



# UML 2

version support : 1.3

## Objectifs

- Connaitre la notation UML
  - qui est simple et intuitive
- Montrer comment à partir de la notation UML il est possible
  - de définir des modèles
    - plus ou moins abstraits
    - plus ou moins élaborés
    - décrivant plusieurs vues complémentaires d'un système

# Introduction

- Des modèles plutôt que du code
  - un modèle est la simplification (abstraction ?) d'un aspect de la réalité
  - la construction de modèles nous aide à mieux comprendre les systèmes que nous développons
  - la modélisation nous aide à comprendre des systèmes que nous sommes incapables de comprendre dans leur réalité
    - modélisation du "monde réel"
- NOUS DEVELOPPONS POUR DES UTILISATEURS

# Introduction

- Contexte historique
  - issu de la fusion de trois méthodes de modélisation
    - la méthode Booch développée par Grady Booch
    - OMT (Object Modeling Technic) de James Rumbaugh
    - la méthode Objectory conçue par Ivar Jacobson
  - première version introduite en 1994
  - prise en compte par l'OMG (Object Management Group) en 1997 sous le nom de UML v1.1
  - la spécification UML évolue pour résoudre les problèmes ou défauts identifiés dans les versions antérieures

# Introduction

- UML est un langage graphique
  - il est donc assorti d'une syntaxe et d'une sémantique
- UML n'est pas un processus logiciel
  - le langage UML peut être utilisé au sein d'une méthode de développement
- UML est conçu pour être utilisé par des outils de conception automatique
  - offre encore balbutiante
  - se prête très bien à la réalisation de dessins et croquis sommaires

# Introduction

- UML 2 se compose d'un ensemble de spécifications émises par l'OMG
  - la spécification d'échange de diagramme
    - permet l'échange des modèles entre outils
  - la spécification d'infrastructure d'UML
    - méta modèle pour la construction du langage
  - la spécification de superstructure d'UML
    - définit la définition formelle des éléments UML
    - utilisée par les fournisseurs d'outils
  - la spécification OCL (Object Constraint Language)
    - langage simple de rédaction des contraintes et expressions

# Introduction

- UML 2 permet une représentation graphique d'un système
  - c'est une vue nécessaire pour la documentation ou la construction du système
- UML 2 propose 13 diagrammes différents
  - il y en avait 9 dans la version 1
  - l'utilisation de tous les digrammes n'est pas requise
  - il faut utiliser le diagramme le plus adapté pour la modélisation du concept
    - en tenant compte aussi du public qui va consulter les documents UML

# Introduction

- UML peut être étendu
  - de manière informelle en utilisant les stéréotypes, les contraintes, les notes...
  - de manière formelle en utilisant les profils UML
    - collection de stéréotypes associé à des définitions UML génériques
      - profils CORBA, EAI (Enterprise Application Integration)

## Introduction – les diagrammes

- Deux familles de diagrammes sont proposés par UML 2
  - les diagrammes structurels
    - ils décrivent une organisation physique des objets dans le système à modéliser
    - ce sont des vues statiques du système
  - les diagrammes comportementaux
    - ils décrivent le comportement des éléments du systèmes
    - ce sont des vues dynamiques du système

## Introduction - les diagrammes

- Diagrammes structurels
  - diagramme de classes
    - décrit les classes et interfaces composant le système à modéliser, ainsi que leurs relations statiques
  - diagramme de composants
    - composants physiques du système
  - diagramme de structure composite (UML 2)
    - permet de décrire des éléments interconnectés
  - diagramme de déploiement
    - décrit les modalités de déploiement et d'installation

# Introduction – les diagrammes

- Diagrammes structurels
  - diagramme de paquetages (UML 2)
    - montre comment les classes et interfaces sont regroupées
    - évolution du diagramme de composants UML 1.x
  - diagramme d'objets
    - montre une vue statique des instances de classe

# Introduction – les diagrammes

- Diagrammes comportementaux
  - diagramme d'activité
    - diagramme de flux
  - diagramme de communication
    - ancien diagramme de collaboration UML 1.x
  - diagramme d'interaction d'ensemble (UML 2)
    - vue simplifiée du diagramme d'activité mettant en évidence les éléments intervenant dans la réalisation d'une activité, sans entrer dans les détails
  - diagramme de séquences
    - décrit les messages échangés entre les composants du système

# Introduction – les diagrammes

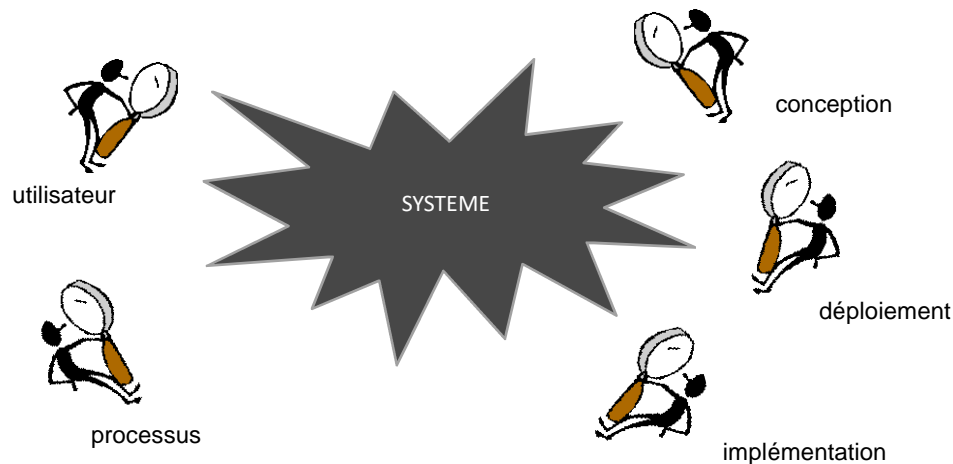
- Diagrammes comportementaux
  - diagramme de machines d'état
    - mise en évidence des transitions d'état interne à un élément du système
  - diagramme de chronométrage (UML 2)
    - pose des contraintes temporelles sur les échanges de message
    - orienté "temps réel"
  - diagramme des cas d'utilisation
    - expression des besoins fonctionnels du système

# Introduction – les diagrammes

- Les diagrammes d'interactions sont une famille de diagrammes définis par UML 2
  - ils regroupent
    - les diagrammes de communication
    - les diagrammes d'interaction d'ensemble
    - les diagrammes de séquences
    - les diagrammes de chronométrage

# Introduction – les vues du système

- Il est courant de voir le système à modéliser au moyen de vues différentes
  - ce concept ne fait pas partie de la spécification UML



# Introduction – les vues du système

- La vue du concepteur
  - description de de la manière dont le système sera implémenté et exécuté
    - définition des classes, interfaces, structures, ...
  - diagrammes de classes, d'objets, activité, séquences
- La vue de déploiement
  - description de la façon dont le système sera installé, configuré et exécuté
  - diagrammes de déploiement, d'interactions



## Introduction – les vues du système

- La vue d'implémentation
  - définition des composants, fichiers et ressources utilisés par le système
  - diagrammes de composants, d'interactions
- La vue du processus
  - formalisation de la cohérence, performance et évolutivité du système
  - diagrammes d'interaction, d'activité

## Introduction – les vues du système

- La vue de l'utilisateur
  - définition des fonctionnalités requises par l'utilisateur du système
  - diagrammes des cas d'utilisation
- Ces vues ne sont là que pour guider dans le choix des diagrammes.

# Syntaxe

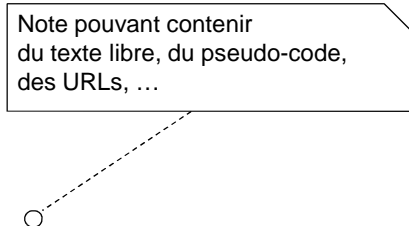
- UML est un langage graphique
  - simple à utiliser dans un cadre informel : papier, tableau, ...
    - mais pas de formalisation des détails
  - conçu pour être utilisé avec des outils
    - niveau de performance et de maturité des outils très différents
    - permet d'aller loin dans le détail
      - doit aller jusqu'à la génération du code
    - gestion de la synchronisation des diagrammes et du code
      - rétro-ingénierie

# Spécifications

- Les spécifications UML sont maintenues par l'OMG
  - [www.uml.org](http://www.uml.org)
  - spécifications du méta-modèles (600 pages), des schémas XML, d'OCL, certains profils...
- Une spécification XML permet l'échange des modèles entre outils
  - XMI (XML Metadata Interchange)

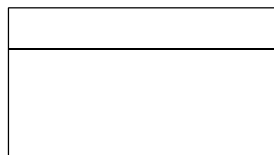
# Méta-modèle

- Les notes
  - utilisées comme "commentaire" dans les diagrammes



# Méta-modèle

- Les classificateurs
  - élément de base du méta-modèle
  - groupe d'objet possédant des propriétés communes
    - les classes par exemple, dans la superstructure UML, qui possèdent un nom, des attributs et des opérations
  - notation : rectangle pouvant être divisé avec des compartiments

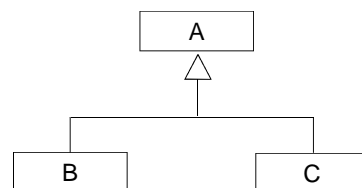


# Méta-modèle

- Décoration
  - information supplémentaire pour un classificateur
    - restriction sur des valeurs
    - contraintes
      - {ordered, unique}
    - stéréotypes
      - donne au lecteur une information particulière
      - certains outils de conceptions ajoutent du comportement particulier avec les stéréotypes
      - souvent associé à des concepts d'implémentation
        - <<interface>>, <<singleton>>

# Méta - modèle

- Généralisation
  - permet de factoriser des éléments communs à des classificateurs
  - le classificateur A factorise des éléments des classificateurs B et C



## Bonnes pratiques

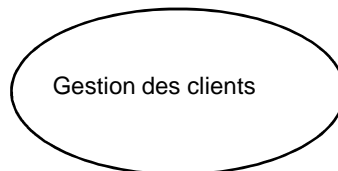
- Aucun diagramme n'est obligatoire
  - utiliser seulement les graphiques et symboles qui permettent le clarifier le problème
  - utiliser les diagrammes qui "parlent" le mieux aux différents interlocuteurs
- Un modèle UML est rarement complet
  - sauf si vous utilisez des outils
- Donnez le bon niveau de détail
  - rien ne sert de faire figurer toutes les classes d'un framework, ou tous les membres d'une classe
    - les outils permettent de cacher les détails

## Diagramme de cas d'utilisation

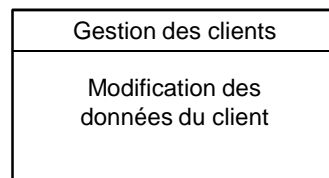
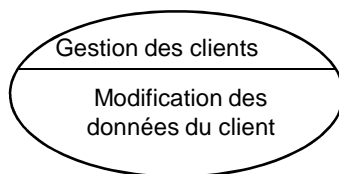
- Identification des fonctionnalités d'un système
  - vue des utilisateurs (acteurs) d'un système
  - indépendant de l'implémentation
- Met en avant
  - des éléments fonctionnels : les cas d'utilisations qui sont identifiés par un nom
  - des acteurs (roles) qui interagissent avec les cas d'utilisation
    - humain ou système

# Diagramme de cas d'utilisation

- Syntaxe du cas d'utilisation
  - ellipse contenant le nom du cas d'utilisation

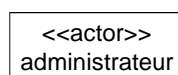
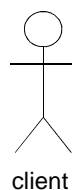


- ou utilisation d'un classificateur



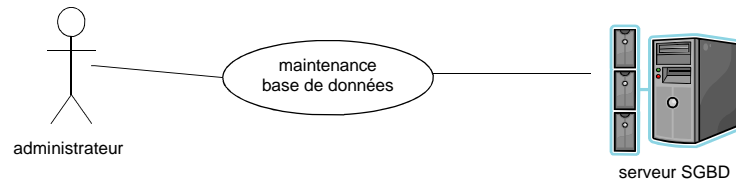
# Diagramme de cas d'utilisation

- Syntaxe de l'acteur
  - l'acteur interagit avec les cas d'utilisation
    - il est formaliser sous la forme d'un bonhomme fil de fer, puls rarement sous forme d'un classificateur
      - les outils peuvent proposer d'autres icônes

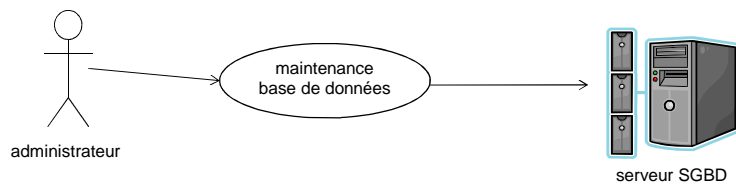


# Diagramme de cas d'utilisation

- Association entre acteurs et cas d'utilisation

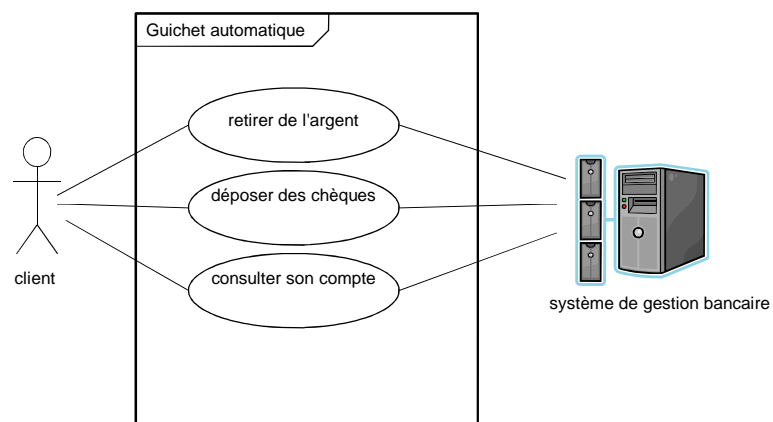


- les associations peuvent être orientées (non UML)



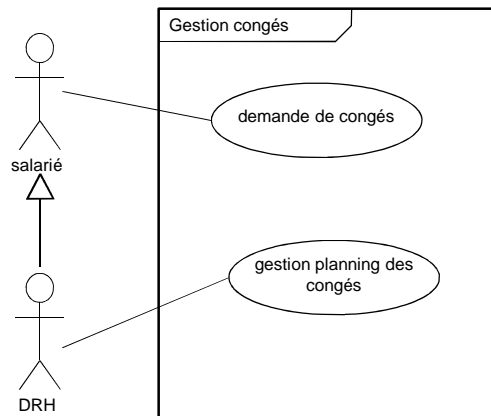
# Diagramme de cas d'utilisation

- Frontières du système
  - limite la modélisation par rapport au mode extérieur



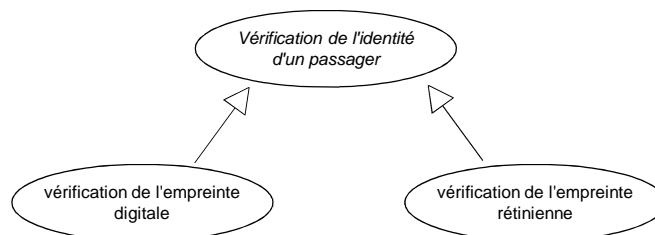
# Diagramme de cas d'utilisation

- Généralisation des acteurs
  - la généralisation des acteurs permet de rendre le diagramme plus simple
    - pas dans les spécifications UML



# Diagramme de cas d'utilisation

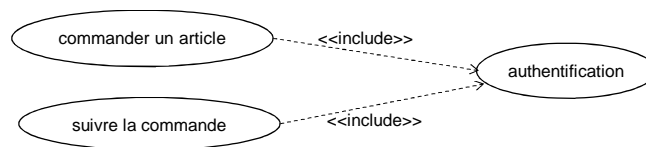
- Généralisation des cas d'utilisation
  - le cas d'utilisation général est abstrait
    - titre en italique
    - ne précise pas comment le cas d'utilisation est mis en œuvre
  - les cas d'utilisation spécialisés sont autonomes





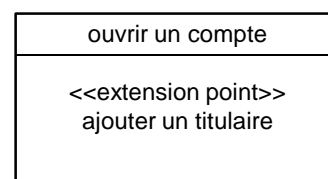
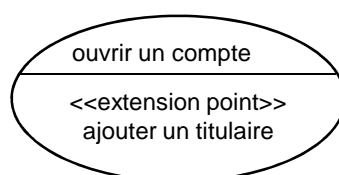
# Diagramme de cas d'utilisation

- Inclusion de cas d'utilisation
  - regroupement de fonctionnalités communes à plusieurs cas d'utilisation
  - le cas d'utilisation inclus est partagé par d'autres cas d'utilisation
  - le cas d'utilisation inclus n'est pas autonome
  - la fonctionnalité incluse n'est pas optionnelle



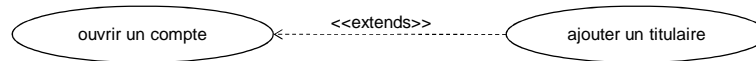
# Diagramme de cas d'utilisation

- Extension de cas d'utilisation
  - extension de fonctionnalités supplémentaires à un cas d'utilisation de base
    - fonctionnalités supplémentaires non pertinentes en dehors du contexte du cas de base
  - le cas d'utilisation de base est complet et autonome

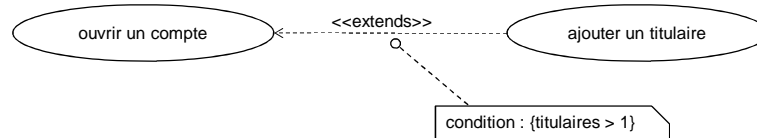


# Diagramme de cas d'utilisation

- Extension de cas d'utilisation



- on peut y ajouter des conditions pour préciser l'exécution de l'extension



# Diagramme de cas d'utilisation

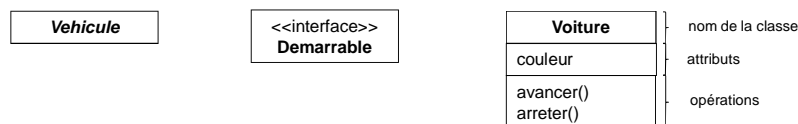
- Un cas d'utilisation est associé à une description textuelle

- véritable cahier des charges du cas d'utilisation
- utilise texte, images, maquette, etc...
- reprend en général
  - le nom du cas d'utilisation
  - les acteurs utilisant le cas d'utilisation
  - une description des fonctionnalités
    - précondition
    - post-condition
    - chemin d'erreur
    - ...

# Diagramme de classes

- Syntaxe

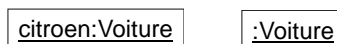
- la classe est symbolisée par un classificateur
  - contient au minimum le nom (titre) de la classe et des compartiments supplémentaire pour les attributs et opérations
  - même conventions de nommage que Java
  - nom en gras
    - italique si abstraite
    - avec stéréotype <<interface>> si interface au sens Java, C#



# Diagramme de classes

- Instances de classe

- peuvent être anonyme ou nommées
  - ne sont pas utilisées dans le diagramme de classe
  - pas de compartiment supplémentaire



- Attributs

- correspondent aux propriétés
- peuvent être définis en ligne ou par relation entre les classes

- Opérations

- méthodes de la classe

# Diagramme de classes

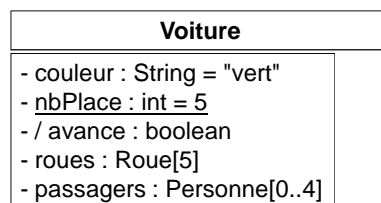
- Attributs en ligne

- syntaxe :

visibilité / nom : type [multiplicité] = défaut {propriétés et contraintes}

- seul le nom est obligatoire

- le niveau de détail aidera le développeur
    - ou est imposé par l'outil
      - les détails peuvent alors être masqués



# Diagramme de classes

- Attribut en ligne : éléments syntaxiques

- visibilité

- publique : +
    - privée : -
    - protégée : #
    - paquetage : ~
    - par défaut : rien

- / attribut dérivé

- peut être calculé à partir d'autres attributs

- nom

- nom de l'attribut – obligatoire

- souligné

- attribut statique

# Diagramme de classes

- Attribut en ligne : éléments syntaxiques

- type

- type de l'attribut
      - pseudo type, ou un type lié au langage

- multiplicité

- par défaut 1
    - peut-être un nombre, un intervalle [2..3]
      - borne infinie : \*

- défaut : valeur par défaut

- propriété et contraintes (tags)

- collections de tags pouvant être attaché à l'attribut
      - clients : Client[1..\*] {unique,ordered}

# Diagramme de classes

- Multiplicité d'un attribut

- collections UML

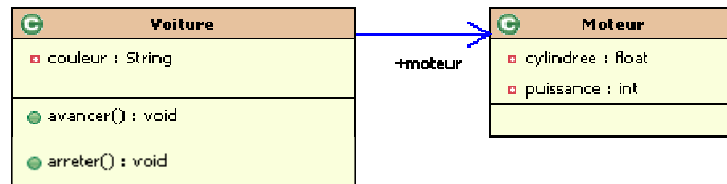
ordre	unicité	type UML
faux	faux	Bag
vrai	vrai	OrderedSet
faux	vrai	Set
vrai	faux	Sequence

- les outils adaptent les collections aux types proposés par le langage

- généralement paramétrable
    - attention aux méthodes ajoutées automatiquement
      - accesseurs, itérateurs, ...
      - généralement aussi paramétrable

# Diagramme de classes

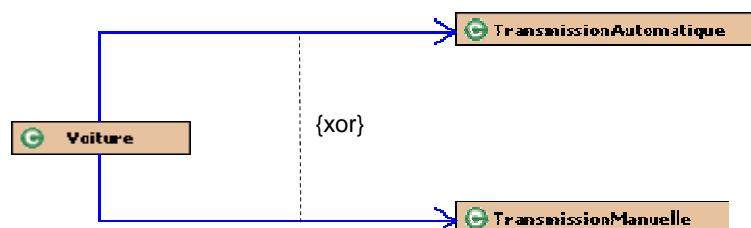
- Attributs par relation
  - les diagrammes sont plus détaillés
    - mais plus volumineux



- le nom de l'attribut n'apparaît plus alors dans la classe mais dans la relation

# Diagramme de classes

- Attributs par relation
  - utilisation de contraintes



- selon les langages l'implémentation pourra se faire sous forme de famille de classes et d'une fabrique
- les relations entre classe sont détaillées plus loin

# Diagramme de classes

- Les opérations
  - en UML l'opération est le moyen d'invoquer une fonctionnalité sur la classe
  - la méthode est l'implémentation concrète de cette fonctionnalité
  - les opérations sont placées dans un compartiment séparé
  - les opérations abstraites sont indiquées en italique
  - les opérations statiques sont sous-lignées

# Diagramme de classes

- Opérations : éléments syntaxiques

visibilité nom (paramètres) : type-retourné {propriétés}

- visibilité : +, -, # ou ~
- nom : description succincte de l'opération
  - selon les outils peut être différent du nom de la méthode
- type-retourné : type du résultat retourné
  - optionnel, si non indiqué ne signifie par qu'il n'y a pas de retour
- propriétés : contraintes pouvant exister entre {}
  - coutPanier() : float {precondition : panier.articles.nombre > 0}

# Diagramme de classes

- Opérations : éléments syntaxique des paramètres

direction nom : type [multiplicité] = défaut {propriétés}

- même syntaxe que pour les attributs
- direction peut prendre les valeurs
  - in : paramètre en entrée fournit par la procédure appelante, non modifiable
  - out : paramètre non fournit par la procédure appelante, calculé par l'opération et utilisable par la procédure appelante
  - inout : paramètre fournit par la procédure appelante et modifié par l'opération
  - return : paramètre contenant une valeur retournée en fin d'exécution

# Diagramme de classes

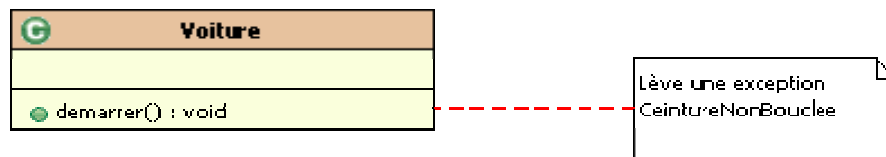
- Contraintes sur une opération
  - les contraintes établissent un contrat qui doit être respecter à l'implémentation
  - pré condition (contrainte a priori)
    - mot clé : precondition
    - identification des valeurs licites pour les paramètres
    - UML précise que les contraintes à priori ne doivent pas être vérifiée par l'opération. L'opération ne doit pas être appelée si la contrainte n'est pas vérifiée
  - post condition (contrainte a postérieur)
    - mot clé : postcondition
    - défini l'état du système une fois l'opération exécutée



# Diagramme de classes

- Exceptions

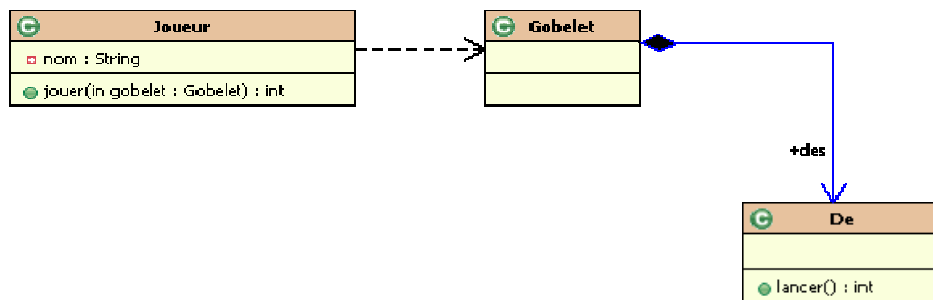
- les exceptions susceptibles d'être levées par une opération sont représentées comme une contrainte



# Diagramme de classes - relations

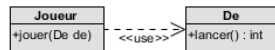
- Dépendance

- la forme la plus faible de relation
- se traduit par "... utilise ..."
- syntaxe : flèche pointillée →

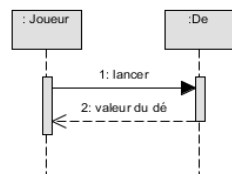


# Diagramme de classe - relations

- Dépendance
  - diagramme de classes



- diagramme de séquences



# Diagramme de classes - relations

- Dépendance
  - codes sources

```
class Joueur
{
public:
    Joueur(void);
    ~Joueur(void);
    void jouer(De de);
};
```

Joueur.hpp

```
#include "Joueur.hpp"

Joueur::Joueur(void){}

Joueur::~Joueur(void){}

void Joueur::jouer(De de){de.lancer();}
```

Joueur.cpp

# Diagramme de classes - relations


- Associations

- les associations sont une relation entre classe plus forte que les dépendances

- peut s'interpréter comme " ... a un ..."

- différent de "... est propriétaire de ..."

- » par exemple la fenêtre a un curseur mais le curseur est partagé par toutes les fenêtres

- syntaxe : flèche trait plein 
    - le nom de la propriété peut être précisée
    - la multiplicité peut être indiquée



# Diagramme de classes - relations

- Dépendance entre classe

- codes sources

```
class Monopoly
{
private:
    vector<Joueur> joueurs;
public:
    Monopoly(void);
    ~Monopoly(void);
    void add(Joueur &joueur);
};
```

Monopoly.hpp

```
#include "Monopoly.hpp"

Monopoly::Monopoly(void){}

Monopoly::~~Monopoly(void){}

void Monopoly::add(Joueur &joueur){
    joueurs.push_back(joueur);
}
```

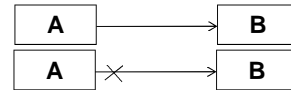
Monopoly.cpp

# Diagramme de classes - relations

- Navigabilité

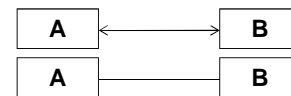
- unidirectionnelle

- une croix peut indiquer le manque de navigabilité



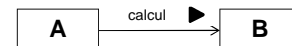
- bidirectionnelle

- indiquée par deux flèches, ou aucune



- nom de l'association et décoration

- permet de mieux comprendre l'association



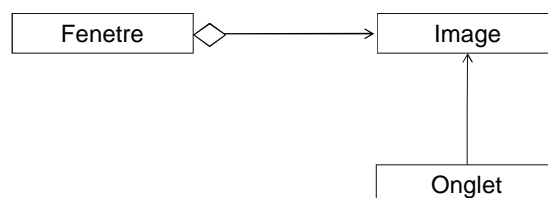
# Diagramme de classes - relations

- Agrégation

- relations plus forte que l'association

- peut s'interpréter comme " ... est propriétaire de ... "
      - dans l'exemple la fenêtre est propriétaire de l'image, mais l'image peut être partagée par d'autres instances de classe

- syntaxe :  →



# Diagrammes de classes - relations

- Agrégation
  - codes sources

```
class Monopoly
{
private:
    vector<Joueur *> joueurs;
public:
    Monopoly(vector<Joueur *> joueurs);
    ~Monopoly(void);
};
```

Monopoly.hpp

```
#include "Monopoly.hpp"


using namespace std;

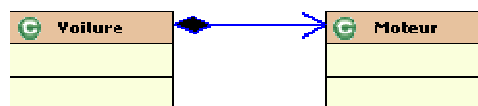
Monopoly::Monopoly(vector<Joueur *>
joueurs):joueurs(joueurs){}

Monopoly::~Monopoly(void){}
```

Monopoly.cpp

# Diagrammes de classes - relations

- Composition
  - c'est la relation la plus forte
    - possession totale
  - peut s'interpréter comme "... fait partie de ..."
  - syntaxe : 



# Diagramme de classes - relations

- Composition
  - codes sources

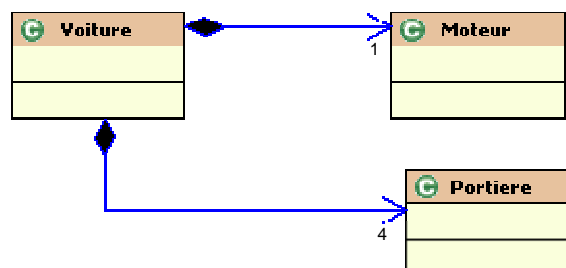
```
Monopoly.hpp
class Monopoly
{
private:
    vector<Joueur *> joueurs;
public:
    Monopoly(int nbJoueurs);
    ~Monopoly(void);
};

Monopoly.cpp
Monopoly::Monopoly(int nbJoueurs)
{
    for(int i=0 ; i<nbJoueurs ; i++)
        joueurs.push_back(new Joueur());
}

Monopoly::~~Monopoly(void){
    for(int i=0 ; i<joueurs.size() ; i++)
        delete joueurs[i];
}
```

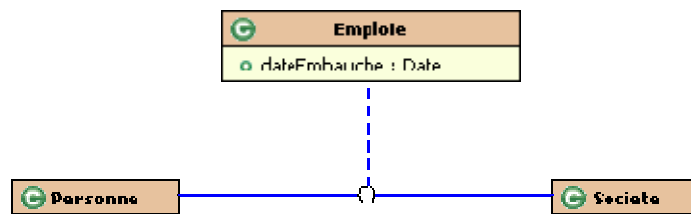
# Diagramme de classes - relations

- Cardinalités
  - les cardinalités peuvent être indiquées sur les relations
    - précise le nombre d'instances qui participent à la relation
    - syntaxe :
      - nombre : 1
      - intervalle : 0..3
      - de 0 à l'infini : \*



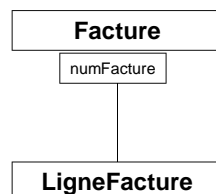
## Diagramme de classes - relations

- Classes d'association
  - une association peut posséder des propriétés qui se traduira par une classe
  - syntaxe : la classe d'association est relié à l'association par une ligne pointillée



## Diagramme de classes - relations

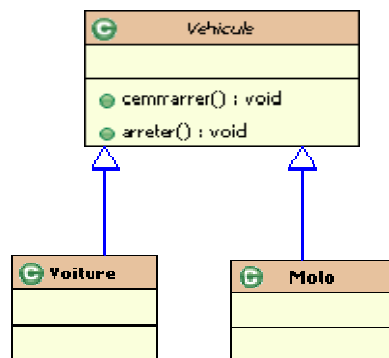
- Qualificateur d'association
  - la relation entre deux éléments peut faire l'objet d'une indexation
    - la clé est alors une valeur particulière



# Diagramme de classe - généralisation

- Généralisation

- la relation de généralisation ne reçoit pas de décoration
  - pas de nom, pas multiplicité,...



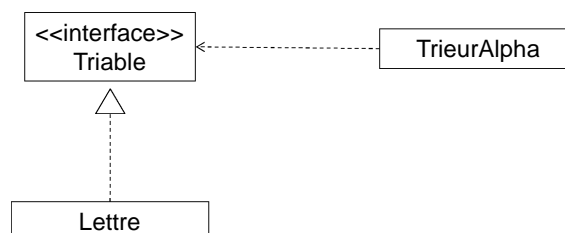
# Diagramme de classes – les interfaces

- Représentation par stéréotype



- Utilisation de l'interface

- implémentation et utilisation





# Diagramme de classes – les interfaces

- Utilisation des interfaces
  - implémentation et utilisation par la notation "rotule"
  - utilisée pour les interfaces de framework, ou librairies tiers
  - diagramme allégé



# Diagramme de classes

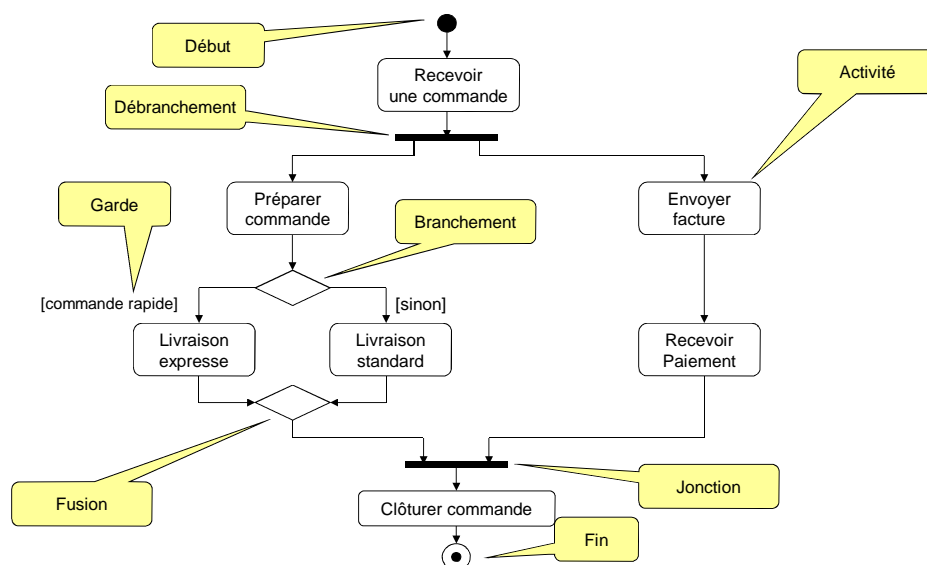
- Modèle de classe
  - abstraction pour le type de classe avec lequel une classe donnée peut interagir
  - classes paramétrées



# Diagramme d'activité

- Décrit l'organisation des activités
  - comportements conditionnels et parallèles
- Comportement conditionnel décrit par des branchements et des fusions
  - branchement : une transition entrante, plusieurs sortantes, dont une seule peut-être empruntée
  - fusion : plusieurs transitions entrantes, une seule sortante
- Comportement parallèle décrit par des débranchements et des jonctions
  - débranchement : une transition entrante et plusieurs sortantes empruntées en parallèle
  - jonction : plusieurs transitions entrantes, une seule sortante

# Diagramme d'activité

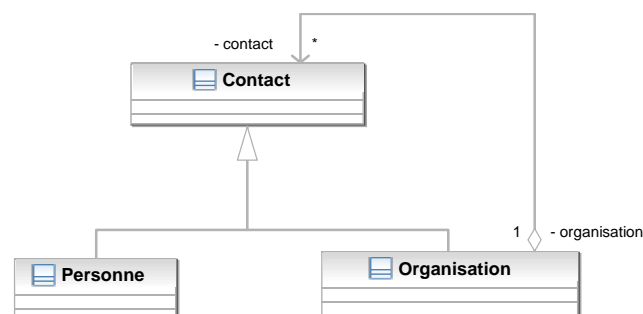


# Diagramme d'objets

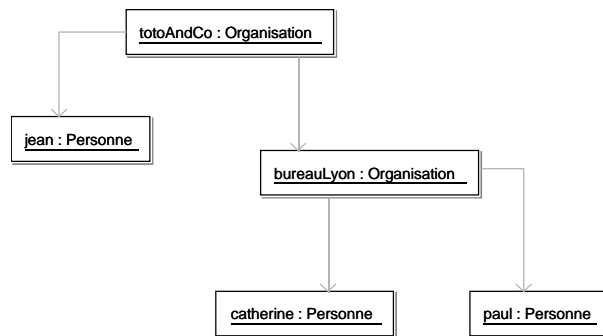
- C'est un instantané d'une partie des objets présents dans le système à un moment donné
  - appelé aussi diagrammes d'instances
- Employé pour représenter une configuration particulière d'objets
- Permet de mieux comprendre certains diagrammes de classes complexes

# Diagramme d'objets

- Exemple :
  - diagramme de classes des entrées d'un répertoire



# Diagramme d'objets

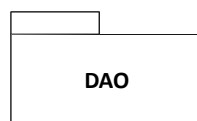


# Diagramme de paquetage

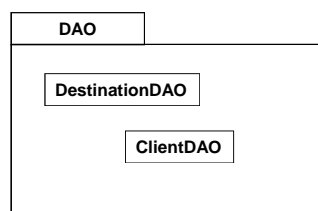
- Permet de regrouper des éléments au sein d'un même espace de nommage

— syntaxe :

- paquetage seul

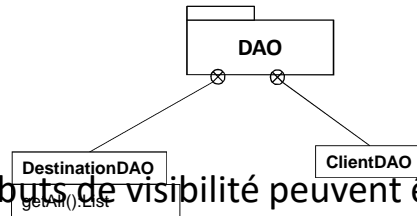


- paquetage avec ses classes

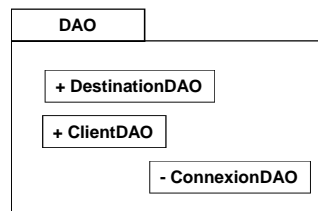


# Diagramme de paquetage

- Autre représentation permettant de détailler les classes



- Les attributs de visibilité peuvent être placés devant les classes



# Diagramme de paquetage

## • Import d'un paquetage

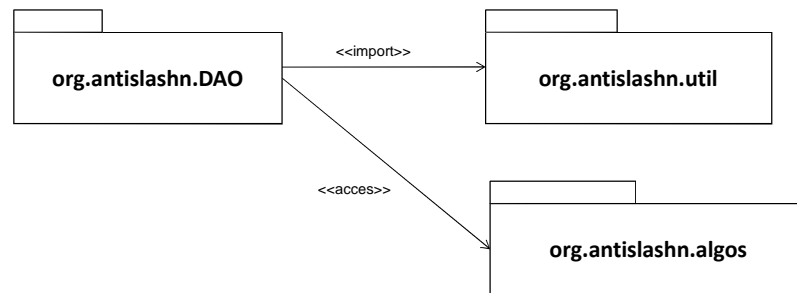
- `org.antislashn.DAO` doit utiliser des paquetages, il est possible de les importer



- par défaut les éléments importer reçoivent une visibilité public
- l'implémentation concrète de l'import (et de l'accès) varie énormément en fonction des langages.

## Diagramme de paquetages

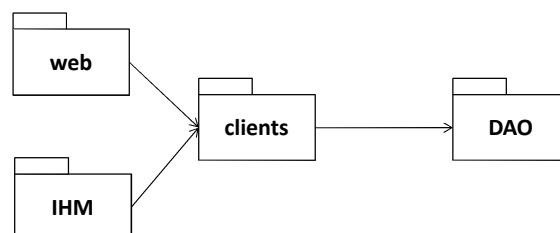
- Accès à un paquetage
  - pour spécifier que les éléments importés doivent avoir une visibilité `private`, il faut spécifier `access`



- si un paquetage importe `org.antislashn.DAO`, il pourra accéder à `org.antislashn.util`, mais pas à `org.antislashn.algos`

## Diagramme de paquetage

- Structuration des projets à l'aide du diagramme de paquetage
  - les diagrammes de paquetages permettent de structurer la logique de l'organisation d'un système
  - les dépendances entre paquetages permettent de visualiser l'impact des évolutions du système

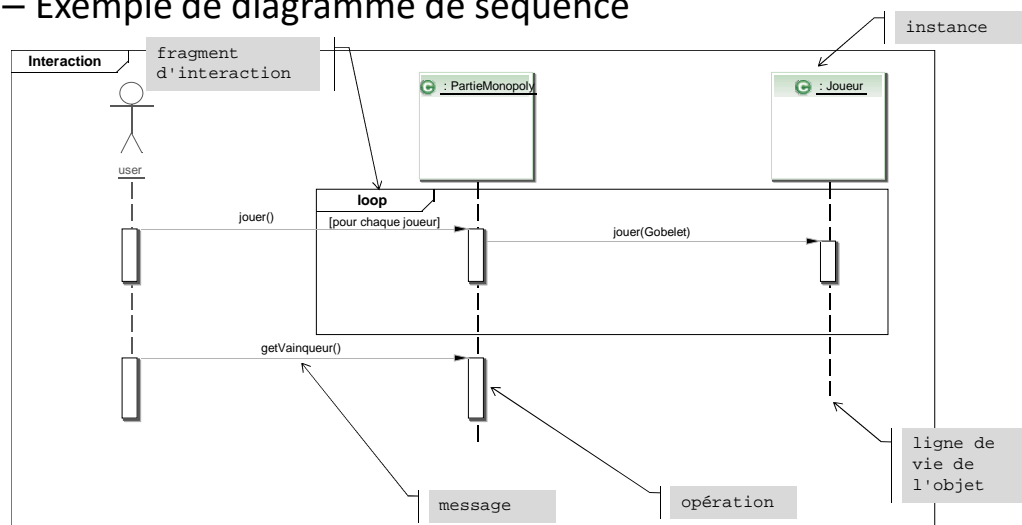


# Diagrammes d'interaction

- Les diagrammes d'interaction décrivent les communications entre les objets d'une application
  - diagramme de séquence
    - le plus utilisé
  - diagramme de communication
  - diagramme d'interaction d'ensemble
  - diagramme de chronométrage

## Diagramme d'interaction

### – Exemple de diagramme de séquence



## Diagramme de séquence - acteurs

- Les acteurs d'une interaction
  - rectangle : instance de l'objet
  - ligne pointillée : ligne de vie de l'objet
  - nom de l'acteur sous la forme  
`nom_objet [sélecteur] : nom_classe [ref décomposition]`
    - `nom_objet` : nom de l'instance, si absent : instance anonyme
    - `sélecteur` : si nécessaire, représente l'indice de l'instance dans une collection
    - `nom_classe` : nom du type (obligatoire, toujours préfixé par ":" )
    - `ref décomposition` : permet de faire référence (ref) à un autre diagramme d'interaction ou est engagé le même acteur

## Diagramme de séquence - acteurs

- L'acteur qui initie le premier diagramme de séquence est souvent représenté par un bonhomme fil de fer
  - le premier diagramme de séquence correspond à un use case
- UML définit un nom d'acteur réservé : `self`
- Les créations et destructions d'objets peuvent être représentés

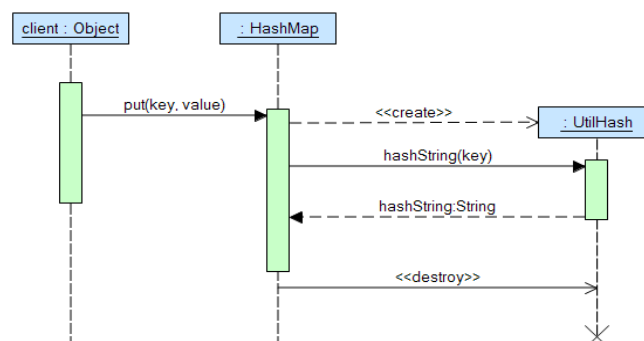


# Diagramme de séquence

- Le message d'interaction se focalise sur la communication entre lignes de vie
  - envoi d'un signal
  - création d'une instance
  - destruction d'une instance
- Un message spécifie
  - le type de communication
  - l'émetteur
  - le destinataire
- L'utilisation la plus courante des messages est la symbolisation des appels de méthode

# Diagramme de séquence

- Création et destruction



# Diagramme de séquence

- Syntaxe d'un message

`attribut = nom_signal (arguments) : valeur_retournée`

- attribut : indique que la valeur retournée est stockée dans l'attribut nécessaire
- nom\_signal : le nom de l'opération à invoquer
- arguments : liste des arguments séparés par des virgules
  - possibilité de préfixer le nom de l'argument par in, out, inout
- valeur\_retournée : indique explicitement la valeur retournée par le message

# Diagramme de séquence

- Représentation du message

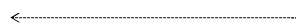
- une flèche représente un message asynchrone



- une flèche pleine représente un message synchrone

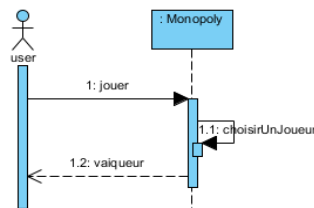


- le retour de l'opération peut-être représenté par une flèche dont la ligne est en pointillée



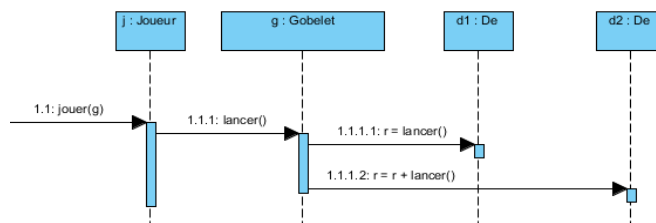
# Diagramme de séquence

- Occurrence d'exécution
  - désigne l'objet impliqué dans l'exécution d'un type d'action pendant une durée notable
    - typiquement l'appel d'une méthode
  - l'occurrence est représentée par un rectangle placé sur la ligne de vie de l'objet



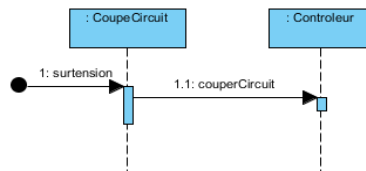
# Diagramme de séquence

- Les variables locales peuvent figurer dans le diagramme
  - elles peuvent être déclarées dans le diagramme au moyen de la syntaxe employée pour les attributs de classe

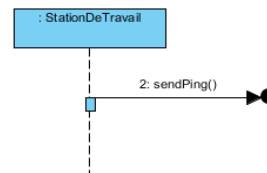


# Diagramme de séquence

- Un message trouvé est un message reçu par un objet dont l'émetteur est inconnu

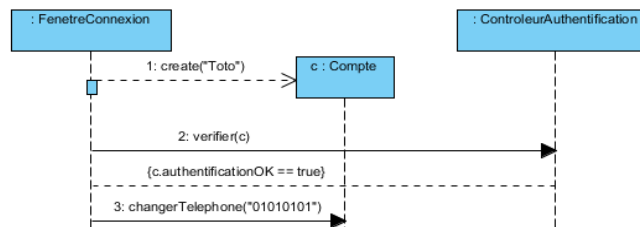


- Un message perdu est un message qui a été envoyé, mais qui n'a jamais atteint son destinataire



# Diagramme de séquence

- Invariant d'état
  - condition qui doit restée vraie pour que le reste d'une interaction soit valide
  - représentation sous forme de contrainte, de note ou dans un symbole d'état (rectangle aux coins arrondis)

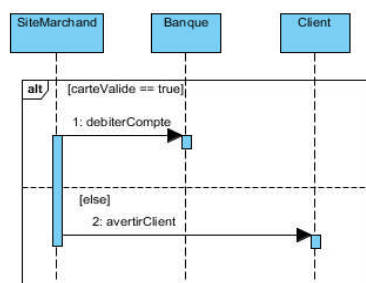


# Fragments combinés

- Un fragment combiné est constitué
  - d'un opérateur d'interaction
  - un ou plusieurs fragments d'interaction
    - qui constituent les opérandes de l'interaction
- Un fragment d'interaction peut posséder une condition de garde indiquant si le fragment est valide ou non
  - syntaxe : condition entre crochets
    - `[ expression_booléenne ]`
- Chaque opérateur d'interaction possède un mot clé qui est placé dans l'onglet nommant le fragment combiné

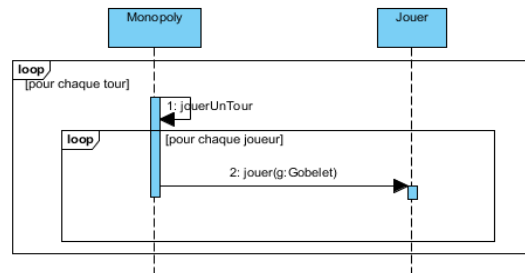
# Fragments combinés

- Alternatives
  - correspond au `if...else`
  - opérateur : `alt`



# Fragments combinés

- Boucle
  - opérateur : `loop`



# Fragments combinés

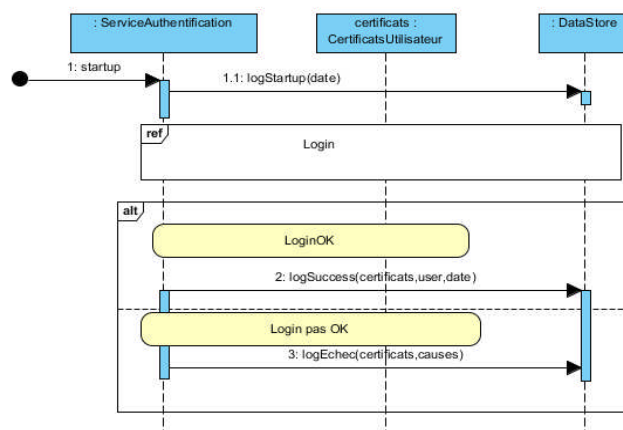
- D'autres types de fragments existent
  - diversement pris en charge par les outils
  - opérateurs
    - de rupture : `break`
      - comme un `return`
    - exécution parallèle : `par`
    - région critique : `critical`
      - doit être exécuté sous forme atomique
    - assertion : `assert`

# Diagrammes d'interaction

- Les diagrammes peuvent devenir vite complexes
- UML permet de les décomposer et de les lier entre eux en créant une décomposition en sous-parties
  - une référence vers un autre diagramme est mis en place par un fragment d'interaction avec l'opérateur `ref`

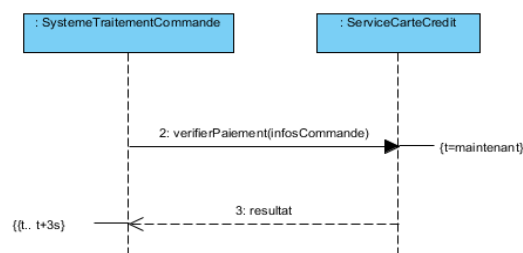
# Diagrammes d'interaction

- Exemple de référence



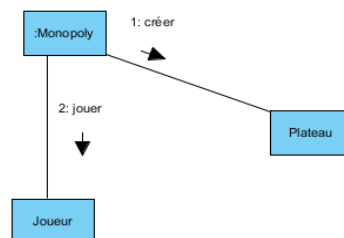
# Diagrammes d'interaction

- Chronométrage de séquence
  - on utilise une ligne horizontale face à l'événement, sur laquelle est indiquée la contrainte de temps
  - on utilise souvent une variable pour identifier l'instant précis de départ



# Diagramme de communication

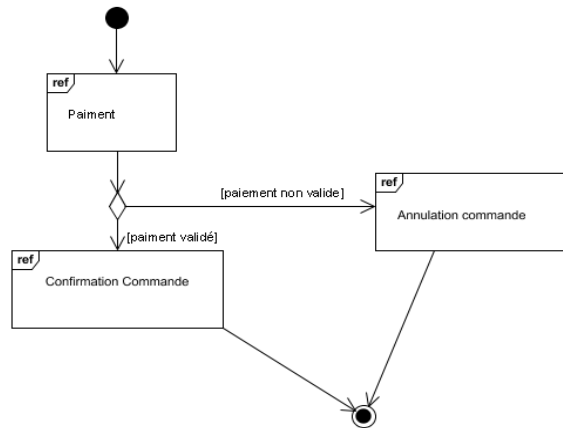
- Ancien diagramme de collaboration en UML 1
- Formalisme différent du diagramme de séquence





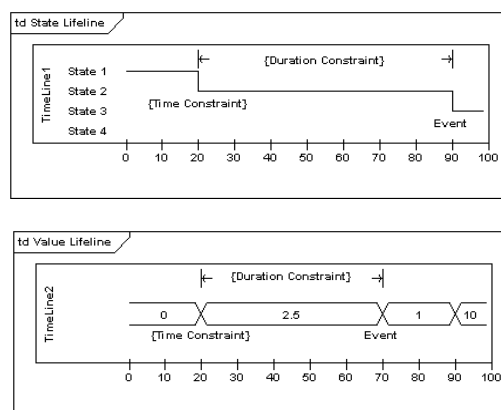
# Diagrammes d'interaction d'ensemble

- Vue simplifiée de l'activité
- Visualisation des grand flux de contrôle



# Diagrammes de chronométrage

- Représente les interactions sous forme temporelle



# Diagramme d'état

- UML distingue
  - les machines d'état comportementales
    - décrivent le comportement des éléments d'un système
    - une machine d'état comportemental représente une implémentation spécifique d'un élément
      - états d'une facture par exemple
  - les machines d'état à protocole
    - décrivent le fonctionnement d'un protocole
    - une machine d'état à protocole n'est pas liée à une implémentation, elle ne fait que décrire le protocole

# Diagramme d'état

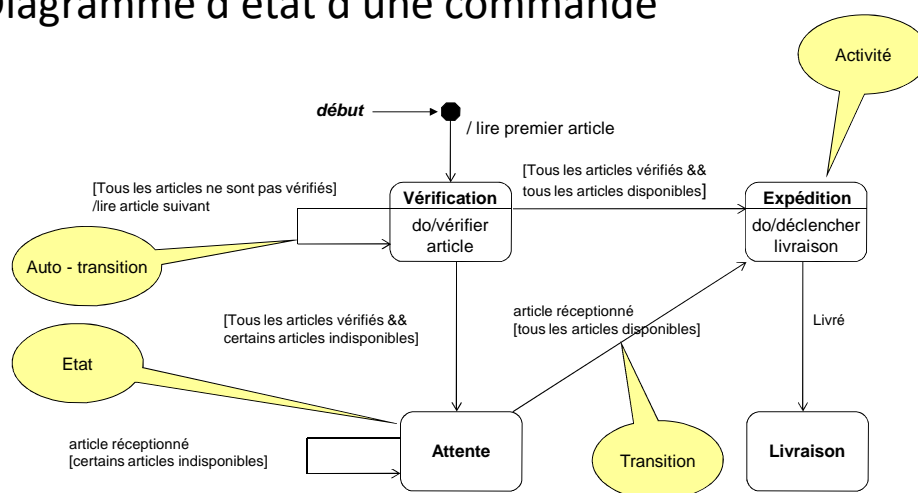
- Description du comportement d'un système
  - description des états que peut prendre un objet
- Un état
  - modélise un moment spécifique, une situation
  - caractérisé par
    - son nom
    - des activités internes (facultatif)
      - liste des activités qui sont exécutées dans l'état
        - » activités en entrée (entry), liées à l'état (do), en sortie (exit)
        - » syntaxe : étiquette / activité
    - liste des transitions internes (facultatif)

# Diagramme d'état

- Une transition
  - représente une relation, chemin, entre deux états
  - syntaxe :
    - déclencheur[garde] / action
      - déclencheur : événement
      - garde : contrainte évaluée lorsqu'un événement est produit
      - action : activité qui est exécutée

# Diagramme d'état

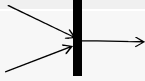
- Diagramme d'état d'une commande



# Diagramme d'état

- Pseudo-états
  - décrivent un comportement spécifique durant des transitions entre des états réguliers
  - syntaxe de base des diagrammes d'état
  - ils sont liés au moyen de transitions élémentaires

# Diagramme d'état

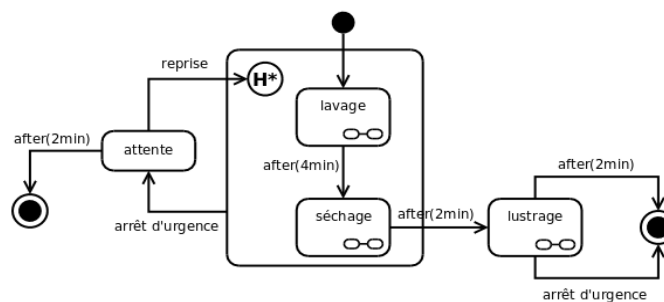
pseudo-état	symbole	description
Initial pseudo-state	●	point de départ
Choice	◊	permet de choisir entre différents états en fonction de garde sur les transitions
Deep History	⊕	indique que la machine doit se replacer dans le dernier sous-état, quel que soit le niveau de profondeur
Entry point	○	point d'entrée pour une transition vers un état composite
Exit point	⊗	point de sortie pour une transition à partir d'un état composite
Fork and join		séparation et rassemblement d'état. La machine ne subit pas de transition tant que tous les rassemblement ne seront pas effectués

# Diagramme d'état

pseudo-état	symbole	description
Junction	●	fusionne différentes transitions possibles
Shallow History	⊙	indique que la machine doit se replacer dans le dernier sous-état, ce sous état devant être de même profondeur
Terminal node	⦿	provoque la fin de l'exécution de la machine à état

# Diagramme d'état

- Exemple de diagramme d'état avec historique



# Diagramme de déploiement

- Définit les modalités d'installation des composants logiciels sur les dispositifs matériels
  - chaque nœud représente une unité informatique
  - les connexions entre les nœuds représentent les voies de communication

