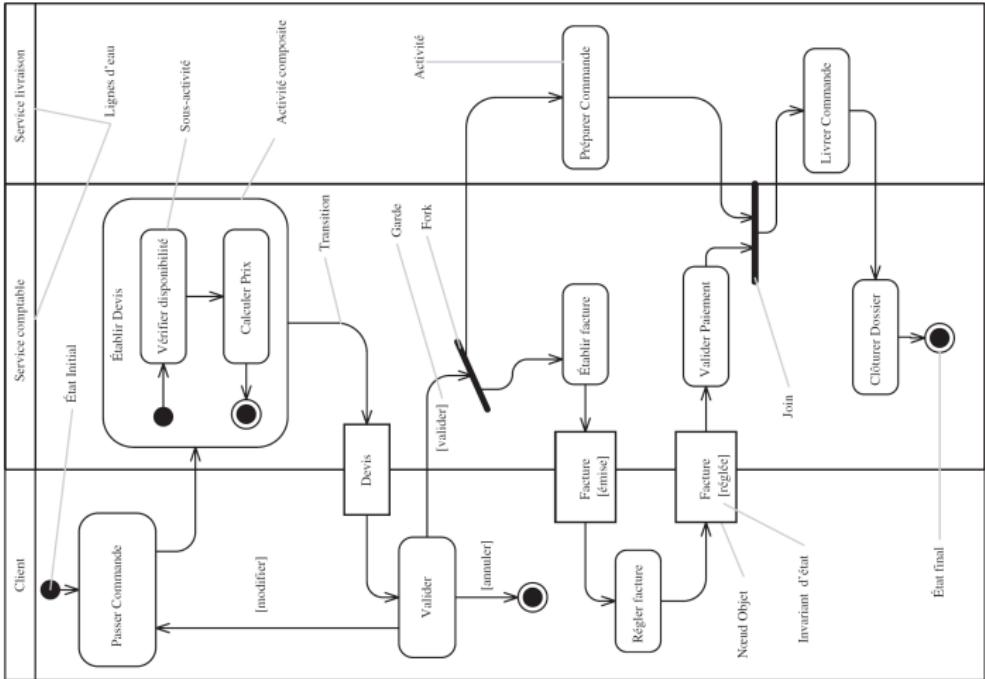
## Bonnes pratiques de la modélisation objet

# Modélisation des traitements

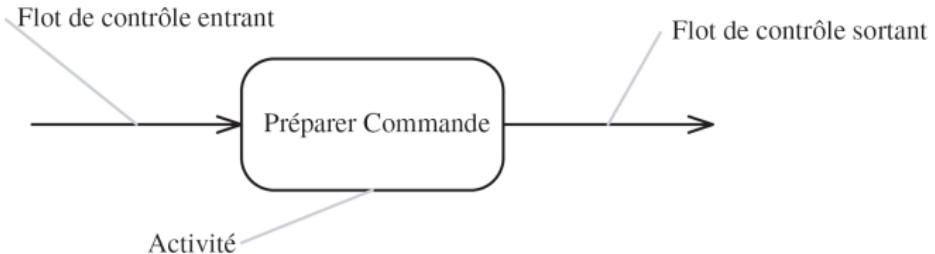
- ▶ Les **diagrammes d'activité** offrent une manière graphique et non ambiguë pour modéliser les traitements.
  - ▶ Comportement d'une méthode
  - ▶ Déroulement d'un cas d'utilisation
- ▶ Une **activité** représente une exécution d'un mécanisme, un déroulement d'étapes séquentielles. Le passage d'une activité à l'autre autre est matérialisé par une **transition**.
- ▶ Ces diagrammes sont assez semblables aux états-transitions mais avec une interprétation différente.

- ▶ Les diagrammes d'**états-transitions** sont définis pour chaque classeur et n'en font pas intervenir plusieurs.
- ▶ A l'inverse, les **diagrammes d'activité** permettent une description s'affranchissant (partiellement) de la structuration de l'application en classeurs.
- ▶ La vision des diagrammes d'activités se rapproche des langages de programmation impérative (C, C++, Java)
  - ▶ Les états représentent des calculs
  - ▶ Il n'y a pas d'événements externes mais des attentes de fins de calculs
  - ▶ Il peut y avoir de la concurrence entre activités

# Exemple de diagramme d'activités



- ▶ Les activités décrivent un traitement.
  - ▶ Le flux de contrôle reste dans l'activité jusqu'à ce que les traitements soient terminés.
  - ▶ On peut définir des variables locales à une activité et manipuler les variables accessibles depuis le contexte de l'activité (classe contenante en particulier).
  - ▶ Les activités peuvent être imbriquées hiérarchiquement : on parle alors d'activités composites.

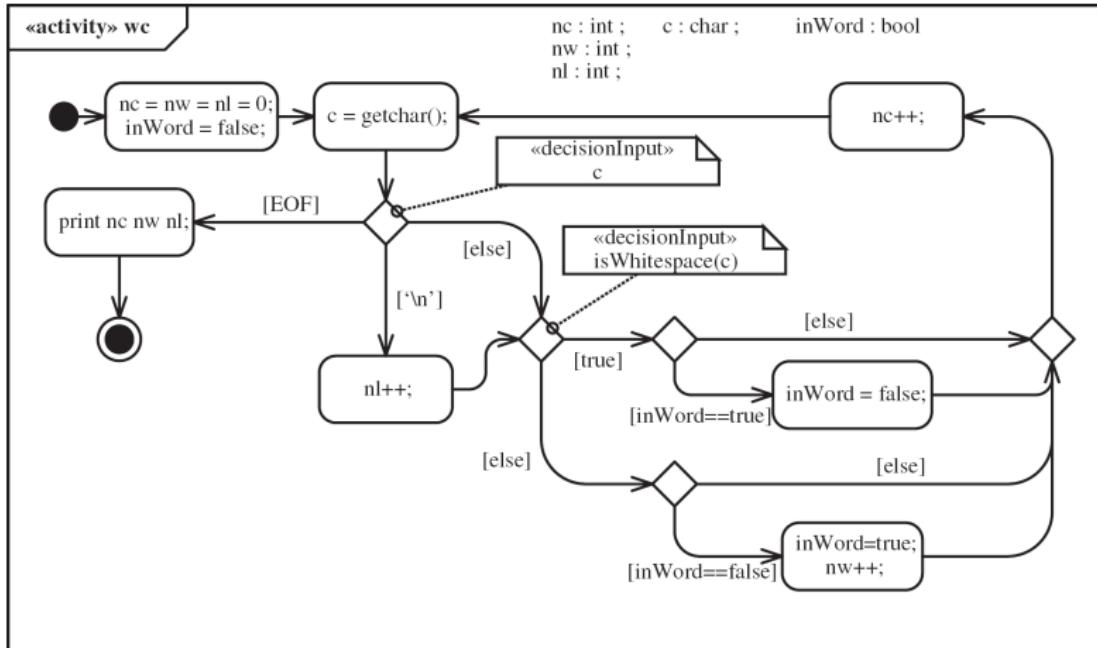


- ▶ Les **transitions** sont représentées par des flèches pleines qui connectent les activités entre elles.
  - ▶ Elles sont déclenchées dès que l'activité source est terminée.
  - ▶ Elles provoquent automatiquement le début immédiat de la prochaine activité à déclencher (l'activité cible).
  - ▶ Contrairement aux activités, les transitions sont franchies de manière atomique, en principe sans durée perceptible.

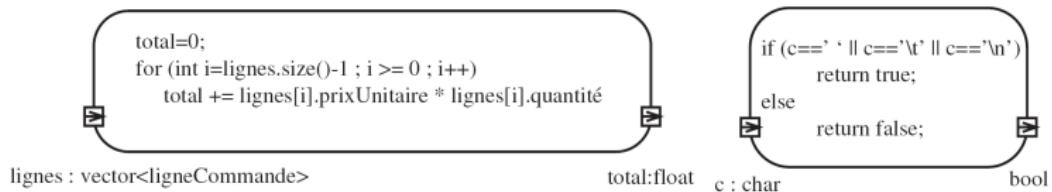
Les transitions spécifient l'enchaînement des traitements et définissent le flux de contrôle.

- ▶ Expression de conditions au moyen de transitions munies de **gardes conditionnelles**
  - ▶ Ces transitions ne peuvent être empruntées que si la garde est vraie.
  - ▶ On dispose d'une clause [**else**] qui est validée si et seulement si toutes les autres gardes des transitions ayant la même source sont fausses.
- ▶ Les conditions sont notées entre crochets
- ▶ Pour mieux mettre en évidence un branchement conditionnel, on peut utiliser les points de choix (losanges).
  - ▶ Les points de choix montrent un aiguillage du flot de contrôle.

# Exemples de structures conditionnelles



- ▶ Les activités structurées utilisent les structures de contrôle usuelles (conditionnelles et boucle) à travers une syntaxe qui dépend de l'outil.
- ▶ La syntaxe précise de ces annotations pas définie dans la norme UML.
- ▶ Dans une activité structurée, on définit les arguments d'entrée et les sorties par des flèches encadrées.

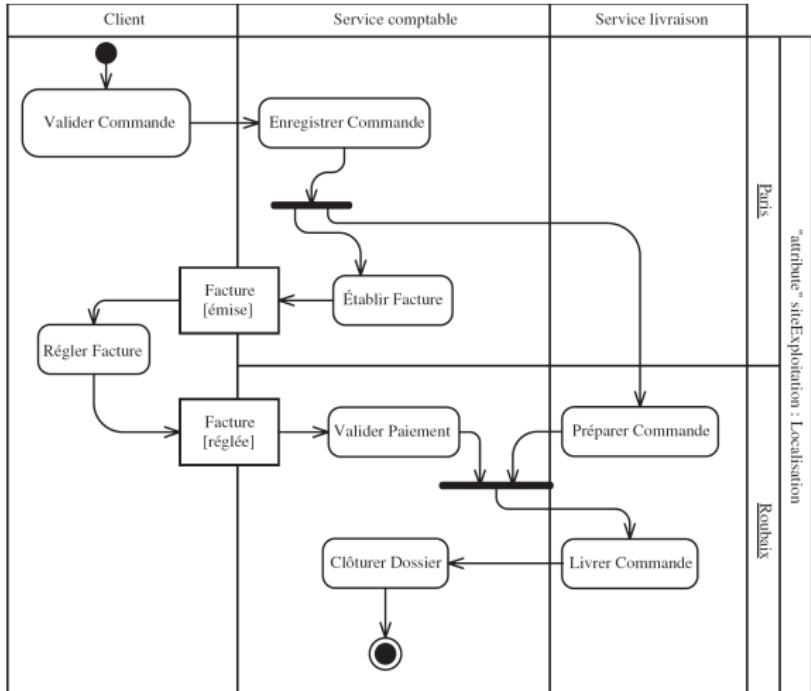


- ▶ Pour modéliser un traitement mettant en oeuvre plusieurs classeurs, on peut spécifier le classeur responsable de chaque activité.
- ▶ Les **partitions** permettent d'attribuer les activités à des éléments particuliers du modèle.
  - ▶ Une partition peut elle-même être décomposée en sous-partitions.
- ▶ Pour spécifier qu'une activité est effectuée par un classeur particulier, on la positionne dans la partition correspondante.

| «external» | «attribute» libelléService : Service |                          |
|------------|--------------------------------------|--------------------------|
| Client     | <u>Service comptable</u>             | <u>Service livraison</u> |
|            |                                      |                          |



# Partitions multidimensionnelles



# Partitions explicites

- ▶ Cette notation est moins encombrante graphiquement
- ▶ Toutefois, elle met moins bien en valeur l'appartenance de groupes d'activités à un même conteneur.

«external»  
(Client)  
Régler facture

(Service Comptable)  
Etablir facture

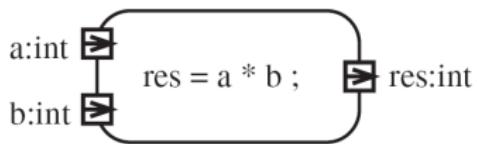
(Service Livraison)  
Livrer Commande

## Arguments et valeurs retournées

- ▶ Les diagrammes d'activités présentés jusqu'ici montrent bien le comportement du flot de contrôle.
- ▶ Pourtant, le flot de données n'apparaît pas.
  - ▶ Si une activité est bien adaptée à la description d'une opération d'un classeur, il faut un moyen de spécifier les arguments et valeurs de retour de l'opération. C'est le rôle
    - ▶ des **pins**,
    - ▶ des **noeuds**,
    - ▶ des **flots d'objets** associés.

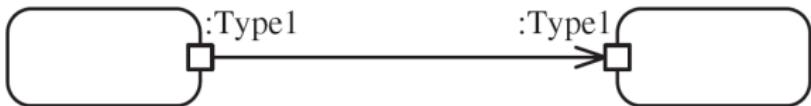
# Pin

- ▶ Un **pin** représente un point de connexion pour une action.
- ▶ L'action ne peut débuter que si l'on affecte une valeur à chacun de ses pins d'entrée.
- ▶ Les valeurs sont passées par copie.
- ▶ Quand l'activité se termine, une valeur doit être affectée à chacun des pins de sortie

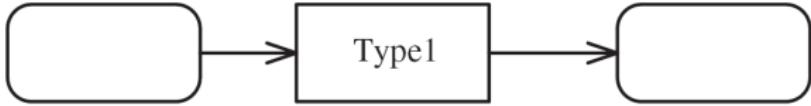


# Flot de données

- ▶ Un **flot d'objets** permet de passer des données d'une activité à une autre.
  - ▶ De fait, un arc qui a pour origine et destination un pin correspond à un flot de données.

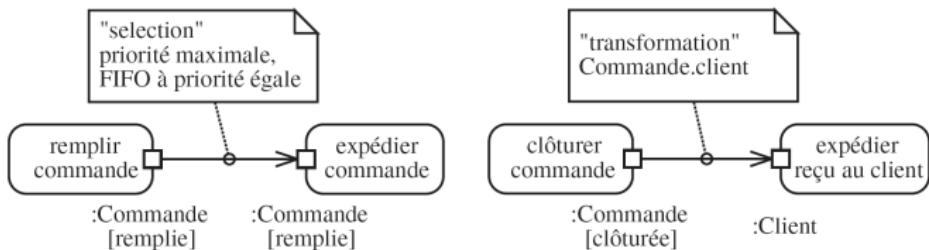


- ▶ Un **noeud d'objets** permettent de mieux mettre en valeur les données.
  - ▶ C'est un conteneur typé qui permet le transit des données.

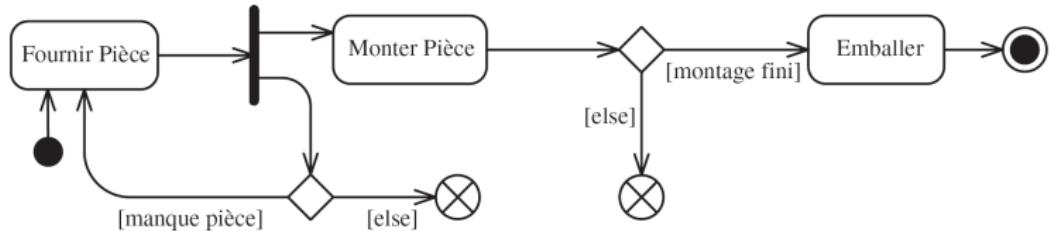


# Annotation des flots de données

- ▶ Un flot d'objets peut porter une étiquette mentionnant deux annotations particulières :
  - ▶ “**transformation**” indique une interprétation particulière de la donnée transmise par le flot.
  - ▶ “**selection**” indique l'ordre dans lequel les objets sont choisis dans le noeud pour le quitter.



- ▶ Les diagrammes d'activités permettent en principe de représenter des activités séquentielles.
- ▶ Néanmoins, on peut représenter des **activités concurrentes** avec :
  - ▶ Les barres de synchronisation,
  - ▶ Les noeuds de contrôle de type "flow final".



## Barre de synchronisation

- ▶ Plusieurs transitions peuvent avoir pour source ou pour cible une **barre de synchronisation**.
  - ▶ Lorsque la barre de synchronisation a plusieurs transitions en **sortie**, on parle de transition de type **fork** qui correspond à une **duplication du flot de contrôle** en plusieurs flots indépendants.
  - ▶ Lorsque la barre de synchronisation a plusieurs transitions en **entrée**, on parle de transition de type **join** qui correspond à un **rendez-vous** entre des flots de contrôle.
- ▶ Pour plus de commodité, il est possible de fusionner des barres de synchronisation de type **join** et **fork**.
  - ▶ On a alors plusieurs transitions entrantes et sortantes sur une même barre.



## Noeud de contrôle de type “flow final”

- ▶ Un flot de contrôle qui atteint un noeud de contrôle de type “flow final” est détruit.
  - ▶ Les autres flots de contrôle ne sont pas affectés.

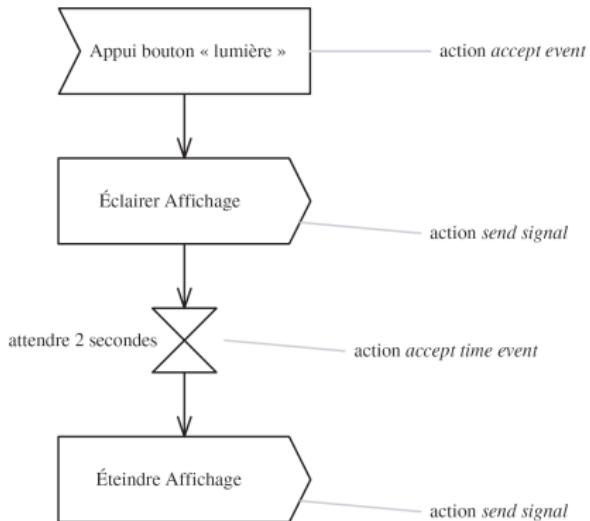


- ▶ Ce type de noeud est moins “fort” qu'un noeud de contrôle final.
  - ▶ Dans ce cas, tous les autres flots de contrôle de l'activité sont interrompus et détruits.



# Actions de communication

- ▶ **Les actions de communication gèrent**
  - ▶ le passage et le retour de paramètres,
  - ▶ les mécanismes d'appels d'opérations synchrones et asynchrones.
  
- ▶ **Les actions de communication peuvent être employés comme des activités dans les diagrammes d'activité.**



# Actions de communication

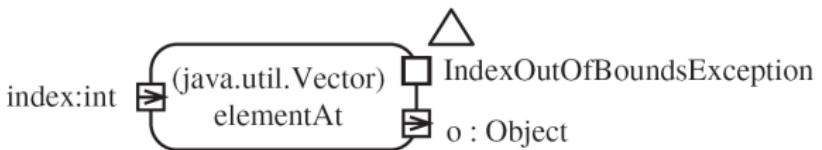
- ▶ Les actions de type **call** correspondent à des appels de procédure ou de méthode.
  - ▶ **Call operation** correspond à l'appel d'une opération sur un classeur.
  - ▶ **Call behavior** correspond à l'invocation d'un comportement spécifié à l'aide d'un diagramme UML
  - ▶ Dans les deux cas, il est possible de spécifier des arguments et d'obtenir des valeurs en retour.
- ▶ Les actions **accept call** et **reply** peuvent être utilisées du côté récepteur pour décomposer la réception de l'appel.

# Appel asynchrone

- ▶ Les appels asynchrones de type **send** correspondent à des envois de messages asynchrones.
- ▶ Le **broadcast signal** permet d'émettre vers plusieurs destinataires à la fois
  - ▶ Cette possibilité est rarement offerte par les langages de programmation.
- ▶ L'action symétrique côté récepteur est **accept event**, qui permet la réception d'un signal.

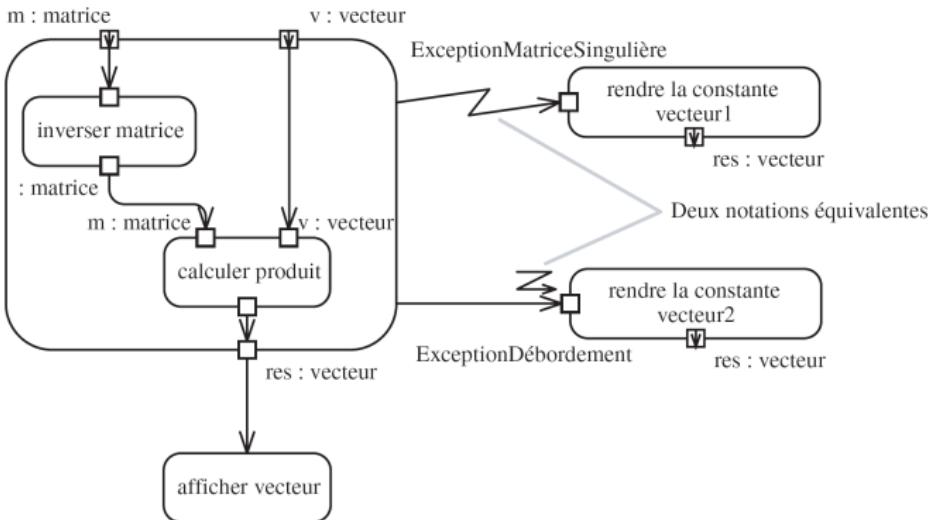
# Exceptions

- ▶ Les exceptions permettent d'interrompre un traitement quand une situation qui dévie du traitement normal se produit. Elles assurent une gestion plus propre des erreurs qui peuvent se produire au cours d'un traitement.
- ▶ On utilise des pins d'exception (avec un triangle) un pour gérer l'envoi d'exceptions et la capture d'exceptions.



- ▶ Un flot de données correspondant à une exception est matérialisé par une flèche en “zigue zague”.

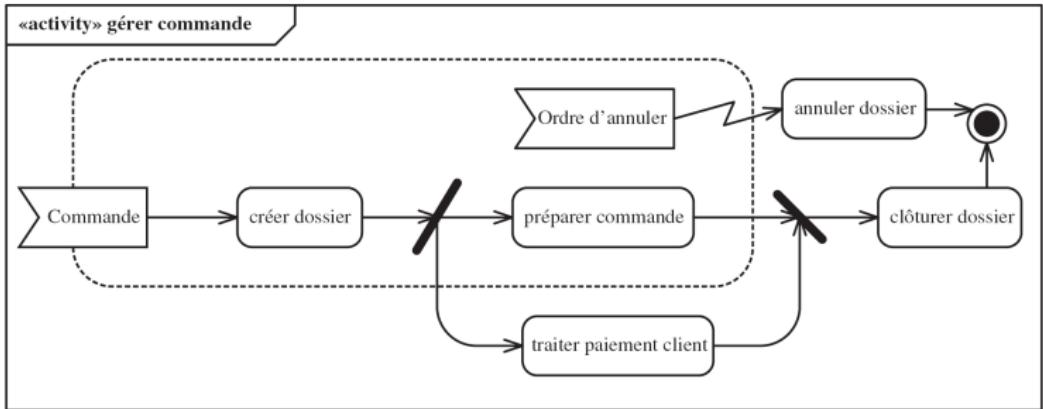
# Exemple de traitement d'exceptions



# Régions interruptibles

- ▶ Ce mécanisme est analogue à celui des **interruptions**, mais il est moins détaillé.
  - ▶ Les **régions interruptibles** sont mieux adaptées aux phases de modélisation fonctionnelles.
- ▶ Une région interruptible est représentée par un cadre arrondi en pointillés.
  - ▶ Si l'événement d'interruption se produit, toutes les activités en cours dans la région interruptible sont stoppées
  - ▶ Le flot de contrôle suit la flèche en zigzag qui quitte la région.

# Exemple de région interruptible



# Types de zones d'expansion

## ► Parallel

- Les exécutions de l'activité interne sur les éléments de la collection sont indépendantes et peuvent être réalisées dans n'importe quel ordre ou même simultanément.

## ► Iterative

- Les occurrences d'exécution de l'activité interne doivent s'enchaîner séquentiellement, en suivant l'ordre de la collection d'entrée.

## ► Stream

- Les éléments de la collection sont passés sous la forme d'un flux de données à l'activité interne, qui doit être adaptée aux traitements de flux.

# Plan

Conception objet élémentaire avec UML

UML et méthodologie

Modélisation avancée avec UML

Expression de contraintes avec OCL

Diagrammes d'états-transitions

Diagrammes d'activités

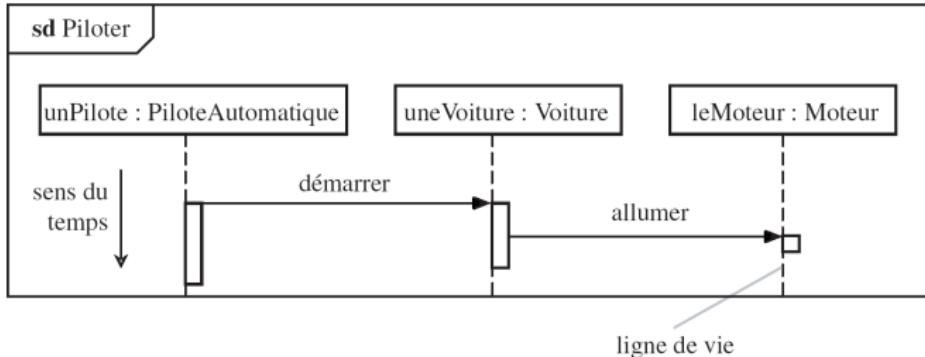
Diagrammes de communication

Diagrammes de composants et de déploiement

Bonnes pratiques de la modélisation objet



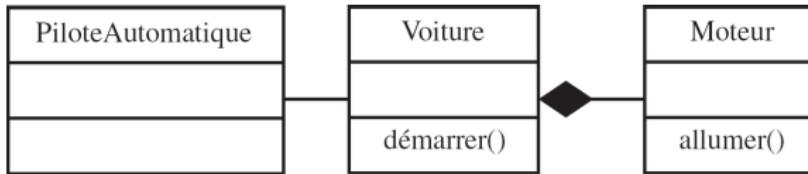
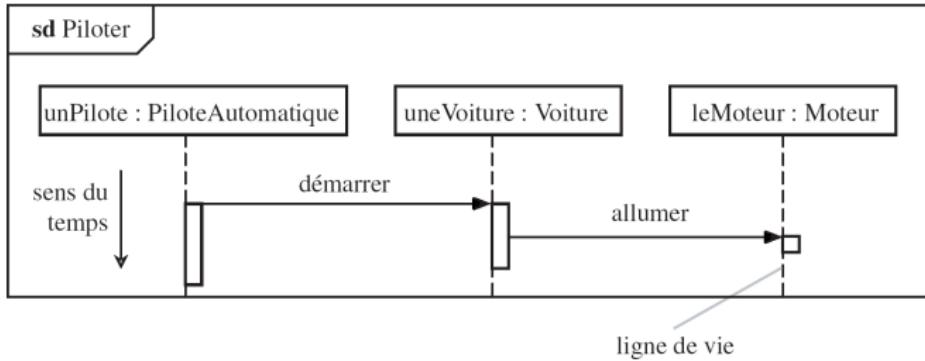
# Diagrammes de séquence



- ▶ Les diagrammes de séquence permettent de modéliser l'échange d'informations entre différents classeurs
- ▶ Ils sont un type de diagrammes d'**interaction**

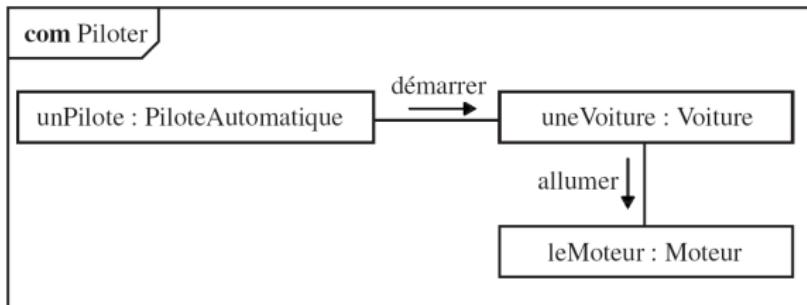
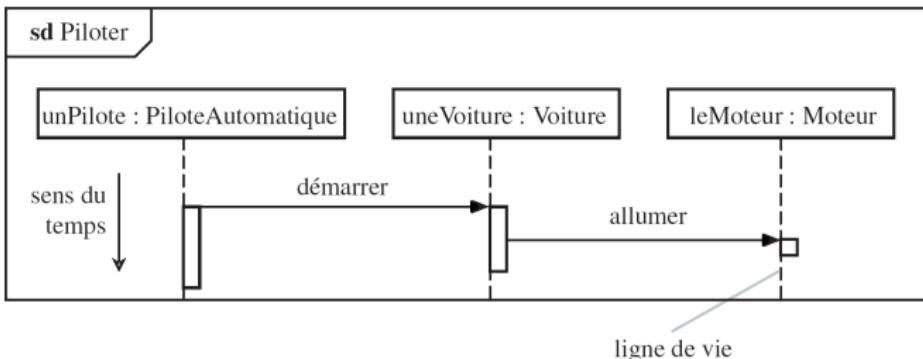
# Diagrammes d'interaction

- ▶ Les diagrammes de communication et les diagrammes de séquences sont deux types de diagramme d'**interaction**
  - ▶ Un **diagramme de séquence** montre des interactions sous un angle temporel, en mettant l'emphase sur le séquencement temporel de messages échangés entre des lignes de vie
  - ▶ Un **diagramme de communication** montre une représentation spatiale des lignes de vie.
  - ▶ Ils représentent la même chose, mais sous des formes différentes.
- ▶ A ces diagrammes, UML 2.0 en ajoute un troisième : le **diagramme de timing**.
  - ▶ Son usage est limité à la modélisation des systèmes qui s'exécutent sous de fortes contraintes de temps, comme les systèmes temps réel.



# Diagrammes de séquence et de communication

283



## Rôles et connecteurs

- ▶ Le rôle permet de définir le contexte d'utilisation de l'interaction.

Une rôle dans un diagramme de communication correspond à une ligne de vie dans un diagramme de séquences.

- ▶ Les relations entre les lignes de vie sont appelées "connecteurs".
  - ▶ Un connecteur se représente de la même façon qu'une association mais la sémantique est plus large : un connecteur est souvent une association transitoire.
- ▶ Comme pour les diagrammes de séquence, la syntaxe d'une ligne de vie est :

```
<nomRole>[<selecteur>]:<nomClasseur>
```



# Types de messages

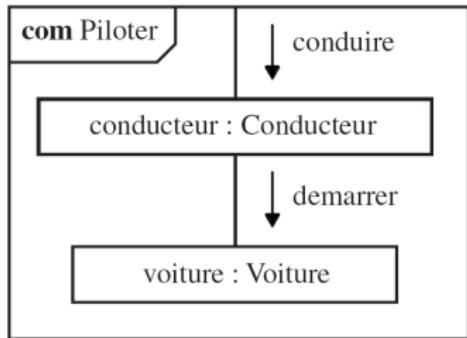
- ▶ Comme dans les diagrammes de séquence, on distingue deux types de messages :
  - ▶ Un message **synchrone** bloque l'expéditeur jusqu'à la réponse du destinataire. Le flux de contrôle passe de l'émetteur au récepteur.  
→
  - ▶ Un message **asynchrone** n'est pas bloquant pour l'expéditeur. Le message envoyé peut être pris en compte par le récepteur à tout moment ou ignoré.  
→

# Représentation des messages

- ▶ Les flèches représentant les messages sont tracées à côté des connecteurs qui les supportent.

## Attention

Bien faire la distinction entre les messages et les connecteurs : on pourrait avoir un connecteur sans message, mais jamais l'inverse.



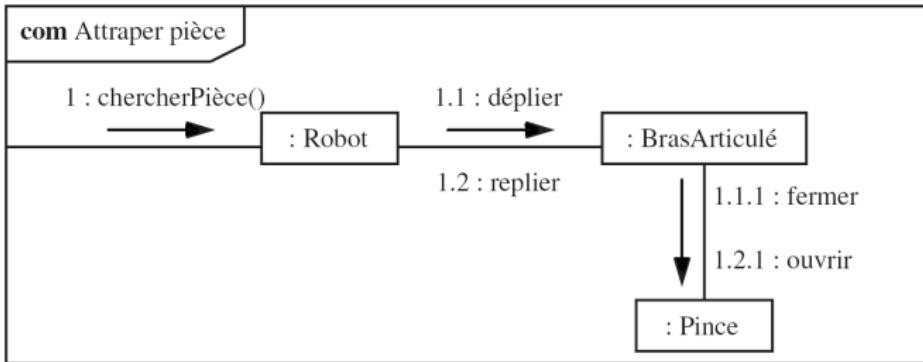
```
public class Conducteur{  
    private Voiture voiture;  
    public void addVoiture( Voiture voiture ){  
        if( voiture != null ){  
            this.voiture = voiture;  
        }  
    }  
    public void conduire(){  
        voiture.demarrer();  
    }  
    public static void main( String [] argv ){  
        conducteur.conduire();  
    }  
}  
public class Voiture{  
    public void demarrer(){...}  
}
```



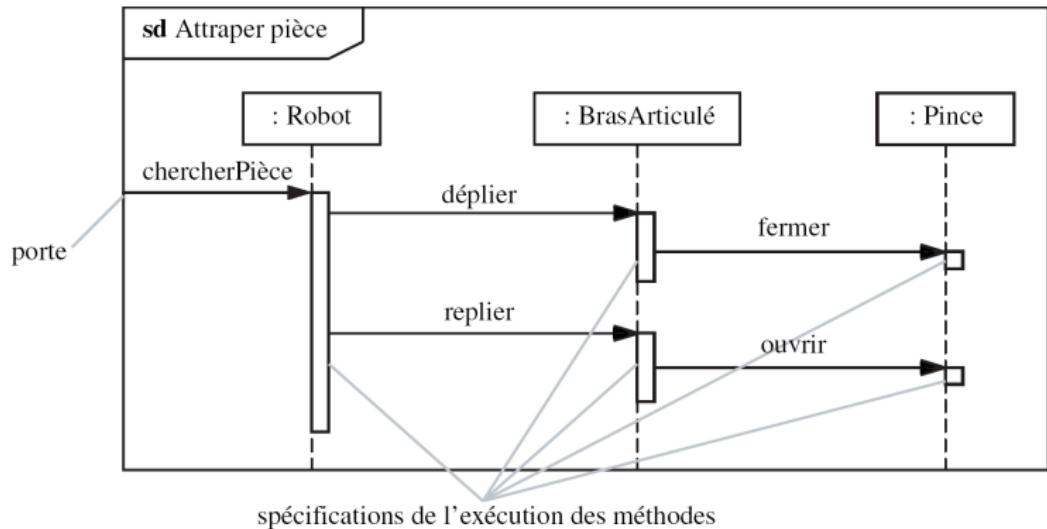
# Numéros de séquence des messages

- ▶ Pour représenter les aspects temporels, on adjoint des **numéros de séquence** aux messages.
  - ▶ Des messages successifs sont ordonnés selon un numéro de séquence croissant (1, 2, 3, ... ou encore 3.1, 3.2, 3.3, ...).
  - ▶ Des messages envoyés en cascade (ex : appel de méthode à l'intérieur d'une méthode) portent un numéro d'emboîtement avec une notation pointée
    - ▶ 1.1, 1.2, ... pour des messages appelés par une méthode dont l'appel portait le numéro 1
    - ▶ 2.a.1, 2.a.2, ... pour des messages appelés par une méthode dont l'appel portait le numéro 2.a

# Équivalence avec un diagramme de séquence

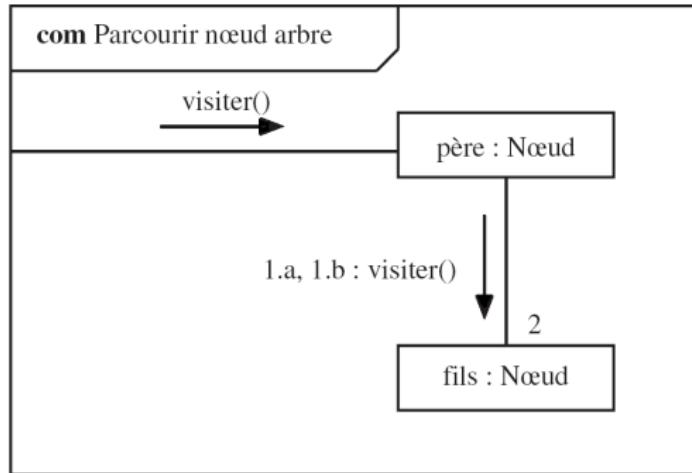


# Équivalence avec un diagramme de séquence



# Messages simultanés

- ▶ Lorsque des messages sont envoyés en **parallèle**, on les numérote avec des lettres
  - ▶ 1.a, 1.b,... pour des messages simultanés envoyés en réponse à un message dont l'envoi portait le numéro 1



- ▶ **Pas d'opérateurs combinés dans les diagrammes de communication**
  - ▶ \*[<clauseItération>] représente une itération.
    - ▶ La clause d'itération peut être exprimée dans le format  $i:=1..n$
    - ▶ Si c'est une condition booléenne, celle-ci représente la condition d'arrêt
  - ▶ \*[<clauseItération>] représente un choix. La clause de condition est une condition booléenne.

# Syntaxe des messages

- ▶ Syntaxe complète des messages est :
- ▶ numéroSéquence [expression] : message
  - ▶ "message" a la même forme que dans les diagrammes de séquence ;
  - ▶ "numéroSéquence" représente le numéro de séquencement des messages ;
  - ▶ "expression" précise une itération ou un embranchement.
- ▶ Exemples :
  - ▶ 2 : affiche(x,y) : message simple.
  - ▶ 1.3.1 : trouve("Haddock") : appel emboîté.
  - ▶ 4 [x < 0] : inverse(x,couleur) : conditionnel.
  - ▶ 3.1 \*[i:=1..10] : recommencer() : itération.



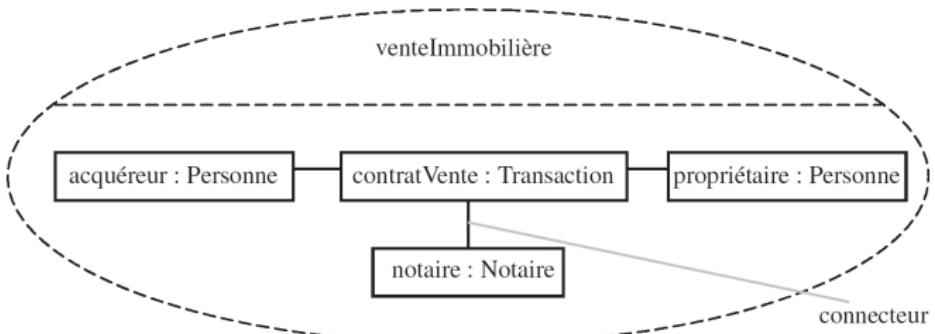
# Collaboration

- ▶ Une **collaboration** montre des instances qui collaborent dans un contexte donné pour mettre en oeuvre une fonctionnalité d'un système.
- ▶ Une collaboration est représentée par une ellipse en traits pointillés comprenant deux compartiments.
  - ▶ Le compartiment supérieur contient le nom de la collaboration ayant pour syntaxe :

`<nomRole>:<nomType>[<multiplicite>]`

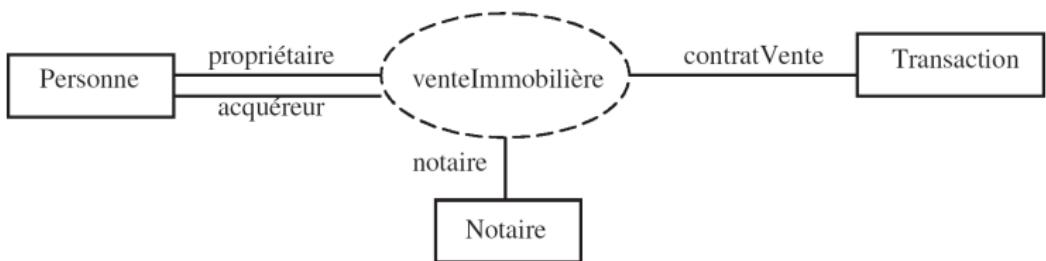
- ▶ Le compartiment inférieur montre les participants à la collaboration.

# Exemple de collaboration



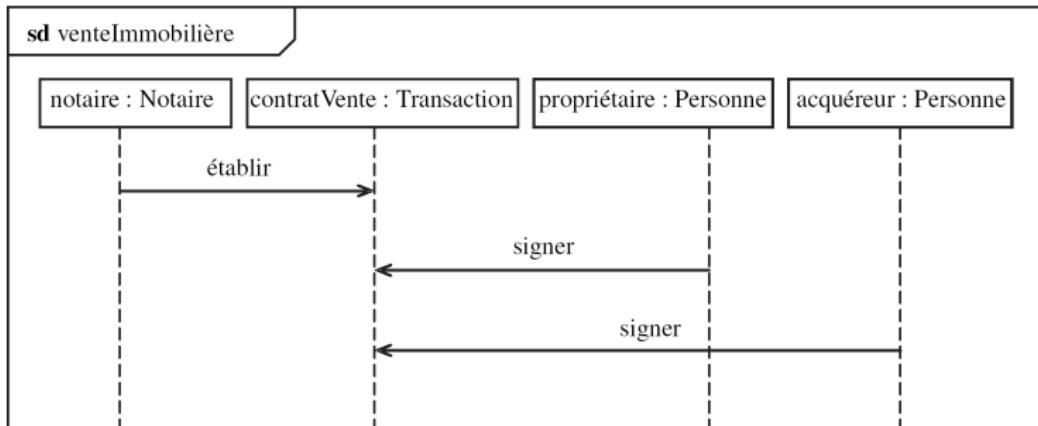
# Diagramme de collaboration

- ▶ Une collaboration peut aussi se représenter par une ellipse sans compartiment, portant le nom de la collaboration en son sein. Les instances sont reliées à l'ellipse par des lignes qui portent le nom du rôle de chaque instance.
- ▶ On peut ainsi former des **diagrammes de collaborations**.



# Collaborations et interactions

- ▶ Les **collaborations** donnent lieu à des **interactions**
  - ▶ Les interactions documentent les collaborations
  - ▶ Les collaborations organisent les interactions.
- ▶ Les interactions se représentent indifféremment par des diagrammes de **communication** ou de **séquence**



## Conception objet élémentaire avec UML

### UML et méthodologie

## Modélisation avancée avec UML

Expression de contraintes avec OCL

Diagrammes d'états-transitions

Diagrammes d'activités

Diagrammes de communication

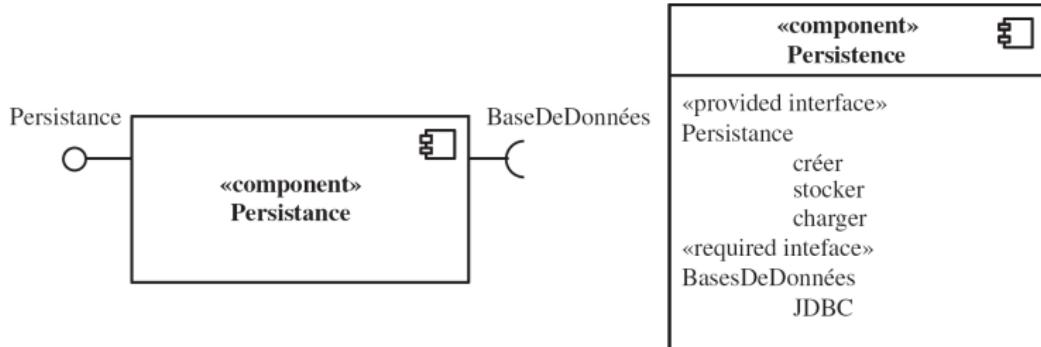
**Diagrammes de composants et de déploiement**

## Bonnes pratiques de la modélisation objet



# Composant

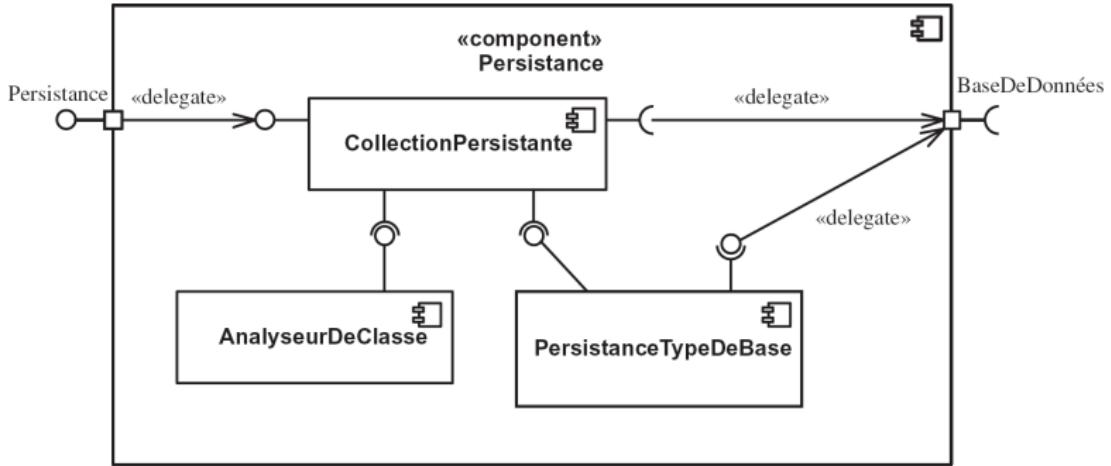
- ▶ Un **composant logiciel** est une unité logicielle autonome au sein d'un système global ou d'un sous-système.
- ▶ C'est un classeur structuré particulier, muni d'une ou plusieurs interfaces requises ou offertes.



# Diagramme de composant

- ▶ Les **diagrammes de composants** représentent l'architecture logicielle du système.
  - ▶ Les composants peuvent être décomposés en **sous-composants**, le cas échéant.
  - ▶ Les liens entre composants sont spécifiés à l'aide de **dépendances entre leurs interfaces**.
    - ▶ Le "câblage interne" d'un composant est spécifié par les **connecteurs de délégation**. Un tel connecteur connecte un port externe du composant avec un port de l'un de ses sous-composants internes.

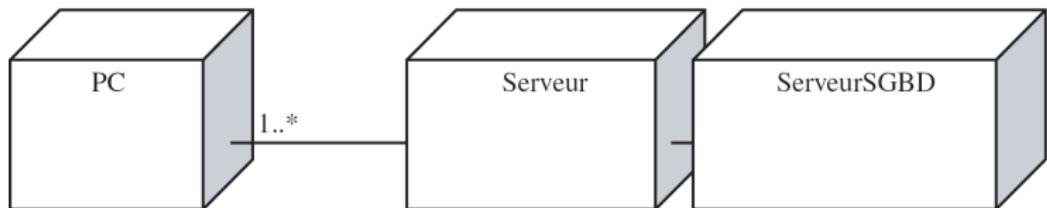
# Exemple de modélisation d'un composant



- ▶ Les diagrammes de composants permettent de
  - ▶ Structurer une architecture logicielle à un niveau de granularité moindre que les classes.
    - ▶ Les composants peuvent contenir des classes.
  - ▶ Spécifier l'intégration de briques logicielles tierces (composants EJB, CORBA, COM+ ou .Net, WSDL...) ;
  - ▶ Identifier les composants réutilisables.
- ▶ Un composant est un espace de noms qu'on peut employer pour organiser les éléments de conception, les cas d'utilisation, les interactions et les artefacts de code.

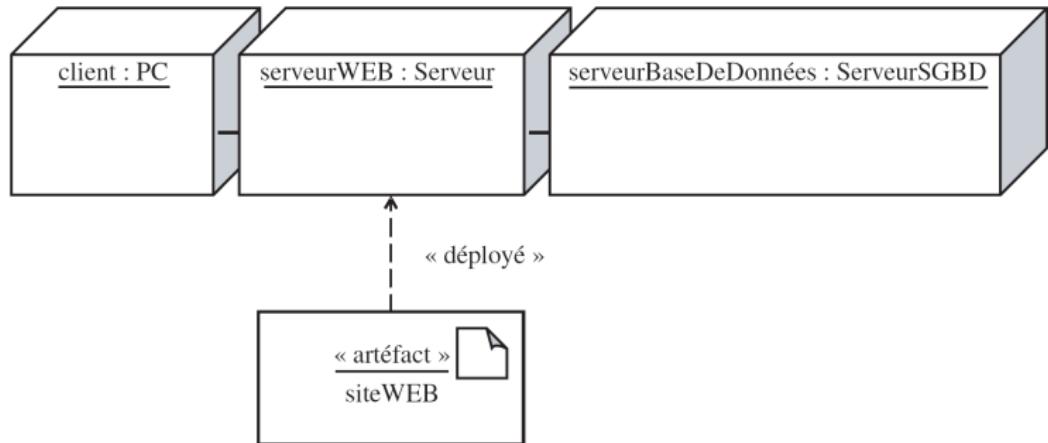
- ▶ En dernier lieu, un système doit s'exécuter sur des ressources matérielles dans un environnement matériel particulier.
- ▶ UML permet de représenter un environnement d'exécution ainsi que des ressources physiques (avec les parties du système qui s'y exécutent) grâce aux **diagrammes de déploiement**.
  - ▶ L'environnement d'exécution ou les ressources matérielles sont appelés "noeuds".
  - ▶ Les parties d'un système qui s'exécutent sur un noeud sont appelées "artefacts".

- ▶ Un **noeud** est une ressource sur laquelle des artefacts peuvent être déployés pour être exécutés.
- ▶ C'est un classeur qui peut prendre des attributs.
  - ▶ Un noeud se représente par un cube dont le nom respecte la syntaxe des noms de classes.
  - ▶ Les noeuds peuvent être associés comme des classes et on peut spécifier des multiplicités.



- ▶ Un **artifact** est la spécification d'une entité physique du monde réel.
- ▶ Il se représente comme une classe par un rectangle contenant le mot-clé **artifact** suivi du nom de l'artifact.
  - ▶ Un artifact déployé sur un noeud est symbolisé par une flèche en trait pointillé qui porte le stéréotype déployé et qui pointe vers le noeud.
  - ▶ L'artifact peut aussi être inclus directement dans le cube représentant le noeud.
- ▶ Plusieurs stéréotypes standard existent pour les artifacts : **document, exécutable, fichier, librairie, source**.

- ▶ Les noms des instances de noeuds et d'artefacts sont soulignés.



## Exécution des composants

- ▶ On utilise des flèches de dépendance pour montrer la capacité d'un noeud à prendre en charge un type de composant.



# Plan

Conception objet élémentaire avec UML

UML et méthodologie

Modélisation avancée avec UML

Bonnes pratiques de la modélisation objet

Design Patterns (1/2)

Design Patterns (2/2)

Architectures



# Plan

Conception objet élémentaire avec UML

UML et méthodologie

Modélisation avancée avec UML

Bonnes pratiques de la modélisation objet

Design Patterns (1/2)

Design Patterns (2/2)

Architectures



- ▶ Formes de comportement pour décrire :
  - ▶ des algorithmes
  - ▶ des comportements entre objets
  - ▶ des formes de communication entre objet

# Exemples de Behavioural patterns

- ▶ **Chain of Responsibility**
- ▶ **Command**
- ▶ **Interpreter**
- ▶ **Iterator**
- ▶ **Mediator**
- ▶ **Memento**
- ▶ **Observer**
- ▶ **State**
- ▶ **Strategy**
- ▶ **Template Method**
- ▶ **Visitor**



# Chain of Responsibility

- ▶ **Intention :**
  - ▶ Permet à un nombre quelconque de classes d'essayer de répondre à une requête sans connaître les possibilités des autres classes sur cette requête
- ▶ **Indications :**
  - ▶ Séparer les différentes étapes d'un traitement et d'implémenter facilement les relations d'héritage.

# Chain of Responsibility

## ► Intention :

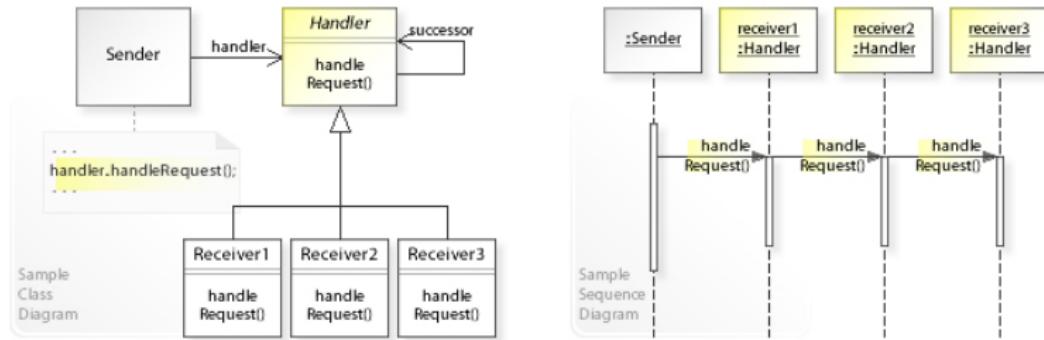
- ▶ Permet à un nombre quelconque de classes d'essayer de répondre à une requête sans connaître les possibilités des autres classes sur cette requête

## ► Indications :

- ▶ Séparer les différentes étapes d'un traitement et d'implémenter facilement les relations d'héritage.

Chain of Responsibility est une version orientée objet de “if ... else if ... else if ... else ... endif”.

# Chain of Responsibility Structure



# Chain of Responsibility Illustration 1/5

```
abstract class PurchasePower {  
    protected static final double BASE = 500;  
    protected PurchasePower successor;  
  
    abstract protected double getAllowable();  
    abstract protected String getRole();  
  
    public void setSuccessor(PurchasePower successor) {  
        this.successor = successor;  
    }  
  
    public void processRequest(PurchaseRequest request) {  
        if (request.getAmount() < this.getAllowable()) {  
            System.out.println(this.getRole() + " will approve $" + request.getAmount());  
        } else if (successor != null) {  
            successor.processRequest(request);  
        }  
    }  
}
```



# Chain of Responsibility Illustration 2/5

```
class ManagerPPower extends PurchasePower {  
  
    protected double getAllowable() {  
        return BASE * 10;  
    }  
  
    protected String getRole() {  
        return "Manager";  
    }  
}  
  
class DirectorPPower extends PurchasePower {  
  
    protected double getAllowable() {  
        return BASE * 20;  
    }  
  
    protected String getRole() {  
        return "Director";  
    }  
}
```



# Chain of Responsibility Illustration 3/5

```
class VicePresidentPPower extends PurchasePower {  
  
    protected double getAllowable() {  
        return BASE * 40;  
    }  
  
    protected String getRole() {  
        return "Vice President";  
    }  
}  
  
class PresidentPPower extends PurchasePower {  
  
    protected double getAllowable() {  
        return BASE * 60;  
    }  
  
    protected String getRole() {  
        return "President";  
    }  
}
```



# Chain of Responsibility Illustration 4/5

```
class PurchaseRequest {  
  
    private double amount;  
    private String purpose;  
  
    public PurchaseRequest(double amount, String purpose) {  
        this.amount = amount;  
        this.purpose = purpose;  
    }  
  
    public double getAmount() {  
        return this.amount;  
    }  
  
    public void setAmount(double amount) {  
        this.amount = amount;  
    }  
  
    public String getPurpose() {  
        return this.purpose;  
    }  
    public void setPurpose(String purpose) {  
        this.purpose = purpose;  
    }  
}
```



# Chain of Responsibility Illustration 5/5

```
class CheckAuthority {  
  
    public static void main(String[] args) {  
        ManagerPPower manager = new ManagerPPower();  
        DirectorPPower director = new DirectorPPower();  
        VicePresidentPPower vp = new VicePresidentPPower();  
        PresidentPPower president = new PresidentPPower();  
        manager.setSuccessor(director);  
        director.setSuccessor(vp);  
        vp.setSuccessor(president);  
  
        // Press Ctrl+C to end.  
        try {  
            while (true) {  
                System.out.println("Enter the amount to check who should approve your expenditure.");  
                System.out.print(">");  
                double d = Double.parseDouble(new BufferedReader(new InputStreamReader(System.in)).readLine());  
                manager.processRequest(new PurchaseRequest(d, "General"));  
            }  
        }  
        catch (Exception exc) {  
            System.exit(1);  
        }  
    }  
}
```



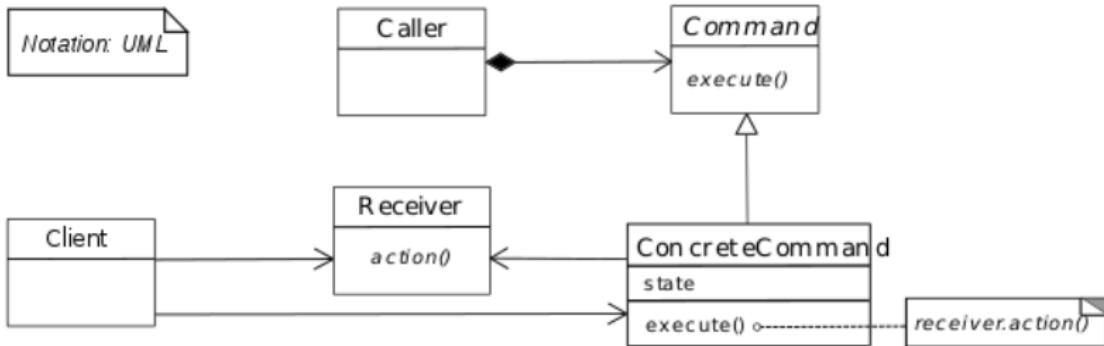
► **Intention :**

- Un objet est utilisé pour encapsuler toutes les informations nécessaires pour réaliser une action ou déclencher un événement. Ces informations peuvent être : le nom de la méthode, les valeurs des paramètres, ...

► **Indications :**

- Une classe délègue une requête à un objet "Command" au lieu d'implémenter la requête directement.

# Command Structure



# Command Illustration 1/3

```
/** The Command interface */
public interface Command {
    void execute();
}

/** The Invoker class */
public class Switch {
    private List<Command> history = new ArrayList<Command>();

    public void storeAndExecute(final Command cmd) {
        this.history.add(cmd); // optional
        cmd.execute();
    }
}

/** The Receiver class */
public class Light {

    public void turnOn() {
        System.out.println("The light is on");
    }

    public void turnOff() {
        System.out.println("The light is off");
    }
}
```



# Command Illustration 2/3

```
/** The Command for turning on the light - ConcreteCommand #1 */
public class FlipUpCommand implements Command {
    private Light theLight;

    public FlipUpCommand(final Light light) {
        this.theLight = light;
    }

    @Override      // Command
    public void execute() {
        theLight.turnOn();
    }
}

/** The Command for turning off the light - ConcreteCommand #2 */
public class FlipDownCommand implements Command {
    private Light theLight;

    public FlipDownCommand(final Light light) {
        this.theLight = light;
    }

    @Override      // Command
    public void execute() {
        theLight.turnOff();
    }
}
```



# Command Illustration 3/3

```
/* The test class or client */
public class PressSwitch {
    public static void main(final String[] arguments){
        // Check number of arguments
        if (arguments.length != 1) {
            System.err.println("Argument \"ON\" or \"OFF\" is required.");
            System.exit(-1);
        }

        final Light lamp = new Light();

        final Command switchUp = new FlipUpCommand(lamp);
        final Command switchDown = new FlipDownCommand(lamp);

        final Switch mySwitch = new Switch();

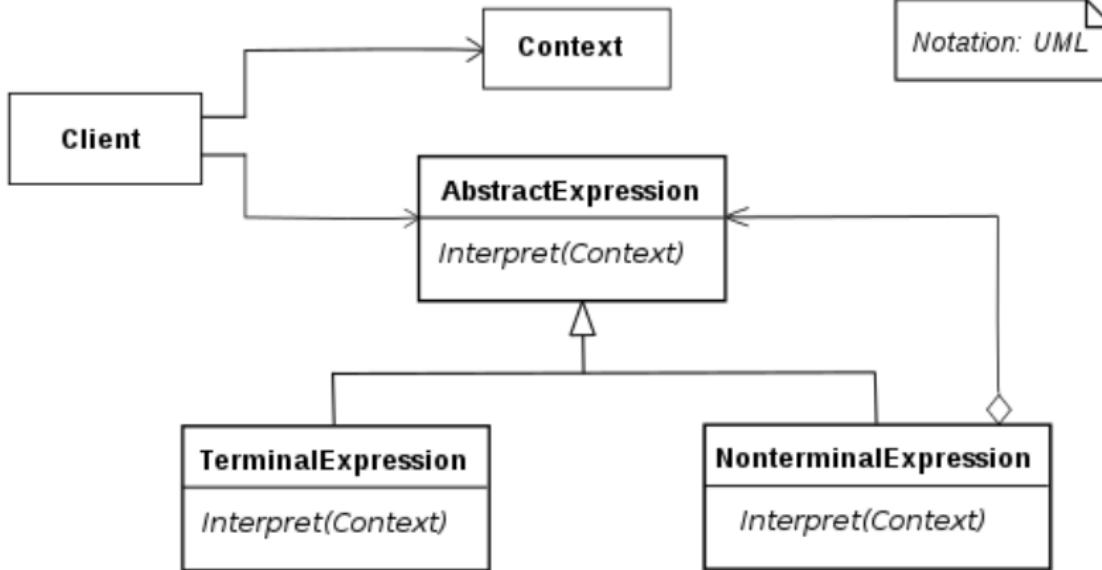
        switch(arguments[0]) {
            case "ON":
                mySwitch.storeAndExecute(switchUp);
                break;
            case "OFF":
                mySwitch.storeAndExecute(switchDown);
                break;
            default:
                System.err.println("Argument \"ON\" or \"OFF\" is required.");
                System.exit(-1);
        }
    }
}
```



Interpreter spécifie comment évaluer des phrases dans un langage.  
L'idée est d'avoir une classe pour chaque symbole (terminal et non-terminal).

L'arbre syntaxique d'une phrase dans le langage est une instance de pattern "Composite".

Notation: UML



```
import java.util.Map;

interface Expression {
    public int interpret(final Map<String, Expression> variables);
}

class Number implements Expression {
    private int number;
    public Number(final int number) { this.number = number; }
    public int interpret(final Map<String, Expression> variables) { return number; }
}
```



# Interpreter Illustration 2/5

```
class Plus implements Expression {  
    Expression leftOperand;  
    Expression rightOperand;  
    public Plus(final Expression left, final Expression right) {  
        leftOperand = left;  
        rightOperand = right;  
    }  
  
    public int interpret(final Map<String, Expression> variables) {  
        return leftOperand.interpret(variables) + rightOperand.interpret(variables);  
    }  
}  
  
class Minus implements Expression {  
    Expression leftOperand;  
    Expression rightOperand;  
    public Minus(final Expression left, final Expression right) {  
        leftOperand = left;  
        rightOperand = right;  
    }  
  
    public int interpret(final Map<String, Expression> variables) {  
        return leftOperand.interpret(variables) - rightOperand.interpret(variables);  
    }  
}
```



```
class Variable implements Expression {  
    private String name;  
    public Variable(final String name) { this.name = name; }  
    public int interpret(final Map<String, Expression> variables) {  
        if (null == variables.get(name)) return 0; // Either return new Number(0).  
        return variables.get(name).interpret(variables);  
    }  
}
```



# Interpreter Illustration 4/5

```

import java.util.Map;
import java.util.Stack;

class Evaluator implements Expression {
    private Expression syntaxTree;

    public Evaluator(final String expression) {
        final Stack<Expression> expressionStack = new Stack<Expression>();
        for (final String token : expression.split(" ")) {
            if (token.equals("+")) {
                final Expression subExpression = new Plus(expressionStack.pop(), expressionStack.pop());
                expressionStack.push(subExpression);
            } else if (token.equals("-")) {
                // it's necessary to remove first the right operand from the stack
                final Expression right = expressionStack.pop();
                // ..and then the left one
                final Expression left = expressionStack.pop();
                final Expression subExpression = new Minus(left, right);
                expressionStack.push(subExpression);
            } else
                expressionStack.push(new Variable(token));
        }
        syntaxTree = expressionStack.pop();
    }

    public int interpret(final Map<String, Expression> context) {
        return syntaxTree.interpret(context);
    }
}

```



# Interpreter Illustration 5/5

```
import java.util.Map;
import java.util.HashMap;

public class InterpreterExample {
    public static void main(final String[] args) {
        final String expression = "w x z - +";
        final Evaluator sentence = new Evaluator(expression);
        final Map<String, Expression> variables = new HashMap<String, Expression>();
        variables.put("w", new Number(5));
        variables.put("x", new Number(10));
        variables.put("z", new Number(42));
        final int result = sentence.interpret(variables);
        System.out.println(result);
    }
}
```



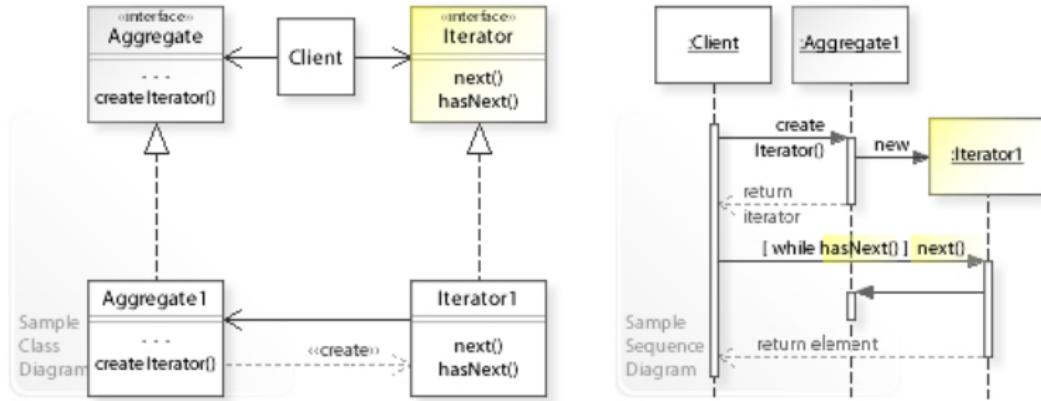
► **Intention :**

- Fournir un moyen d'accès séquentiel aux éléments d'un agrégat d'objets, sans rendre publique sa représentation interne.

► **Indications :**

- Pour accéder au contenu d'un objet d'un agrégat sans en révéler la représentation interne.
- Pour gérer simultanément plusieurs parcours dans un agrégat d'objets.
- Pour fournir une interface uniforme pour les parcours au travers de diverses structures d'agrégats.

# Iterator Structure



# Iterator Illustration (1/4)

```
import java.util.Iterator;
import java.util.HashSet;
import java.util.Set;

class RedHead implements Iterable<RedHead> {
    private Set<RedHead> redHeads = new HashSet<RedHead>();

    public void add(final RedHead redHead) {
        redHeads.add(redHead);
    }
    @Override
    public Iterator<RedHead> iterator() {
        return redHeads.iterator();
    }
}
```



## Iterator Illustration (2/4)

```
class Weasley extends RedHead {  
    private String name;  
  
    public Weasley(final String name) {  
        this.name = name;  
    }  
    public String toString() {  
        return this.name + " Weasley";  
    }  
}
```



# Iterator Illustration (3/4)

```
public static IteratorExample {
    public static void main(String[] args) {
        RedHead redHead = new RedHead();

        redHead.add(new Weasley("Arthur"));
        redHead.add(new Weasley("Molly"));
        redHead.add(new Weasley("Bill"));
        redHead.add(new Weasley("Charlie"));
        redHead.add(new Weasley("Percy"));
        redHead.add(new Weasley("Fred"));
        redHead.add(new Weasley("George"));
        redHead.add(new Weasley("Ron"));
        redHead.add(new Weasley("Ginny"));

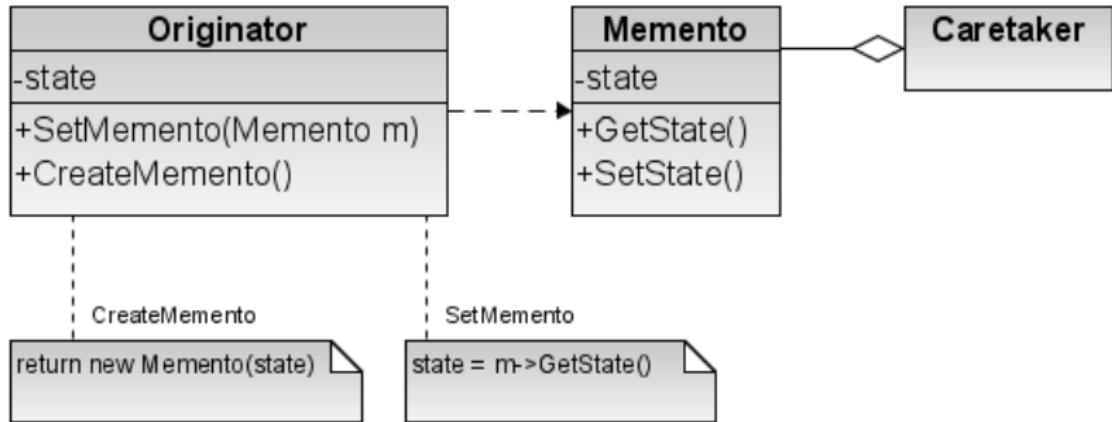
        for (ReadHead rh : redHead) {
            System.out.println(rh);
        }
    }
}
```



Ron Weasley  
Molly Weasley  
Percy Weasley  
Fred Weasley  
Charlie Weasley  
George Weasley  
Arthur Weasley  
Ginny Weasley  
Bill Weasley

- ▶ **Intention :**
  - ▶ Donne la possibilité de restaurer un objet à un état antérieur ("undo via rollback").
- ▶ **Indications :**
  - ▶ L'état interne d'un objet doit être sauvegarder à l'extérieur pour que l'objet puisse être restaurer à cet état plus tard.
  - ▶ Quand l'encapsulation d'un objet ne doit pas être violée.

# Memento Structure



# Memento Illustration (1/3)

```

import java.util.List;
import java.util.ArrayList;
class Originator {
    private String state;
    // The class could also contain additional data that is not part of the
    // state saved in the memento..

    public void set(String state) {
        this.state = state;
        System.out.println("Originator: Setting state to " + state);
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(this.state);
    }

    public void restoreFromMemento(Memento memento) {
        this.state = memento.getSavedState();
        System.out.println("Originator: State after restoring from Memento: " + state);
    }

    public static class Memento {
        private final String state;

        public Memento(String stateToSave) {
            state = stateToSave;
        }

        // accessible by outer class only
        private String getSavedState() {
            return state;
        }
    }
}

```



# Memento Illustration (2/3)

```
class Caretaker {
    public static void main(String[] args) {
        List<Originator.Memento> savedStates = new ArrayList<Originator.Memento>();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        savedStates.add(originator.saveToMemento());
        originator.set("State3");
        // We can request multiple mementos, and choose which one to roll back to.
        savedStates.add(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(savedStates.get(1));
    }
}
```



# Memento Illustration (3/3)

```
Originator: Setting state to State1
Originator: Setting state to State2
Originator: Saving to Memento.
Originator: Setting state to State3
Originator: Saving to Memento.
Originator: Setting state to State4
Originator: State after restoring from Memento: State3
```



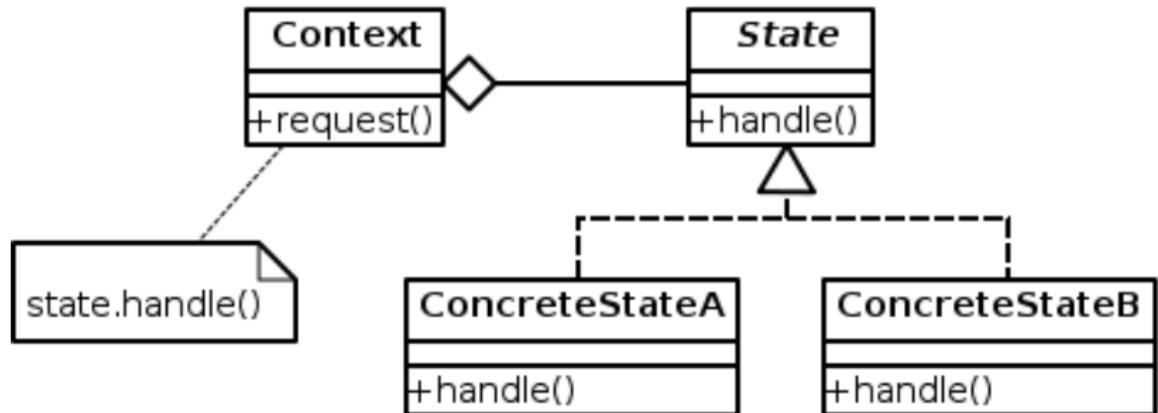
# State

## State

Il est utilisé lorsqu'il est souhaité de pouvoir changer le comportement d'un objet quand son état change, sans pour autant en changer l'instance.

Ce patron s'inspire du modèle des automates à états.





# State Illustration (1/4)

```
interface Statelike {
    void writeName(StateContext context, String name);
}

class StateLowerCase implements Statelike {
    @Override
    public void writeName(final StateContext context, final String name) {
        System.out.println(name.toLowerCase());
        context.setState(new StateMultipleUpperCase());
    }
}

class StateMultipleUpperCase implements Statelike {
    /** Counter local to this state */
    private int count = 0;

    @Override
    public void writeName(final StateContext context, final String name) {
        System.out.println(name.toUpperCase());
        /* Change state after StateMultipleUpperCase's writeName() gets invoked twice */
        if(++count > 1) {
            context.setState(new StateLowerCase());
        }
    }
}
```



# State Illustration (2/4)

```
class StateContext {
    private Statelike myState;
    StateContext() {
        setState(new StateLowerCase());
    }

    /**
     * Setter method for the state.
     * Normally only called by classes implementing the State interface.
     * @param newState the new state of this context
     */
    void setState(final Statelike newState) {
        myState = newState;
    }

    public void writeName(final String name) {
        myState.writeName(this, name);
    }
}
```



```
public class DemoOfClientState {  
    public static void main(String[] args) {  
        final StateContext sc = new StateContext();  
  
        sc.writeName("Monday");  
        sc.writeName("Tuesday");  
        sc.writeName("Wednesday");  
        sc.writeName("Thursday");  
        sc.writeName("Friday");  
        sc.writeName("Saturday");  
        sc.writeName("Sunday");  
    }  
}
```



monday

TUESDAY

WEDNESDAY

thursday

FRIDAY

SATURDAY

sunday



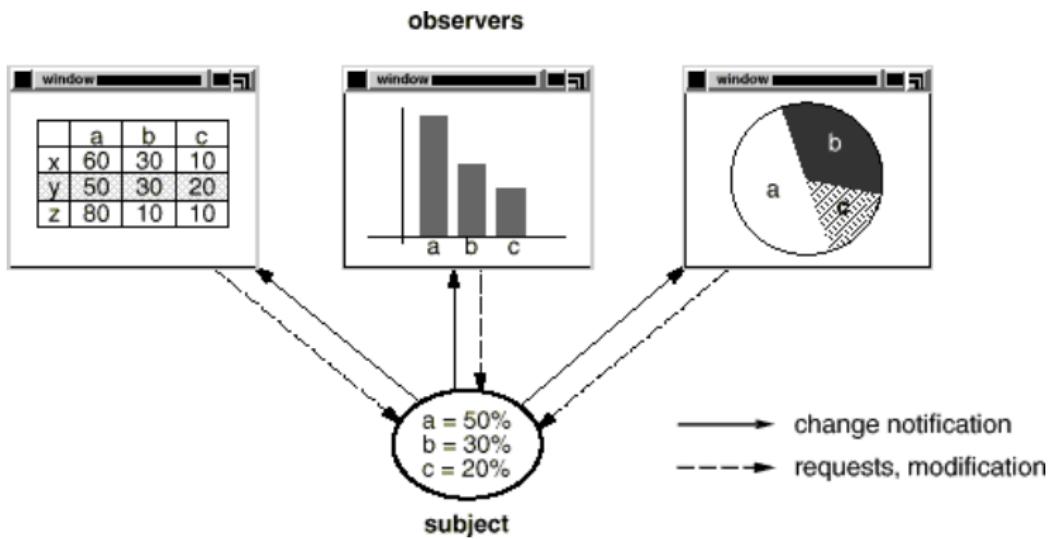
► **Intention :**

- Définir une dépendance de type “un à plusieurs” de façon à ce que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour.

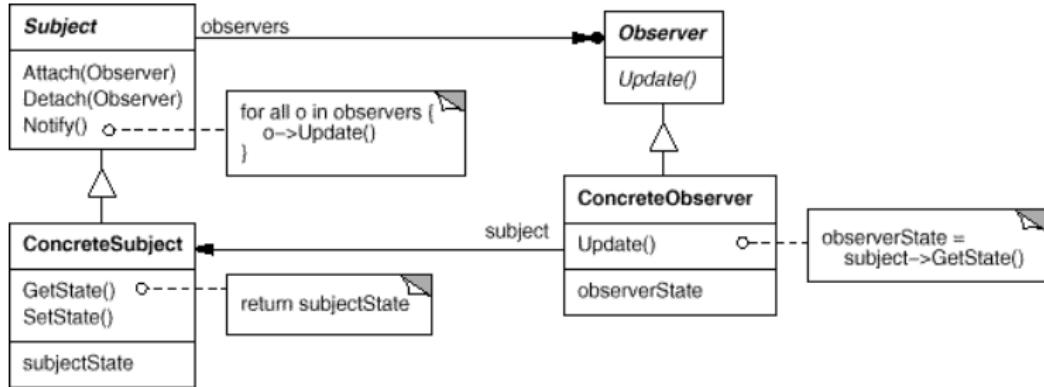
► **Indications :**

- Quand un concept a plusieurs représentations, l'une dépendant de l'autre.
- Quand une modification d'un objet nécessite la modification d'autres objets, sans qu'on en connaisse le nombre exact.
- Quand un objet doit notifier d'autres objets avec lesquels il n'est pas fortement couplé.

# Observer Motivation



# Observer Structure



```
import java.util.Observable;
import java.util.Scanner;

class EventSource extends Observable implements Runnable {
    public void run() {
        while (true) {
            String response = new Scanner(System.in).next();
            setChanged();
            notifyObservers(response);
        }
    }
}
```



```
import java.util.Observable;
import java.util.Observer;

public class MyApp {
    public static void main(String[] args) {
        System.out.println("Enter Text: ");
        EventSource eventSource = new EventSource();

        eventSource.addObserver((obj, arg) -> {
            System.out.println("Received response: " + arg);
        });

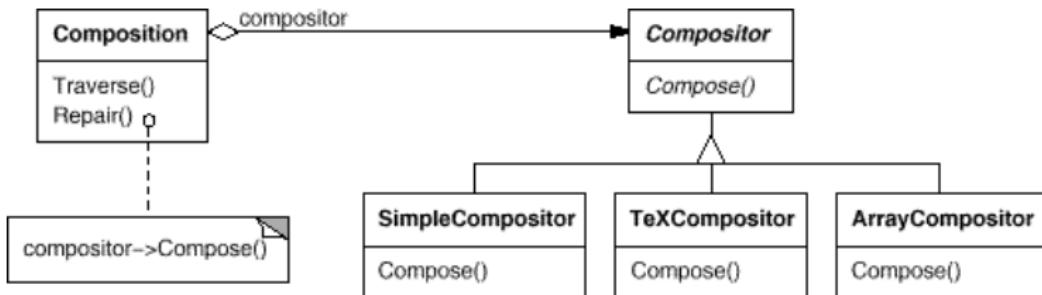
        new Thread(eventSource).start();
    }
}
```



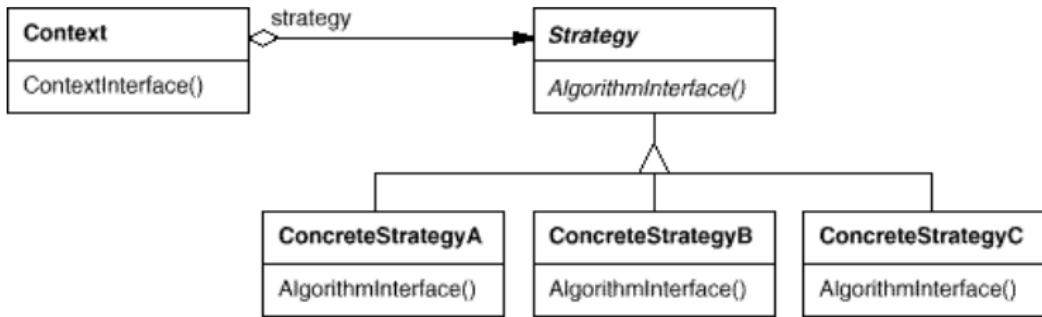
- ▶ **Intention :**
  - ▶ Définir une famille d'algorithmes en encapsulant chacun d'eux et en les rendant interchangeables.
- ▶ **Indications :**
  - ▶ Plusieurs classes apparentées ne diffèrent que par leur comportement.
  - ▶ On a besoin de plusieurs variantes d'un algorithme.
  - ▶ Un algorithme utilise des données que les clients n'ont pas à connaître.
  - ▶ Une classe définit un ensemble de comportements qui figurent dans des opérations sous la forme de déclarations conditionnelles multiples.

# Strategy Motivation

353



# Strategy Structure



# Strategy Illustration 1/3

```
class Customer {  
  
    private List<Double> drinks;  
    private BillingStrategy strategy;  
  
    public Customer(final BillingStrategy strategy) {  
        this.drinks = new ArrayList<Double>();  
        this.strategy = strategy;  
    }  
  
    public void add(final double price, final int quantity) {  
        drinks.add(strategy.getActPrice(price*quantity));  
    }  
  
    // Payment of bill  
    public void printBill() {  
        double sum = 0;  
        for (Double i : drinks) {  
            sum += i;  
        }  
        System.out.println("Total due: " + sum);  
        drinks.clear();  
    }  
  
    // Set Strategy  
    public void setStrategy(final BillingStrategy strategy) {  
        this.strategy = strategy;  
    }  
}
```



# Strategy Illustration 2/3

```
interface BillingStrategy {
    double getActPrice(final double rawPrice);
}

// Normal billing strategy (unchanged price)
class NormalStrategy implements BillingStrategy {

    @Override
    public double getActPrice(final double rawPrice) {
        return rawPrice;
    }
}

// Strategy for Happy hour (50% discount)
class HappyHourStrategy implements BillingStrategy {

    @Override
    public double getActPrice(final double rawPrice) {
        return rawPrice*0.5;
    }
}
```



# Strategy Illustration 3/3

```
public class StrategyPatternWiki {  
  
    public static void main(final String[] arguments) {  
        // Prepare strategies  
        BillingStrategy normalStrategy = new NormalStrategy();  
        BillingStrategy happyHourStrategy = new HappyHourStrategy();  
  
        Customer firstCustomer = new Customer(normalStrategy);  
  
        // Normal billing  
        firstCustomer.add(1.0, 1);  
  
        // Start Happy Hour  
        firstCustomer.setStrategy(happyHourStrategy);  
        firstCustomer.add(1.0, 2);  
  
        // New Customer  
        Customer secondCustomer = new Customer(happyHourStrategy);  
        secondCustomer.add(0.8, 1);  
        // The Customer pays  
        firstCustomer.printBill();  
  
        // End Happy Hour  
        secondCustomer.setStrategy(normalStrategy);  
        secondCustomer.add(1.3, 2);  
        secondCustomer.add(2.5, 1);  
        secondCustomer.printBill();  
    }  
}
```



# Utilisation des design patterns

- ▶ Les design patterns ne sont que des éléments de conception
- ▶ On les combine pour produire une conception de qualité
  - ▶ Trouver les bons objets
  - ▶ Mieux réutiliser, concevoir pour l'évolution
- ▶ On peut aussi produire de nouveaux design patterns
  - ▶ Les DP du GOF ne sont pas les seuls, par exemple :
    - ▶ Architecture 3-tiers
    - ▶ Modèle Vue Contrôleur : Combinaison de Composite, Observer et Strategy
  - ▶ On trouve facilement sur Internet des bibliothèques de patterns



# Plan

Conception objet élémentaire avec UML

UML et méthodologie

Modélisation avancée avec UML

Bonnes pratiques de la modélisation objet

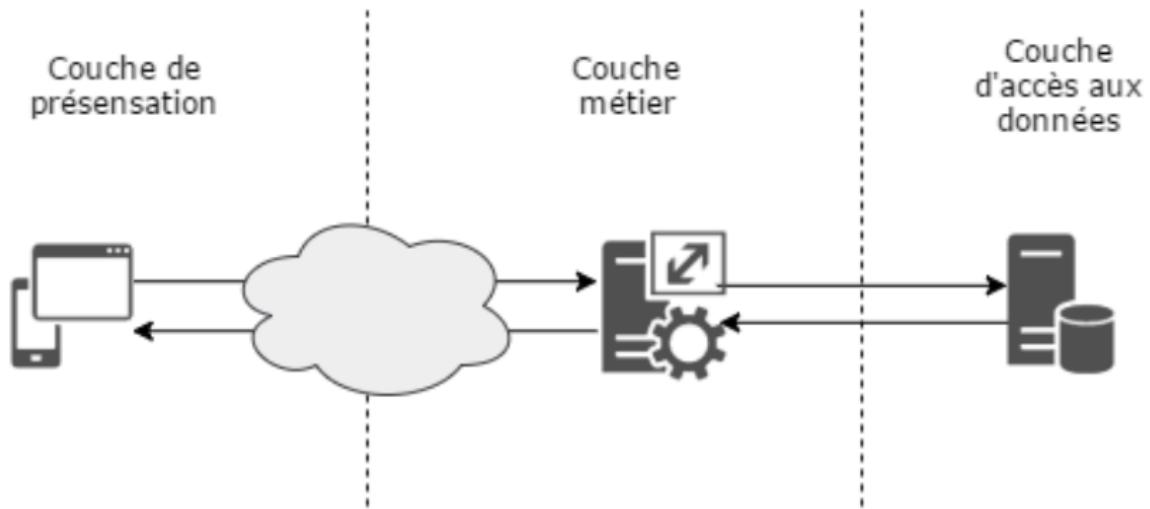
Design Patterns (1/2)

Design Patterns (2/2)

Architectures



# Architecture 3-Tiers



## Avantages

- ▶ Simplicité d'implémentation
- ▶ Évolutivité de l'application



# Architecture 3-Tiers

## Avantages

- ▶ Simplicité d'implémentation
- ▶ Évolutivité de l'application

## Inconvénients

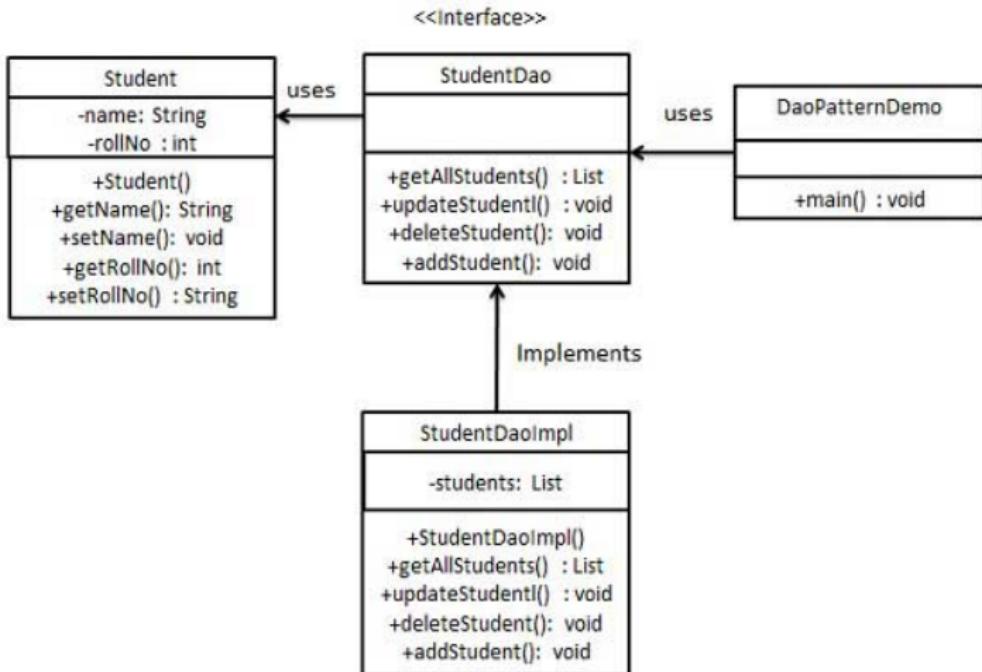
- ▶ La couche métier réalise la majorité des traitements.
- ▶ Difficulté : Les développeurs ont besoins de connaître plusieurs technologies.



## Les couches

- ▶ Présentation: La couche présentation a pour responsabilité d'afficher les résultats auprès du client.
- ▶ Métier: La couche métier est la couche où l'on implémente les traitements du système d'information étudié.
- ▶ Données: La couche données est la couche d'accès et au stockage des données.

# Architecture 3-Tiers : Données (1/6)



## Architecture 3-Tiers : Données (2/6)

```
public class Student {  
    private String name;  
    private int rollNo;  
  
    Student(String name, int rollNo){  
        this.name = name;  
        this.rollNo = rollNo;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getRollNo() {  
        return rollNo;  
    }  
  
    public void setRollNo(int rollNo) {  
        this.rollNo = rollNo;  
    }  
}
```



## Architecture 3-Tiers : Données (3/6)

```
import java.util.List;

public interface StudentDao {
    public List<Student> getAllStudents();
    public Student getStudent(int rollNo);
    public void updateStudent(Student student);
    public void deleteStudent(Student student);
}
```



# Architecture 3-Tiers : Données (4/6)

```
import java.util.ArrayList;
import java.util.List;

public class StudentDaoImpl implements StudentDao {

    //list is working as a database
    List<Student> students;

    public StudentDaoImpl(){
        students = new ArrayList<Student>();
        Student student1 = new Student("Robert",0);
        Student student2 = new Student("John",1);
        students.add(student1);
        students.add(student2);
    }

    @Override
    public void deleteStudent(Student student) {
        students.remove(student.getRollNo());
        System.out.println("Student: Roll No " + student.getRollNo() + ", dele");
    }

    //retrieve list of students from the database
    @Override
    public List<Student> getAllStudents() {
        return students;
    }

    @Override
    public Student getStudent(int rollNo) {
        return students.get(rollNo);
    }

    @Override
    public void updateStudent(Student student) {
        students.get(student.getRollNo()).setName(student.getName());
        System.out.println("Student: Roll No " + student.getRollNo() + ", upda");
    }
}
```



# Architecture 3-Tiers : Données (5/6)

```
public class DaoPatternDemo {  
    public static void main(String[] args) {  
        StudentDao studentDao = new StudentDaoImpl();  
  
        //print all students  
        for (Student student : studentDao.getAllStudents()) {  
            System.out.println("Student: [RollNo : " + student.getRollNo() + ",  
        }  
  
        //update student  
        Student student =studentDao.getAllStudents().get(0);  
        student.setName("Michael");  
        studentDao.updateStudent(student);  
  
        //get the student  
        studentDao.getStudent(0);  
        System.out.println("Student: [RollNo : " + student.getRollNo() + ", Na  
    }  
}
```



```
Student: [RollNo : 0, Name : Robert ]
```

```
Student: [RollNo : 1, Name : John ]
```

```
Student: Roll No 0, updated in the database
```

```
Student: [RollNo : 0, Name : Michael ]
```

## Métier

- ▶ Les filtrages nécessaires entre les données en DAO et l'affichage
- ▶ Envoies des données vers la partie graphique
- ▶ Plein de technologies en Java : Spring, Tapestry, ...

```
package org.example.demo.pages;

public class HelloWorld {

    /** An ordinary getter */
    public String getUsername() {
        return "World";
    }
}
```

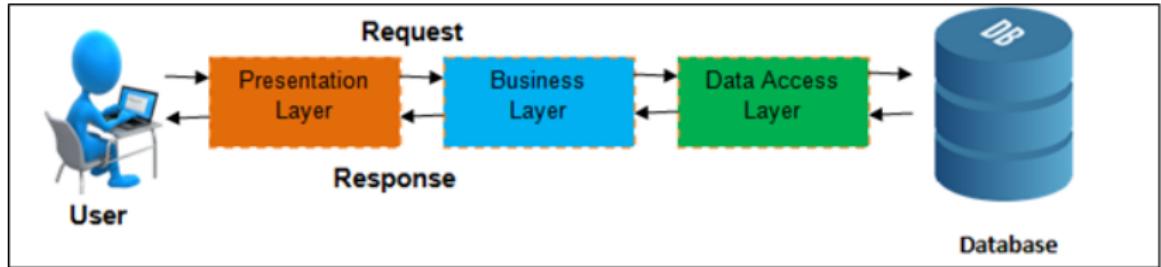


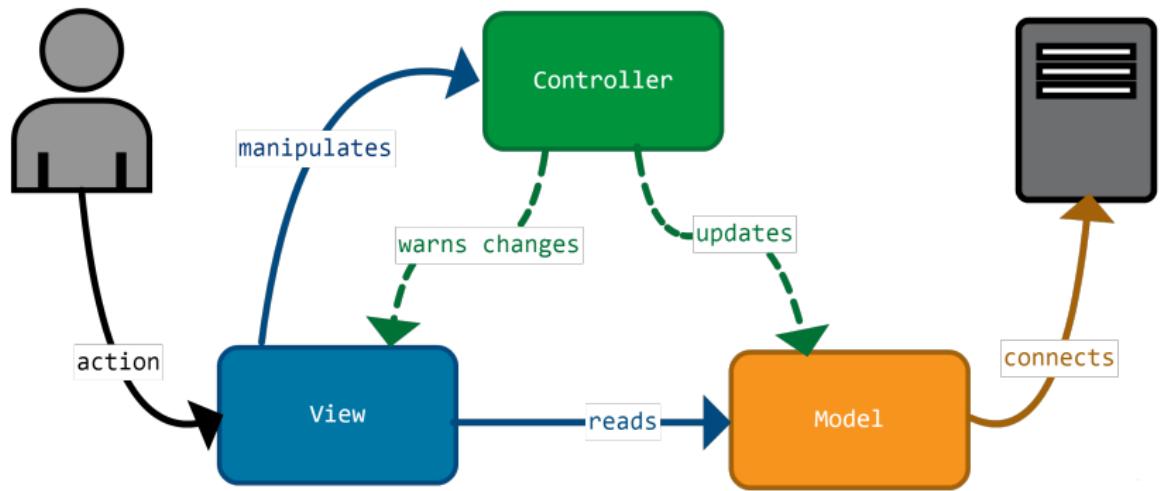
# Architecture 3-Tiers : Tapestry

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:t="http://tapestry.apache.org/schema/tapestry_5_3.xsd">
<body>
    <p>Hello, ${username}</p>
</body>
</html>
```



# Architecture 3-Tiers





## Avantages

- ▶ Développement en simultané
- ▶ Faible couplage
- ▶ Facilité de modification
- ▶ Un modèle peut avoir plusieurs vues



## Avantages

- ▶ Développement en simultané
- ▶ Faible couplage
- ▶ Facilité de modification
- ▶ Un modèle peut avoir plusieurs vues

## Inconvénients

- ▶ La navigabilité dans le code (beaucoup de couches d'abstractions)
- ▶ Difficulté : Les développeurs ont besoins de connaître plusieurs technologies.

## Différence avec 3-Tiers

- ▶ L'architecture trois tiers est un modèle en couches, c'est-à-dire que chaque couche communique seulement avec ses couches adjacentes (supérieures et inférieures) et le flux de contrôle traverse le système de haut en bas.
- ▶ Les couches supérieures sont toujours sources d'interaction (clients) alors que les couches inférieures ne font que répondre à des requêtes (serveurs).

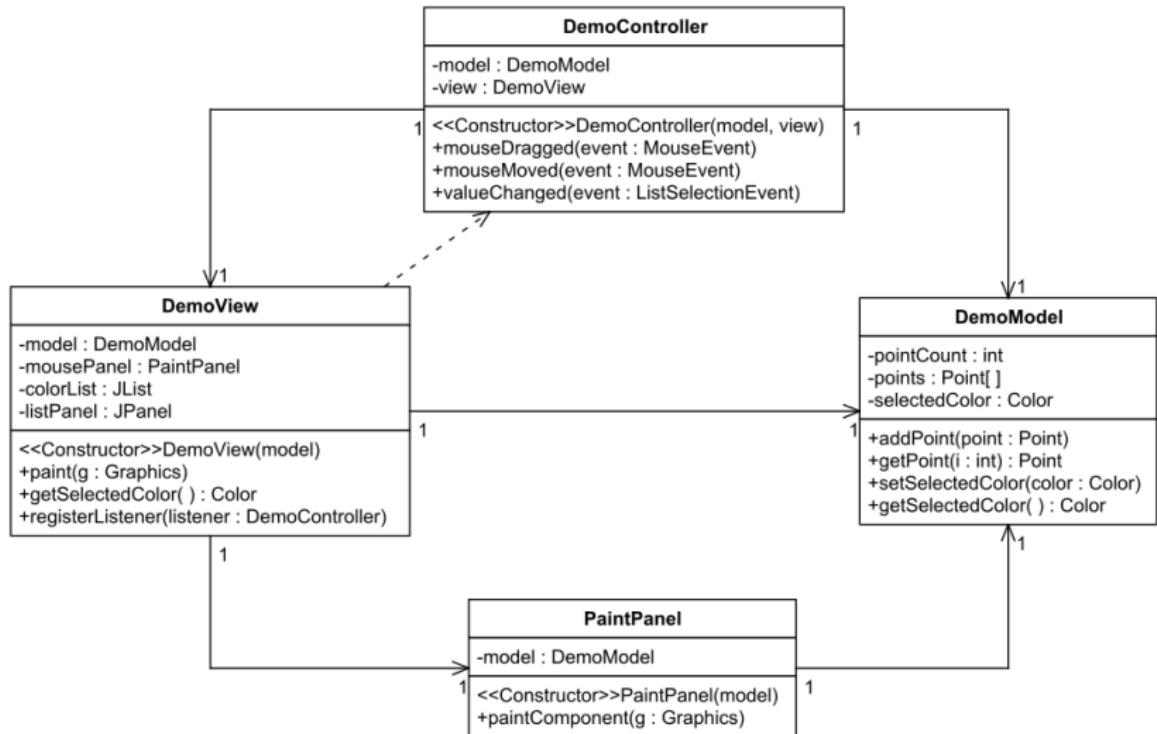


## Différence avec 3-Tiers

- ▶ L'architecture trois tiers est un modèle en couches, c'est-à-dire que chaque couche communique seulement avec ses couches adjacentes (supérieures et inférieures) et le flux de contrôle traverse le système de haut en bas.
- ▶ Les couches supérieures sont toujours sources d'interaction (clients) alors que les couches inférieures ne font que répondre à des requêtes (serveurs).

Dans le modèle MVC, il est généralement admis que la vue puisse consulter directement le modèle (lecture) sans passer par le contrôleur. Par contre, elle doit nécessairement passer par le contrôleur pour effectuer une modification (écriture).

# Modèle Vue Contrôleur





# Modèle Vue Contrôleur

## HelloWorldController.java

```
package com.mkyong.common.controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class HelloWorldController extends AbstractController{

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        ModelAndView model = new ModelAndView("HelloWorldPage");
        model.addObject("msg", "hello world");

        return model;
    }
}
```



## HelloWorldPage.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<body>
    <h1>Spring MVC Hello World Example</h1>

    <h2>${msg}</h2>
</body>
</html>
```



A screenshot of a web browser window. The address bar shows the URL `localhost:8080/spring2/welcome.htm`. The main content area displays the text **Spring MVC Hello World Example** followed by **hello world**.



