

Introduction à UML

Modélisation Orientée Objet

30 mars 2020

Plan

- 1 Introduction à la Modélisation Orientée Objet
- 2 Modélisation objet élémentaire avec UML

Références de ce cours

- Adaptation des slides de Pierre Gérard (P13 IUT Villetaneuse).
- « **UML 2.0, guide de référence** »
 - James Rumbaugh, Ivar Jacobson, Grady Booch

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.

Critères de qualité d'un logiciel

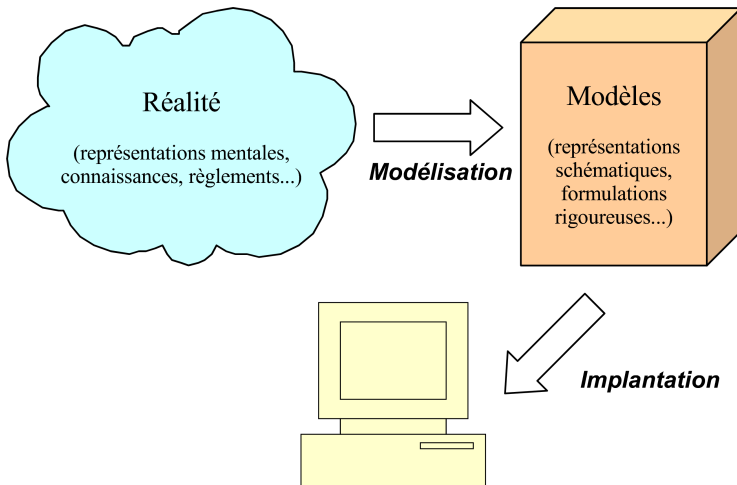
- **Utilité**
 - Adéquation entre le logiciel et les besoins des utilisateurs ;
- **Utilisabilité**
- **Fiabilité**
- **Interopérabilité**
 - Interactions avec d'autres logiciels ;
- **Performance**
- **Portabilité**
- **Réutilisabilité**
- **Facilité de maintenance**
 - Un logiciel ne s'use pas pourtant, 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.

Modélisation

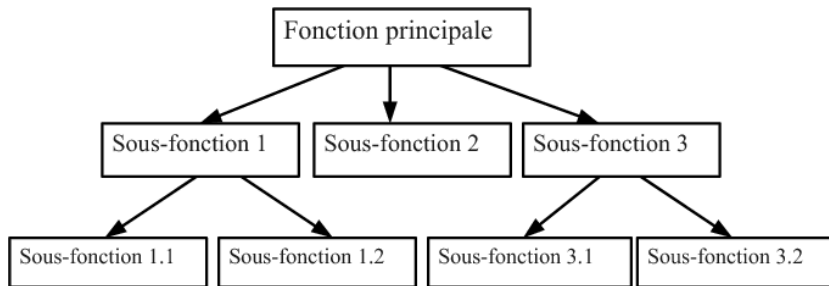


Modélisation par décomposition fonctionnelle

- 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.



Modélisation orientée objets

- 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

- Au milieu des années 90, les auteurs de Booch, OOSE et OMT ont décidé de créer un langage de modélisation unifié 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 mathématiques que des langages de spécification formelle (Z, VDM...).
- Officiellement UML est né en 1994.

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.

Plan

1 Introduction à la Modélisation Orientée Objet

2 Modélisation objet élémentaire avec UML

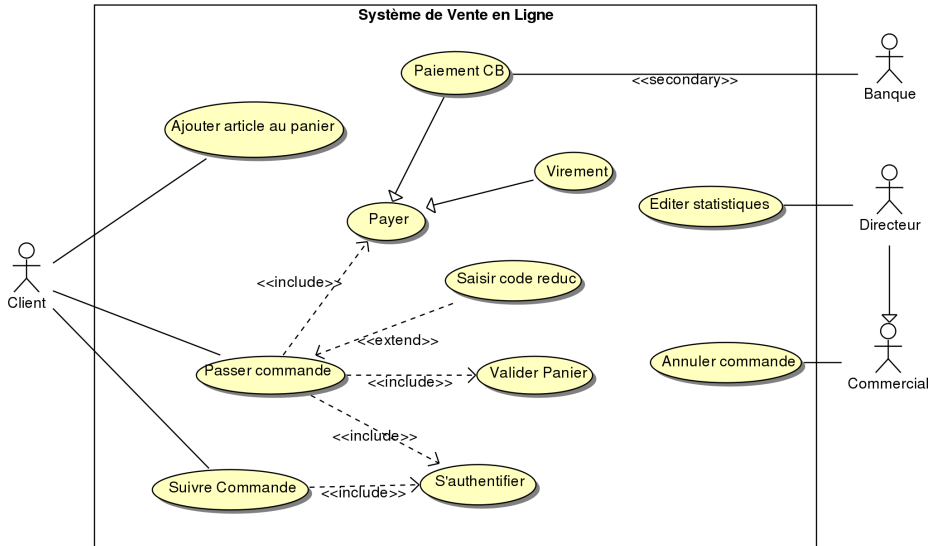
- Diagrammes de cas d'utilisation
- Diagrammes de classes
- Diagrammes d'objets
- Diagrammes de séquences

Modélisation des besoins

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.



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

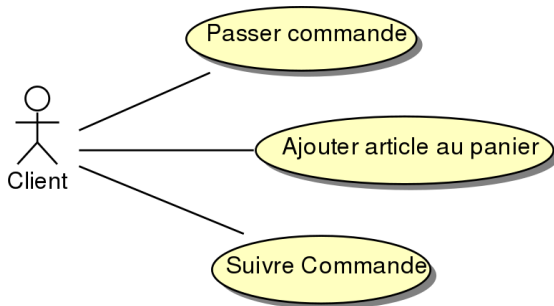


A yellow oval with a black border, representing a use case in UML notation.

Cas d'utilisation

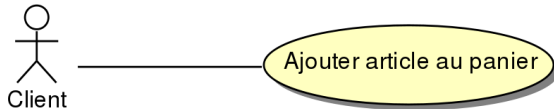
Un cas d'utilisation est l'expression d'un service réalisé de bout en bout, avec un déclenchement, un déroulement et une fin, pour l'acteur qui l'initie.

Acteurs et cas d'utilisation



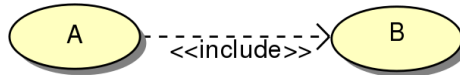
Relations entre cas d'utilisation en acteurs

- 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.

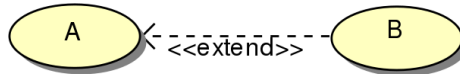


Relations entre cas d'utilisation

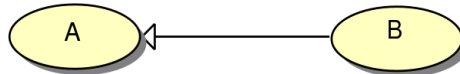
- **Inclusion** : le cas A inclut le cas B (B est une partie *obligatoire* de A).



- **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).



Dépendances d'inclusion et d'extension

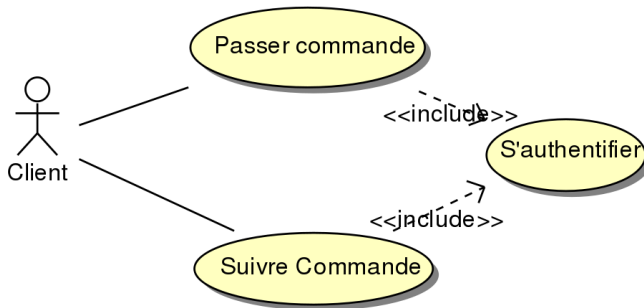
Attention

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 $B \cdots \rightarrow A$ 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 « incude » et les « extend » sont des **stéréotypes** (entre guillemets) des relations de dépendance.

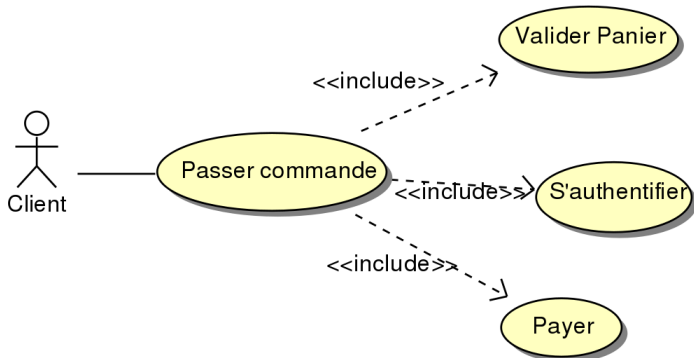
Réutilisabilité avec les inclusions et les extensions

- Les relations entre cas permettent la **réutilisabilité** du cas « s'authentifier » : il sera inutile de développer plusieurs fois un module d'authentification.

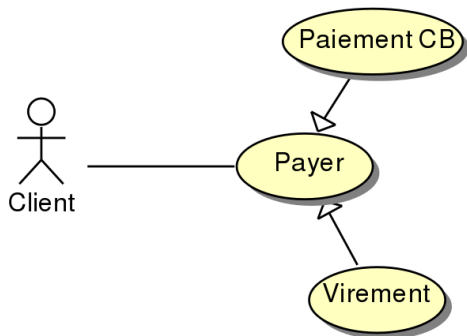


Décomposition grâce aux inclusions et aux extensions

- 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



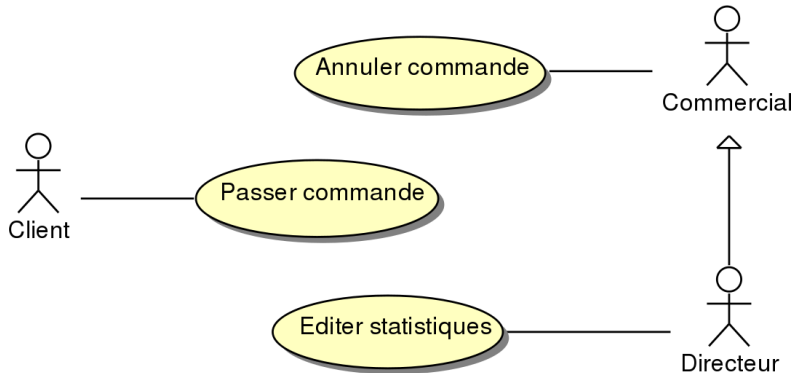
- Un virement est un cas particulier de paiement.

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.

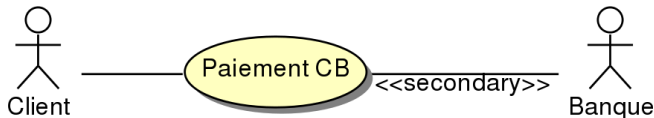
Attention

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

- **Identification :**

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

Description textuelle

- **Séquencements :**

- Le cas d'utilisation commence lorsqu'un client demande le paiement par carte bancaire

- **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

- **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

1 Introduction à la Modélisation Orientée Objet

2 **Modélisation objet élémentaire avec UML**

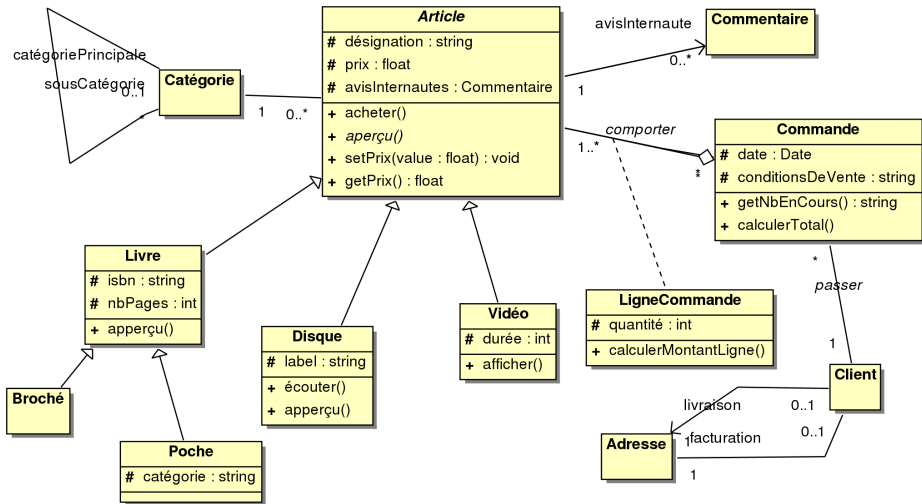
- Diagrammes de cas d'utilisation
- **Diagrammes de classes**
- Diagrammes d'objets
- Diagrammes de séquences

Objectif

- 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

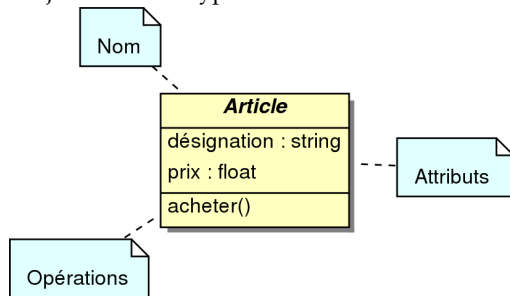
- Une **instance** est la concrétisation d'un **concept** abstrait.
 - Concept : Stylo
 - Instance : le stylo que vous utilisez à ce moment précis est une instance du concept stylo : il a sa propre forme, sa propre couleur, son propre niveau d'usure, etc.
- Un **objet** est une instance d'une **classe**
 - Classe : Vidéo
 - Objets : Pink Floyd (Live à Pompey), 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 forcément toutes connues.
- D'autres compartiments peuvent être ajoutés : responsabilités, exceptions, etc.

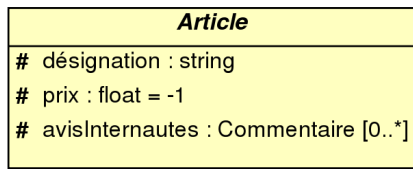
Propriétés : attributs et opérations

- 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]=valeurInit
```



Compartiment des opérations

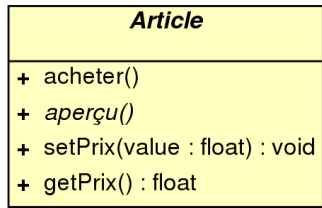
Une opération est définie par son ainsi que par les types de ses paramètres et le type de sa valeur de retour : **sa signature**

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

```
modifAcces nomOperation(parametres):ClasseRetour
```

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

```
nomClasse1 nomParam1, ... , nomClasseN nomParamN
```



Méthodes et Opérations

- Une **opération** est la signature d'une **méthode** indépendamment de son implantation.
 - UML 2 autorise également la définition des méthodes dans n'importe quel langage de programmation donné.
- **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 ;
}
```

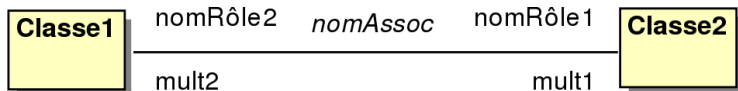
```
{ // implementation recursive
  if (n=0)
    return 1 ;
  return (n * fact(n-1)) ;
}
```

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.

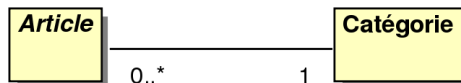
Association

- 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

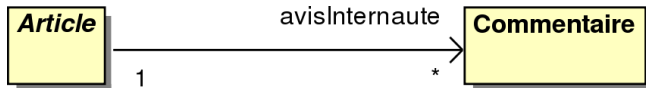
- La notion de **multiplicité** permet de contraindre le nombre d'objets intervenant dans les instanciations des associations.
 - Exemple :** un article n'appartient qu'à une seule catégorie (1); une catégorie concerne plus de 0 articles, sans maximum (0..*).



- 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 « * ».

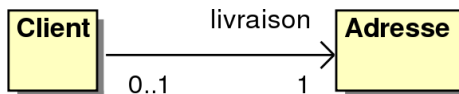
Navigabilité d'une association

- 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.

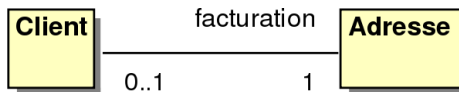
Association unidirectionnelle de 1 vers 1



Implantation

```
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

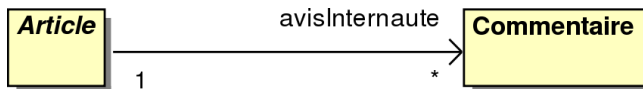


Implantation

```
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 *

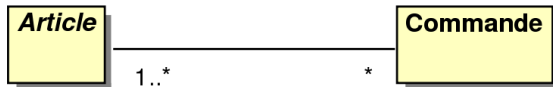


Implantation

```
public class Commentaire{...}

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

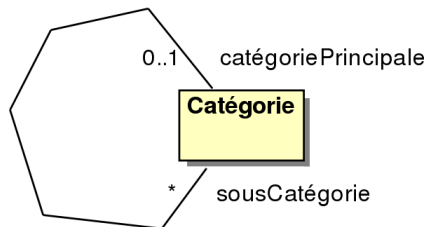

Implantation d'une association bidirectionnelle de * vers *



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

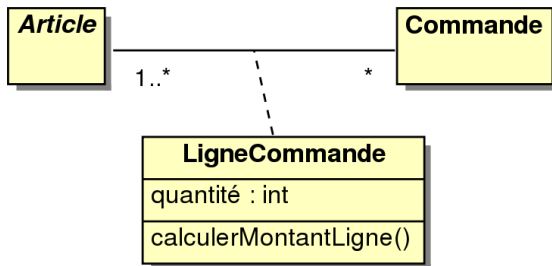
Associations réflexives

- 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** ».



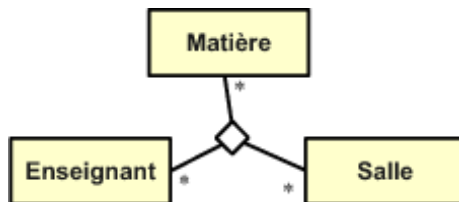
Classe-association

- 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.

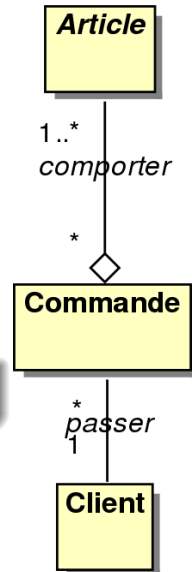


Les associations n-aires sont peu fréquentes et concernent surtout les cas où les multiplicités sont toutes « * ». Dans la plupart des cas, on utilisera plus avantageusement des classes-association ou plusieurs relations binaires.

Association de type agrégation

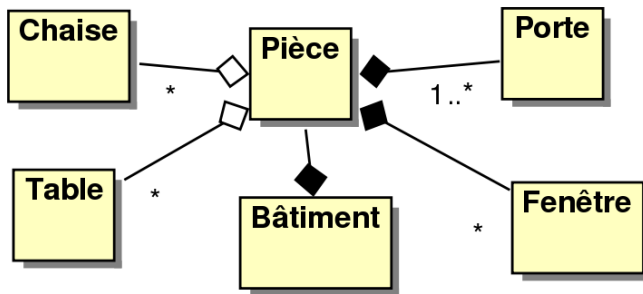
- Une **agrégation** est une forme particulière d'association. Elle représente la relation d'**inclusion** d'un élément dans un ensemble.
- On représente l'agrégation par l'ajout d'un losange vide du côté de l'agrégat.

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** ».

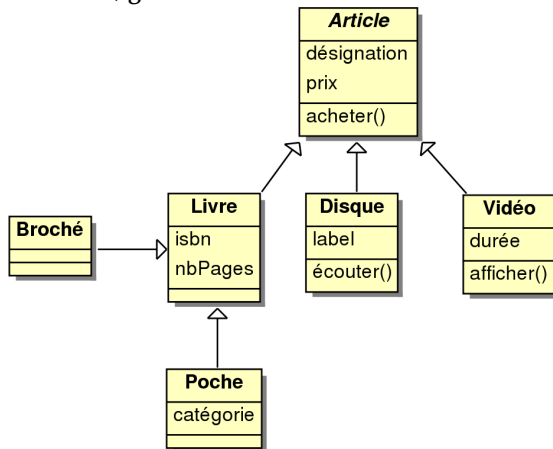
- **Pour décider de mettre une composition plutôt qu'une agrégation**, on doit se poser les questions suivantes :
 - Est-ce que la destruction de l'objet composite (du tout) implique nécessairement la destruction des objets composants (les parties) ? C'est le cas si les composants n'ont pas d'autonomie 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

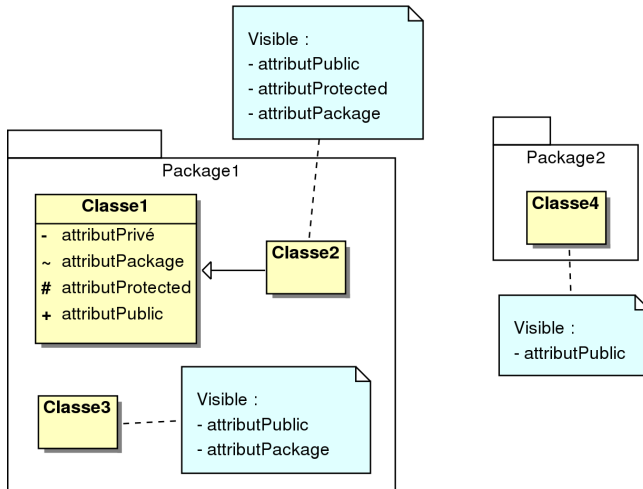
Encapsulation

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

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.

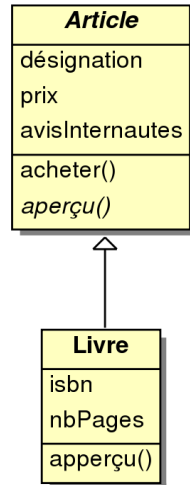
Exemple d'encapsulation



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

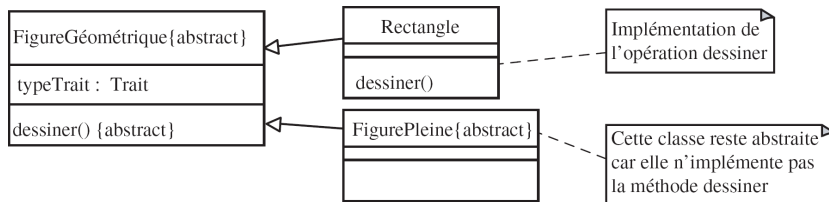
- Une classe enfant peut **redéfinir** (même signature) une ou plusieurs méthodes de la classe parent.
 - Sauf indications contraires, un objet utilise les opérations les plus spécialisées dans la hiérarchie des classes.
 - La **surcharge d'opérations** (même nom, mais signatures des opérations différentes) 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.

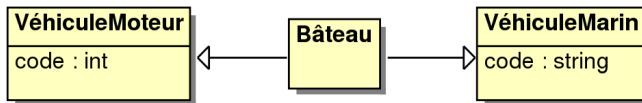
Il appartient aux classes enfant de définir les méthodes abstraites.

- 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.



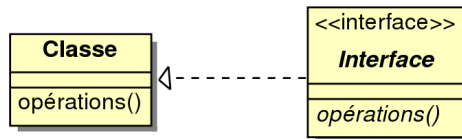
Héritage multiple

- 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

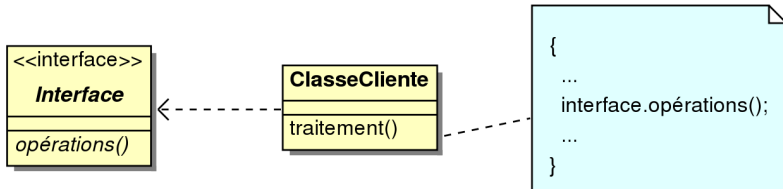
- Le rôle d'une **interface** est de regrouper un ensemble d'opérations assurant un service cohérent offert par une classe.
- Une interface est définie comme une classe, avec les mêmes compartiments. On ajoute le stéréotype « interface » avant le nom de l'interface.
- On utilise une relation de type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



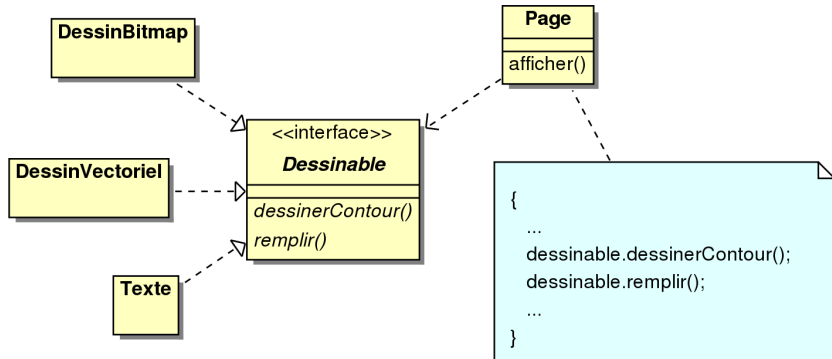
- Les classes implémentant une interface doivent implémenter toutes les opérations décrites dans l'interface

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

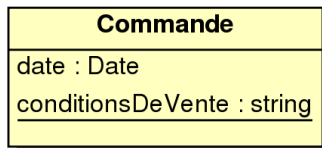


Eléments dérivés

- 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 « / ».

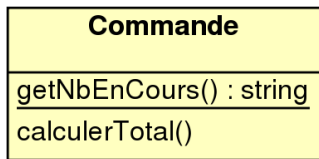
Attributs de classe

- 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



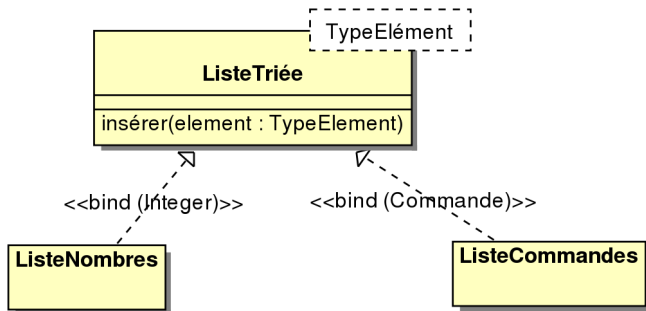
Opérations de classe

- 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**

Diagrammes de classes à différentes étapes de la conception

- 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)

Construction d'un diagramme de classes

- ❶ Trouver les **classes du domaine étudié** ;
 - Souvent, concepts et substantifs du domaine.
- ❷ Trouver les **associations entre classes** ;
 - Souvent, verbes mettant en relation plusieurs classes.
- ❸ 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.
- ❹ **Organiser et simplifier** le modèle en utilisant l'héritage ;
- ❺ **Tester** les chemins d'accès aux classées ;
- ❻ **Itérer et raffiner** le modèle.

Plan

1 Introduction à la Modélisation Orientée Objet

2 Modélisation objet élémentaire avec UML

- Diagrammes de cas d'utilisation
- Diagrammes de classes
- Diagrammes d'objets
- Diagrammes de séquences

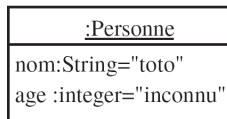
Objectif

- 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).



(a)



(b)



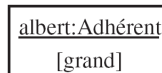
(c)



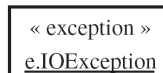
(d)



(e)



(f)



(g)

Diagramme de classes et diagramme d'objets

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

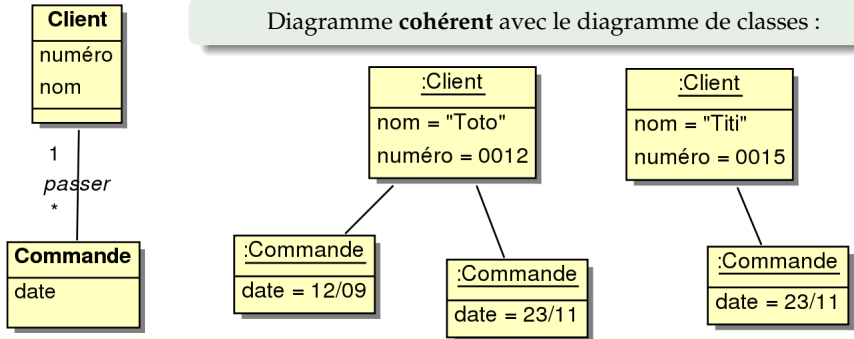


Diagramme de classes et diagramme d'objets

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

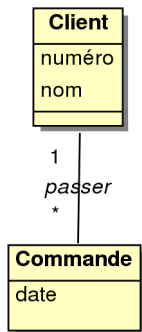
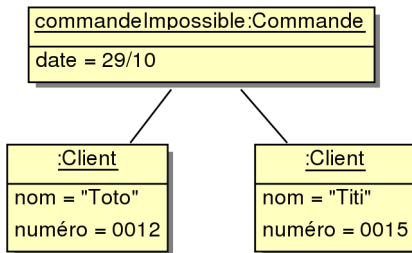


Diagramme **incohérent** avec le diagramme de classes :



Liens

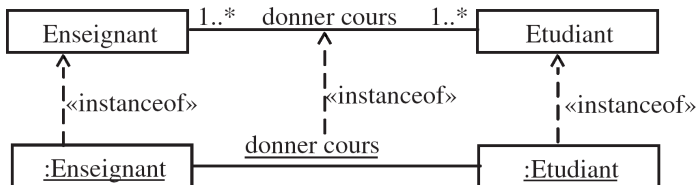
- Un **lien** est une instance d'une association.
- Un lien se représente comme une association mais s'il a un nom, il est souligné.

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.



Plan

1 Introduction à la Modélisation Orientée Objet

2 Modélisation objet élémentaire avec UML

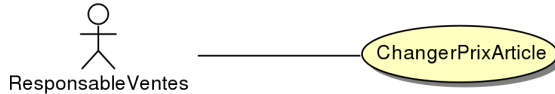
- Diagrammes de cas d'utilisation
- Diagrammes de classes
- Diagrammes d'objets
- Diagrammes de séquences

Objectif des diagrammes de séquence

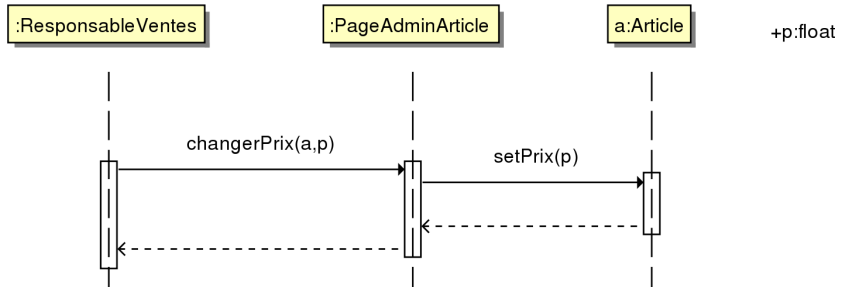
- Les **diagrammes de cas d'utilisation** modélisent à **QUOI** sert le système, en organisant les interactions possibles avec les acteurs.
- Les **diagrammes de classes** permettent de spécifier la structure et les liens entre les objets dont le système est composé : ils spécifie **QUI** sera à l'oeuvre 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'échangeant des messages.
 - Les acteurs interagissent avec le système au moyen d'IHM (Interfaces Homme-Machine).

Exemple d'interaction

- Cas d'utilisation :

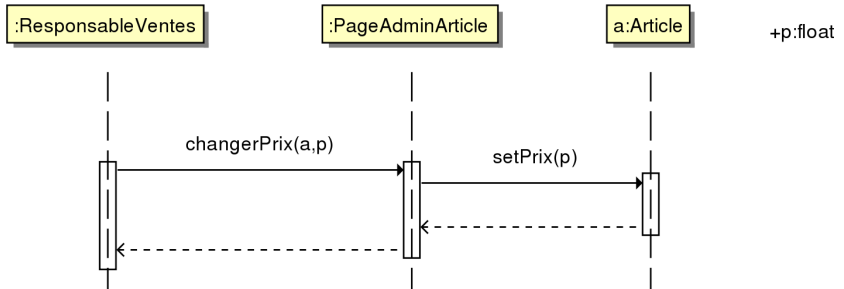


- Diagramme de séquences correspondant :

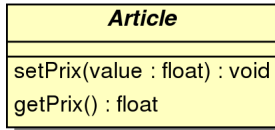


Exemple d'interaction

- Diagramme de séquences correspondant :



- Opérations nécessaires dans le diagramme de classes :

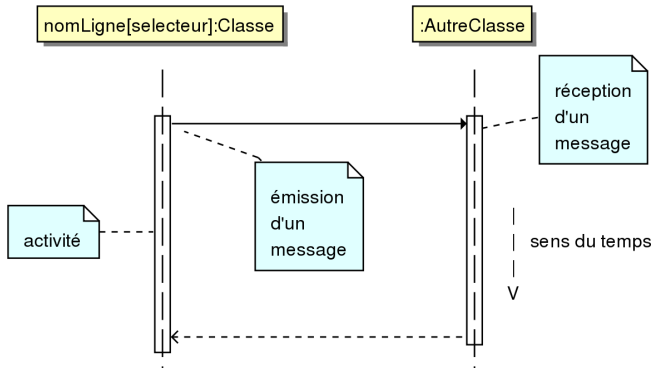


Ligne de vie

- 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

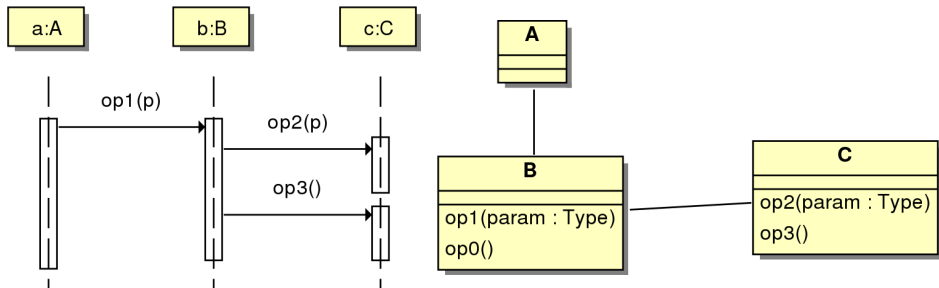
- Un message **synchrone** bloque l'expéditeur jusqu'à la réponse du destinataire. Le flot de contrôle passe de l'émetteur au récepteur.
 - Typiquement : appel de méthode
 - Si un objet A invoque une méthode d'un objet B, A reste bloqué tant que B n'a pas terminé.
- On peut associer aux messages d'appel 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 »).



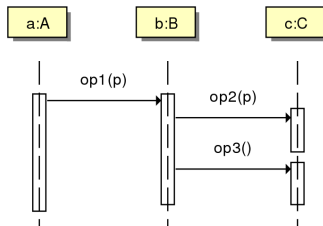
Correspondance messages / opérations

- 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();
    }
}
  
```

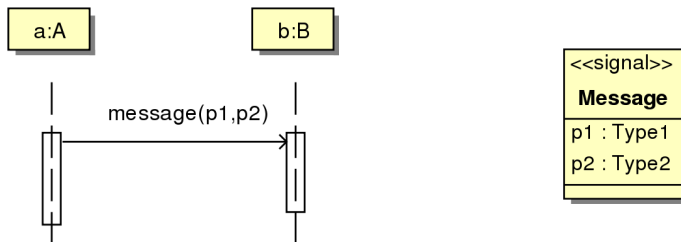
```

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


Correspondance messages / signaux

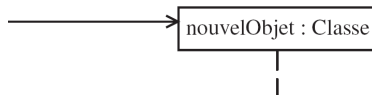
- Les **messages asynchrones** correspondent à des **signaux** dans le diagramme de classes.

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

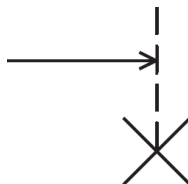


Création et destruction de lignes de vie

- 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 :

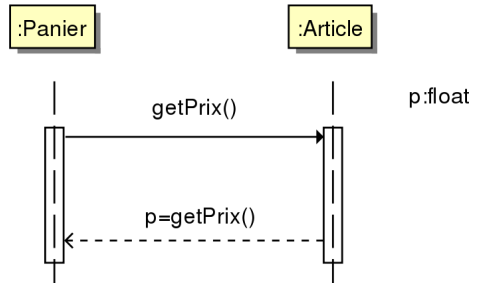
```
nomParametre:valeurParametre
```

- **Exemples :**

- appeler("Capitaine Hadock", 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.

Syntaxe des messages de retour

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

`attributCible=nomOperation(params):valeurRetour`

- La **syntaxe des paramètres** est :

`nomParam=valeurParam`

ou

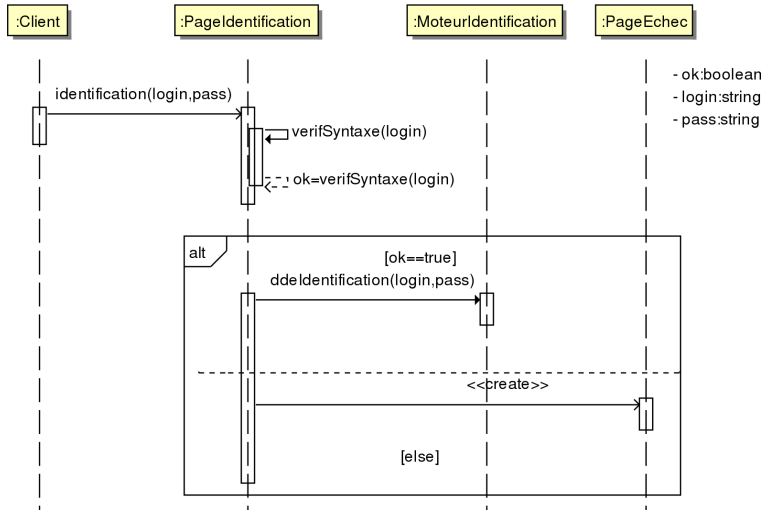
`nomParam:valeurParam`

- Les messages de retour sont représentés en pointillés.

Fragment combiné

- 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** »).

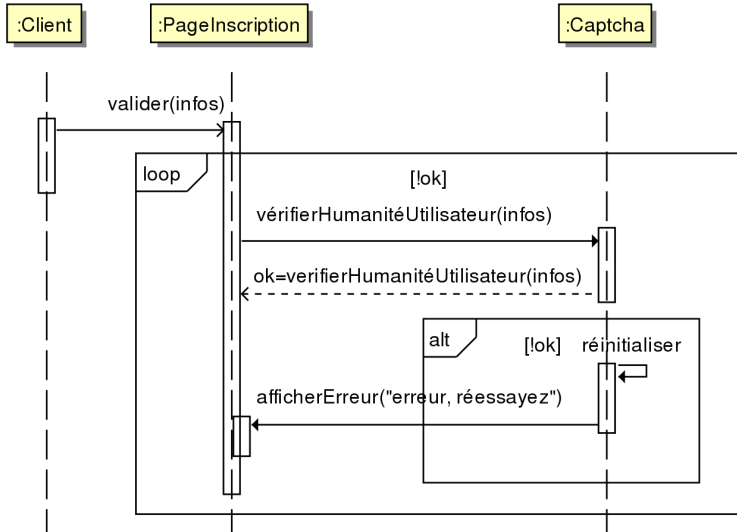
Exemple de fragment avec l'opérateur conditionnel



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



Syntaxe de l'opérateur loop

- **Syntaxe d'une boucle :**

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

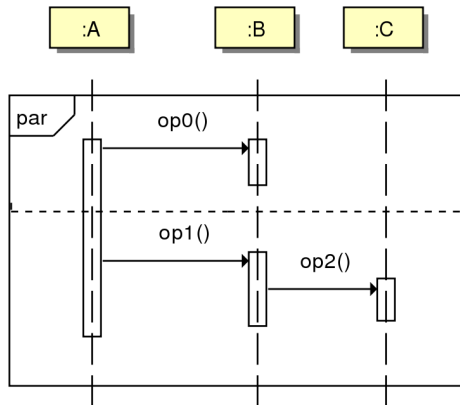
- La boucle est répétée au moins minNbItérations fois avant qu'une éventuelle condition booléenne ne soit testée (la condition est placée entre crochets dans le fragment)
- Tant que la condition est vraie, la boucle continue, au plus maxNbItérations fois.

- **Notations :**

- `loop (valeur)` est équivalent à `loop (valeur, valeur)`.
- `loop` est équivalent à `loop (0, *)`, où `*` signifie « illimité ».

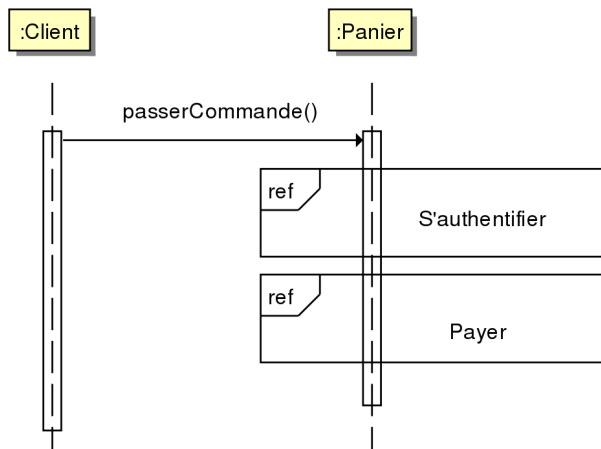
Opérateur parallèle

- 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

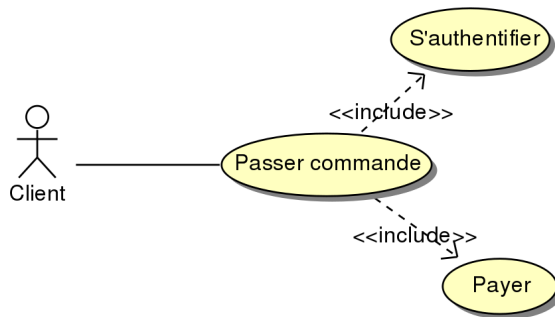
- **Réutiliser une interaction** consiste à placer un fragment portant la référence « ref » là où l'interaction est utile.



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

Utilisation d'un DS pour modéliser un cas d'utilisation

- 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

Utilisation d'un DS pour spécifier une méthode

- Une interaction peut être identifiée par un fragment **sd** (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

