Modélisation O-O avec UML

Reda Bendraou
reda.bendraou{{@}}Lip6.fr
http://pagesperso-systeme.lip6.fr/Reda.Bendraou/

Le contenu de ce cours a été influencé par les lectures citées à la fin de ce support.

La Modélisation

-Définition
-Pourquoi modéliser?
-Quel Langage utiliser?
-Logiciel=Code?

En quoi consiste la modélisation?

Construire une représentation abstraite de la réalité

Abstraction =

- Ignorer les détails insignifiants
- Ressortir des détails les plus importants
  - Important= décider de ce qui est signifiant et de ce qui ne l'est pas, dépend directement de l'utilisation visée du modèle

Abstraction: « Opération intellectuelle qui consiste à isoler par la pensée l'un des caractères de quelque chose et à le considérer indépendamment des autres caractères de l'objet. »

Source Larousse en ligne.

Une autre définition de la modélisation

Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose

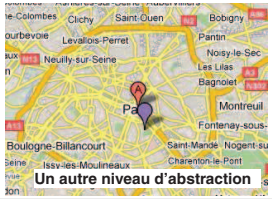
A **Model** represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality

Par Jeff Rothenberg, « The nature of modeling »

- **Attention au débat: abstraction = simplification?**
  - **La modélisation simplifie la compréhension et la communication autour du problème, elle ne simplifie pas le problème lui même!**

Exemple: Google Maps

- Différents niveaux d'abstractions
- Différents points de vues



Pourquoi Modéliser?

Pour gérer la complexité

- Des applications de plus en plus énormes
  - Windows 2000: ~40 millions de lignes de code
  - Impossible à gérer par un seul programmeur
- Permet de réfléchir sur la conception
- Séparation des préoccupations
  - Plusieurs vues sur le même problème (statique, comportementale, etc.)
- Exemple: Google
  - 400 000 Serveurs, une capacité de calcul gigantesque
  - 1 milliard de requêtes par jour
    - Chacune interrogeant 8 milliards de pages web en moins d'1/5 de S.

Pourquoi Modéliser?

Pour communiquer

- Plusieurs acteurs dans le procédé de développement
  - Clients, manager, marketing, ingénieurs système & réseaux, etc.
- Le code n'est pas toujours compréhensible par les développeurs qui ne l'ont pas écrit
  - On peut difficilement communiquer avec du code, **pas assez abstrait!**
- Dans de gros projets souvent :
  - Des centaines de personnes
  - un peu partout dans le monde
- L'apparition de nouvelles façons de travailler
  - L'outsourcing, sous-traitance, etc.

Pourquoi Modéliser?

Pour pérenniser un savoir-faire

- Certains projets peuvent durer des années
  - Pas toujours les mêmes personnes qui travaillent sur le projet
  - Besoin de capitaliser un savoir faire indépendant du code et des technos
  - Capturer le métier sans se soucier des détails techniques

La vision MDE

- Exemples de projets:
  - Contrôleur aérien (Thalès): Projet ~8 ans, durée de vie 40 ans
  - La construction d'un avion (Airbus): Projet ~10 ans, durée de vie 50 ans

Pourquoi Modéliser?

Augmenter la productivité

- Génération de code à partir des modèles
  - La vision MDE (Model-Driven Engineering)
  - 100% du code généré dans certains domaines d'application
    - Exp. Site Web, fichier de configuration, BD, etc.
- Maîtrise de la variabilité
  - La vision ligne de produit
    - Un modèle générique pour un produit, plusieurs variantes
- Exemple: Nokia
  - 1,1 milliard de téléphones portables (100 millions de plus par an)
  - Des milliers de versions de logiciels
  - Time-to-market ~3 mois

Logiciel = Code ?

- Est-ce vraiment nécessaire de poser encore la question?
- Avant oui mais plus aujourd'hui:
  - Logiciel= Documentation + Modèles + Code
  - Plusieurs Modèles, vues pour le même logiciel
  - De la documentation générée en partie des modèles
  - Du code généré en partie des modèles (100% dans certains cas)
- Grâce aux modèles aujourd'hui
  - Une meilleurs gestion de la complexité, de la variabilité et de la productivité

Mais quel langage utiliser pour modéliser?

- Plusieurs langages existaient/existent
- En réalité un seul a réussi à s'imposer comme Le LANGAGE pour la modélisation d'applications OO
- UML (Unified Modeling Language): Pourquoi l'utiliser?
  - Un standard mondialement utilisé (plus de 80% des projets)
  - Standardisé par l'OMG (Open Mangement Group)
  - Très bien outillé et documenté (livres, tuto, forums, etc.)

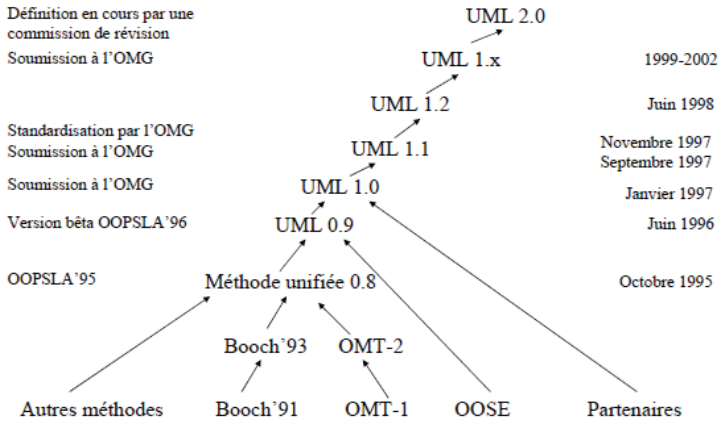
UML

- Historique
- Les utilisations possibles d'UML
-Le processus de développement avec UML
-Les phases de développement couvertes par UML
- UML: des diagrammes et des points de vue

Naissance d'UML

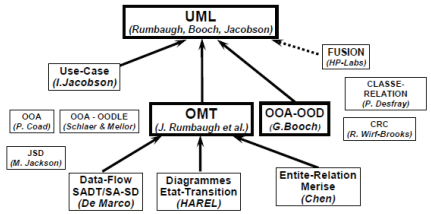
- Entre 89 et 94 : le nombre de méthodes orientées objet est passé de 10 à plus de 50
- **Chaque méthode proposait son propre langage de modélisation**
- Toutes les méthodes avaient pourtant d'énormes points communs (objets, méthode, paramètres, …)
- Au milieu des années 90, G. Booch, I. Jacobson et J. Rumbaugh ont chacun commencé à adopter les idées des autres. Les 3 auteurs ont souhaité créer un Langage de Modélisation Unifié

Historique



Généalogie d'UML

- L'histoire n'a retenu que les 3 amigos alors que tant d'autres ont participé à la standardisation d'UML
- UML évolue régulièrement (sans les 3 amigos ☺)



UML: Principales influences

- **Booch:** Catégories et sous-systèmes
- **Embley:** Classes singletons et objets composites
- **Fusion:** Description des opérations, numérotation des messages
- **Gamma, et al.:** *Frameworks*, *patterns*, et notes
- **Harel:** Automates (*Statecharts*)
- **Jacobson:** Cas d'utilisation (*use cases*)
- **Meyer:** Pré- et post-conditions
- **Odell:** Classification dynamique, éclairage sur les événements
- **OMT:** Associations
- **Shlaer-Mellor:** Cycle de vie des objets
- **Wirfs-Brock:** Responsabilités (CRC)

UML Aujourd'hui

- **UML est le langage de modélisation OO le plus connu et le plus utilisé**
- UML s'applique à plusieurs domaines
  - OO, RT, Déploiement, Requirement, ...
- **UML n'est pas une méthode**
  - Exemple de méthode: RUP (Rational Unified Process)
- **Peu d'utilisateurs connaissent le standard**, ils ont une vision outillée d'UML (Vision Utilisateur)
  - 5% forte compréhension, 45% faible compréhension, 50% aucune compréhension
- **UML est fortement critiqué car pas assez formel**
- Le marché UML est important et s'accroît
  - MDA et MDD, UML2.1, IBM a racheté Rational !!!

Trois utilisations possibles d'UML

(1) Comme un langage pour faire des croquis, esquisses, ébauches... (explorer)

- Pour échanger, communiquer rapidement autour d'une idée
- Des modèles pas forcément 100% complets
- **Objectifs:** Analyser, **réfléchir**, décider (brainstorming)
- Probablement la manière dont UML est le plus utilisé aujourd'hui

Trois utilisations possibles d'UML

(2) Comme un langage de spécification de modèles, patrons... (spécifier)

- Des modèles complets, des modèles prêts à être codés
- Des modèles utilisés pour prendre des décisions poussées de design
- Des modèles issus de Reverse Engineering
  - Cela permet de réfléchir dessus, d'améliorer la conception,
  - De casser des dépendances, d'appliquer des design patterns, etc.
- Possibilité de faire du round-trip engineering (outils pas encore très matures)
  - Génération de code (classes et signatures de méthodes, **pas le corps!!!**)
  - Reverse engineering pour isoler une vue ou un élément du système
- **Objectifs:** (1)+ Concevoir, **Pérenniser**, Générer une partie du code
- **La manière dont est utilisé UML dans de gros projets**

Trois utilisations possibles d'UML

(3) Comme langage de programmation (produire)

- Tout mettre dans le modèle, le modèle UML devient la source du code
  - même le corps des méthodes, en Java et dans d'autres formalismes tels que Executable UML, J, xOCL, etc.
- Génération + compilation du code à partir du modèle
- Un outillage sophistiqué par assez mature malheureusement
- **Objectifs:** (2) + génération automatique de tout le code
- La manière dont on aimerait utiliser UML dans le futur mais le domaine reste encore immature malgré quelques tentatives
  - La difficulté réside à modéliser la logique du comportement
- « It's worth using UML as a programming language only if it results in something that's significantly more productive than using another programming language »
  - Martin Fowler, UML Distilled

Dans ce cours

- On utilisera UML comme dans (1) (Explorer) et (2) (Spécifier)
- On expliquera pourquoi (3) n'est pas encore mature

Le processus de développement avec UML

- UML est indépendant du procédé ou de la méthode de développement
- Cependant le processus de développement avec UML est défini comme étant:
  - Itératif & Incrémental
  - Dirigé par les cas d'utilisation
  - Centré sur l'architecture
- Après la présentation des différents diagrammes nous proposerons une méthode de développement utilisant UML qu'on nommera UML-P6

Les phases du développement couvertes par UML

- Expression des besoins
- Analyse
- Conception

Bien couvertes par UML

- Réalisation
- Validation
- Déploiement
- Maintenance

Discutable:
Réalisation: si utilisation d'UML comme langage de programmation.
Validation: génération automatique de cas de tests mais loin d'être exhaustifs
Déploiement: un diagramme UML pour ça
Maintenance: possibilité de faire du reverse eng. Application de Design patterns, round-trip eng.

Dans ce cours, nous essayerons de voir toutes ces utilisations d'UML

UML: des diagrammes et des points de vue

- Présentation des diagrammes par point de vue

UML: des diagrammes et des points de vue

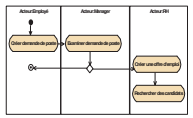


Diagramme d'activité

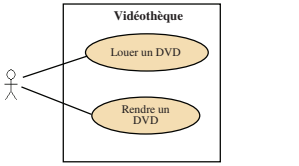


Diagramme de Cas d'Utilisation

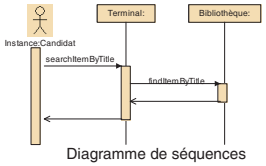
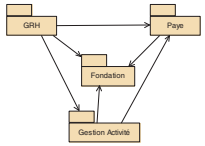


Diagramme de séquences



Digramme de packages

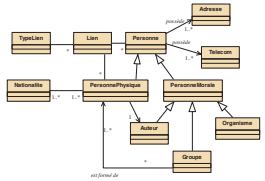


Diagramme classes

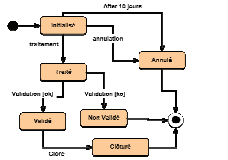


Diagramme états/transitions

UML: des diagrammes et des points de vue

Aspects fonctionnels du Système
-Diagramme de cas d'utilisation
-Diagramme de séquences
-[Diagramme d'activités]

Aspects Dynamiques du Système
-Diagramme de séquences/[communication]
-Diagramme d'états transitions
-Diagramme d'activités
-Diagramme de temps

Aspects Statiques du Système
-Diagramme de classes
-[Diagramme d'objets]
-Diagramme de structure composite

Aspects déploiements du Système
-Diagramme de déploiement
-Diagramme de composants

En rouge: abordé dans ce cours
Entre [ ]: peu utilisé dans la vue

UML: Point de vue Fonctionnel

-Diagramme de Cas d'utilisation
-Les Scénarios

UML: Point de vue Fonctionnel

- UML propose pour cette vue, le diagramme de Cas d'Utilisation ou Use Case Diagram
- Les Cas d'utilisations peuvent être documentés par la suite avec:
  - Des scénarios (langage naturel),
  - Des diagrammes de séquences pour formaliser les scénarios
  - Des diagrammes d'activités pour exprimer le workflow
- Point de départ important dans le processus de dév. avec UML
  - Dirigé par les use cases

Le diagramme de Cas d'Utilisation

Un diagramme centré utilisateur

- Un système est conçu pour les utilisateurs :
  - ils savent ce que le système doit faire mais pas comment
  - ils connaissent l'aspect fonctionnel du système

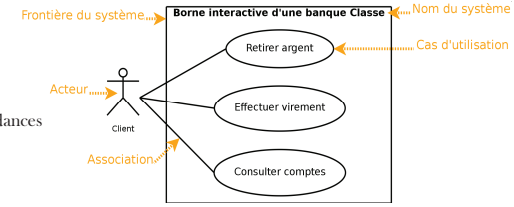
- Les utilisateurs du système sont les plus aptes à décrire comment ils s'en servent

➔ **Le système doit donc être bâti à partir des descriptions des utilisateurs.**

Le diagramme de Cas d'Utilisation

Le diagramme des cas d'utilisation est modélisé par :

- Des acteurs
- Des cas d'utilisation
- Le Système
- Des relations
  - Associations, Héritage, dépendances



Ses objectifs sont :

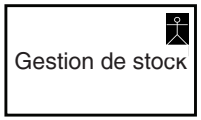
- Communiquer
- Capturer les besoins fonctionnels du système
- Délimiter le système
- Visualiser le cahier des charges graphiquement
- Peut servir à concevoir les tests

Le diagramme de Cas d'Utilisation: Acteur

- **Un Acteur** représente un rôle joué par une entité externe qui interagit directement avec le système étudié (déf. UML2.0, omg)

- Ça peut être un utilisateur humain (ex. Agent, caissier, client, etc.), un dispositif matériel (ex. serveur, imprimante, etc.) ou un autre système (ex. Gestion de stock, etc.);

- Notations graphiques



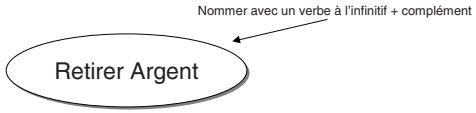
Le diagramme de Cas d'Utilisation: Acteur

- Comment identifier les acteurs?
  - Par un dialogue avec le client et les utilisateurs
  - En délimitant les frontières du système

- Qui sont-ils ?
  - les utilisateurs humains: les profils possibles sans oublier l'administrateur et l'opérateur de maintenance
  - les autres systèmes connexes interagissant avec le système : logiciels, périphériques ...

Le diagramme de Cas d'Utilisation: Cas d'Utilisation

- Un **Cas d'Utilisation** représente un ensemble de séquences d'actions qui sont réalisées par le système et qui produisent un résultat observable intéressant pour un acteur particulier (déf. UML2.0, omg)
- Chaque Cas d'utilisation **spécifie un comportement attendu du système considéré comme un tout**, sans imposer le mode de réalisation de ce comportement.
- Il permet de décrire ce que le futur système devra faire, **sans spécifier comment il le fera**
- Notations graphiques



Le diagramme de Cas d'Utilisation: Cas d'Utilisation

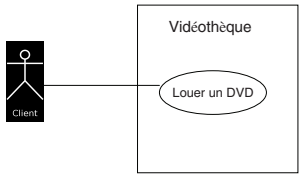
Comment identifier les cas d'utilisations?

- Répondre aux questions suivantes:
  - Quels sont les services rendus par le système ?
    - Chaque fonctionnalité métier est représentée par un use case
  - Quelles sont les interactions Acteurs/ système ?

    Pour chaque acteur identifié
    - Rechercher les différentes intentions métiers avec lesquelles il utilise le système
    - Déterminer les services fonctionnels attendus du système par cet acteur
    - Ne pas oublier celui qui maintient le système
  - Quels sont les évènements perçus par le système (externes, temporels, changement d'état) ?

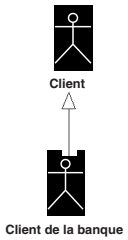
Le diagramme de Cas d'Utilisation: Le Système

- **Le Système** représente les limites de l'application considérée et regroupe un ensemble de cas d'utilisation
- Notations graphiques

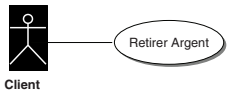


Le diagramme de Cas d'Utilisation: Les Relations

- Entre acteurs
  - Héritage



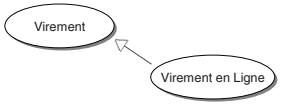
- Entre acteurs & UC
  - Association



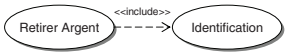
Le diagramme de Cas d'Utilisation: Les Relations

Entre Cas d'utilisation: trois types de relations

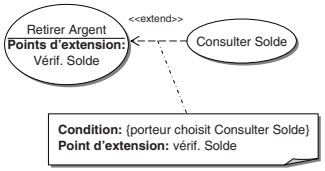
- Généralisation (héritage): le cas fils spécialise le cas père. Sémantique cependant ambigüe dans le cadre des UC. À éviter



- Inclusion (<<include>>): le cas source incorpore directement et **nécessairement** le cas cible à un endroit précis dans son enchaînement. **Le cas inclus n'est jamais exécuté tout seul.** À utiliser pour factoriser



- Extension (<<extend>>): le cas de base en incorpore indirectement et **pas nécessairement** un autre à un endroit précis dans son enchaînement. **Le cas de base peut être exécuté tout seul.** À utiliser avec parcimonie



Le diagramme de Cas d'Utilisation

Quelques conseils:

- Granularité des UC 2 UC Vs. 15 UC
  - « *There is a magic number:7, plus or minus 2. This refers to the number of concepts that we humans can keep in mind at any one time* » (H.A. Miller, 1958)
  - Notions de fonctions fondamentales Vs. Opérationnelles
  - Possibilité après d'abstraire (réduire le nombre de UC en définissant des fonctions plus globales) ou de décomposer (en définissant des fonctions à grain plus fin – **déconseillé!**)
- Eviter d'abuser des <<include>>, <<extend>> et de l'héritage entre UC.
- Processus itératif et centré utilisateurs pour définir les UC

Le diagramme de Cas d'Utilisation: Conclusion

- **Les diagrammes de cas d'utilisation sont souvent employés**
  - Ils permettent de décrire le système de façon très abstraite
  - Ils offrent une vue fonctionnelle (par opposition à une vue OO)
  - Ils sont très simples
- **Les CU se limitent à décrire le « quoi » d'un système, pas le « comment »**
- **Les cas d'utilisation sont une description fonctionnelle d'un système après quoi il faut passer à une description objet (les scénarios) utilisant :**
  - les diagrammes de séquences ou les diagrammes d'activités comme alternative ;

Les Scénarios

- Scénarios et Cas d'utilisation
- Exemple

Les Scénarios

- Les scénarios sont des instances de cas d'utilisation
  - Un scénario est un exemple de dialogue entre le système et un ou plusieurs acteurs
- **Souvent décrits en langage naturel puis formalisés à l'aide de diagrammes de séquences UML**
  - Une séquence d'actions décrivant l'interaction entre utilisateur et système
- Ils font partie de la documentation des CU. (très conseillé de les faire)
- Certains s'en servent pour identifier les Cas d'Utilisation
  - Plusieurs scénarios ayant le même but utilisateur => un CU.

Les Scénarios

- **Chaque cas d'utilisation va avoir un ensemble de scénarios**
- des scénarios primaires (tout va bien, le Happy Path)
- des scénarios secondaires (les exceptions)
- L'ensemble des scénarios + UC constitue + ou - le cahier des charges du système

Les Scénarios: Exemple

- Exemple de documentation d'un UC/Scénario

* Use case #	< Nom du cas d'utilisation rattaché au scénario>	
* Scénario #	**< Nom du scénario correspondant à l'objectif sous forme d'une phrase courte à la forme active>**	
* Objectif dans le contexte	<Description détaillée de l'objectif dans le contexte>	
* Pré conditions	<Etat du système avant « exécution » du use case>	
* Post condition (réussite)	<Etat du système quand l'objectif du use case est réalisé avec succès>	
* Post condition (échec)	<Etat du système quand l'objectif du use case n'est pas atteint>	
* Règles de gestion	<Ensemble de règles permettant de caractériser ou d'exprimer les contraintes sur les éléments manipulés, leur valeur et les contrôles mis en œuvre>	
* Description	étape	Action
	1	<Mettre ici les étapes du scénario depuis l'événement déclencheur jusqu'à
	2	<…>
	3	
Extensions	étape	Action de branchement
	1a	<condition du branchement> : <action ou nom du use case "extend">
Priorité	<niveau de criticité pour le système / l'organisation>	

Les Scénarios: Exemple

- Le scénario Retirer Argent (le Happy Path):

1) Le **client** introduit sa carte bancaire
2) Le **système** demande le code secret
3) Le **client** saisit son code secret. Le **système** vérifie le code
5) Le **système** demande le montant à retirer. Le **client** saisit le montant
6) Le **système** retourne la carte bancaire
7) Le **client** reprend sa carte bancaire
8) Le **système** délivre les billets correspondant au montant
9) Le **client** prend les billets

- Identifier par la suite les différents scénarios d'exceptions
  - La technique du If (faire un « si » ça ne marche pas à chaque étape)
- Prochaine étape, **formaliser les scénarios à l'aide de diagrammes de séquence**

Diagramme de Séquence et d'Activités

- Ces diagrammes peuvent être utilisés dans cet vue
- Pour représenter visuellement les scénarios
- Pour représenter visuellement des fonctionnalités compliquées
- Nous les présenterons dans le contexte de la vue dynamique

UML: Point de vue Statique

-Rappel OO
-Diagramme de Classes
-Diagramme d'Objet
- Diagramme de Structure Composite

Rappel OO

-Vision OO
-Concepts de base

Vision OO

- Moyens d'organiser le logiciel (application) sous forme d'une collection d'objets qui encapsulent ensemble structures de données et comportement
  - Plutôt que sous forme de fonctions
- Concepts dans ce rappel
  - Objet
  - Classe
  - Message
  - Encapsulation
  - Héritage
  - Polymorphisme

Les Objets

- **Les objets informatiques définissent une représentation simplifiée des entités du monde réel**
- Les objets représentent des entités concrètes (avec une masse) ou abstraites (concept)
- Les objets encapsulent une partie de la connaissance du monde dans lequel ils évoluent

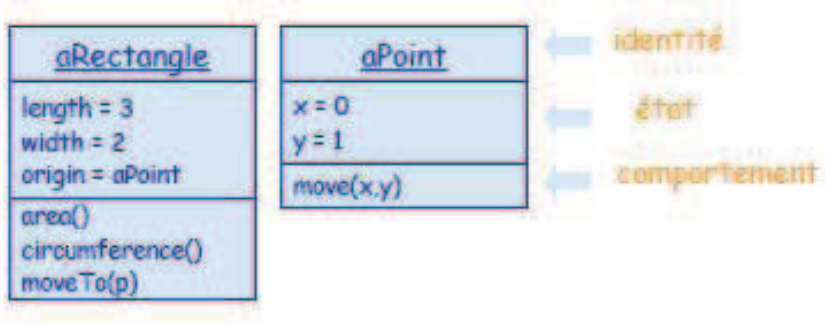
Un Objet

Identité
- constante
- permet à un objet d'être manipulé, référencé par d'autres

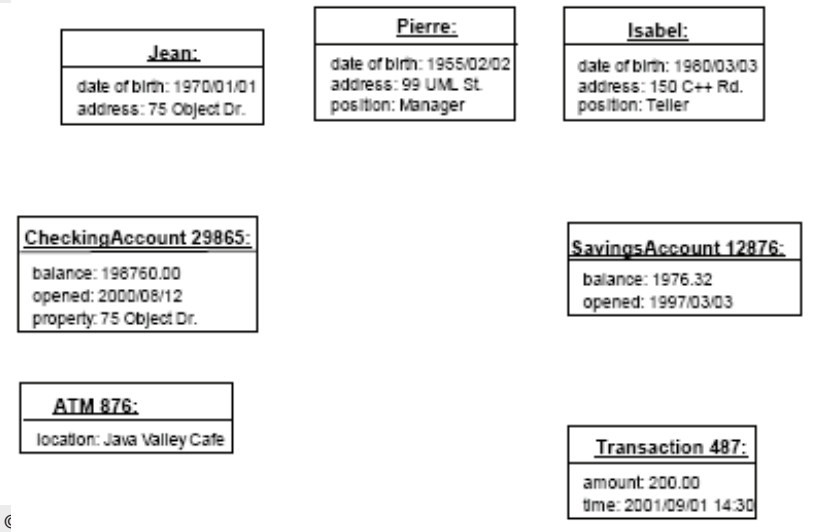
Etat
- variable
- conforme à un certain type (classe)
- caractérise la "valeur" d'un objet à un instant donné

Comportement
- opérations que peut accomplir l'objet
- invoqué via l'interface de l'objet
- modifie ou non l'état

Objet : Exemples



Objet: Exemples



Messages & Méthodes

- **Messages**
  - La seule façon de communiquer entre objets
  - Indiquent quel comportement les objets doivent exécuter
- **Méthodes**
  - Indiquent comment répondre à un message (comment le comportement s'exécute)
  - Ont accès aux données de l'objet (i.e. état)

Encapsulation

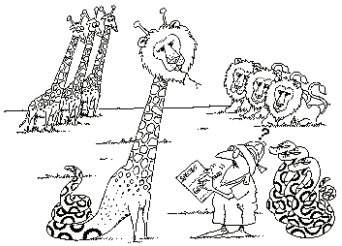
- **Les données de l'objet (son état) sont privées**
- Le comportement d'un objet est encapsulé
  - Les clients d'un objet connaissent seulement l'ensemble de messages que l'objet peut recevoir
  - Les implémentations des méthodes restent cachées aux clients externes

Encapsulation

- **Interface : une vitrine de l'objet**
- Les actions exécutées sont masquées à l'initiateur du message
- Délégation: un objet peut faire appel à un autre objet afin d'honorer son service

Classe

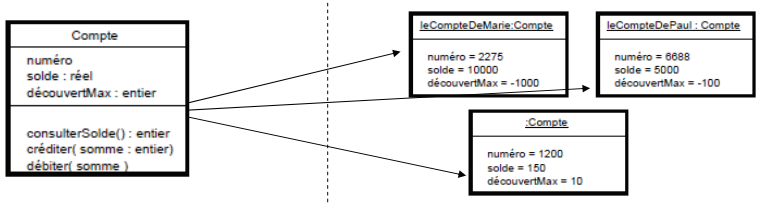
- **Une unité d'abstraction**
- **Un mécanisme de groupage, de classification**
  - Une collection d'objet similaires
  - Chaque objet est l'instance d'une classe
  - La classe est le type de l'objet



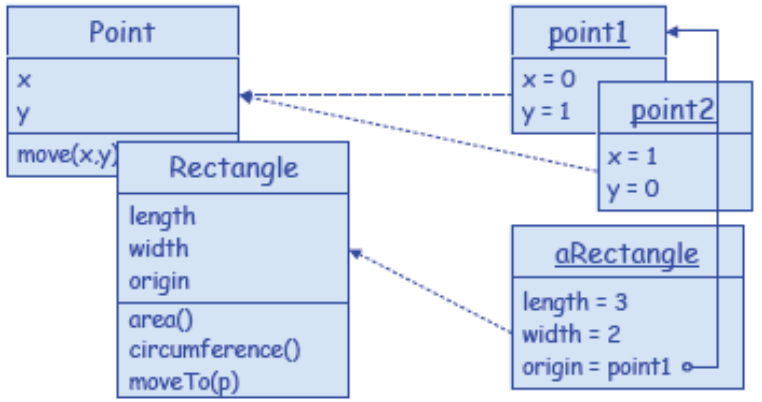
- Décrit la structure commune à une collection d'objets en termes de propriétés et de méthodes (comportement)

Classe Vs. Objets

- Une classe spécifie la structure et le comportement d'un ensemble d'objets de même nature (un moule)
  - La structure d'une classe est constante
- Les objets sont des instances de classe, ils peuvent être détruits ou ajoutés pendant l'exécution
- La valeur des attributs des objets peut changer



Classes et Instances



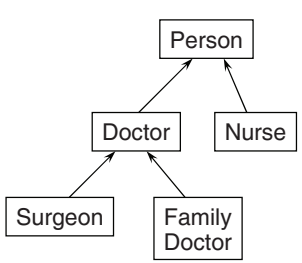
Variables d'instances

- Valeurs de celles-ci propres à l'instance
- Versus Variable de classe: partagée par toutes les instances de la classe
  - Notion de statique en Java ou C++
  - Si une instance modifie la valeur de la variable de classe, toutes les autres seront affectées

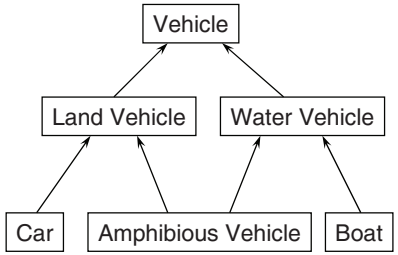
Héritage

- Permet la réutilisation de l'état et du comportement d'une classe par d'autres classes
- Super-classe
  - Contient les éléments communs à un ensemble de sous-classes
  - Les sous-classes héritent toutes les variables et méthodes de la super-classe
  - Les sous-classes raffinent la définition de leur super-classe

Héritage -exemple



simple



multiple

Polymorphisme

- **Polymorphisme ad-hoc**
  - représente la possibilité de définir plusieurs fonctions de même nom mais possédant des paramètres différents (en nombre et/ou en type)
  - Ex.
    - La méthode *int addition(int, int)* pourra retourner la somme de deux entiers
    - La méthode *float addition(float, float)* pourra retourner la somme de deux flottants
    - La méthode *char addition(char, char)* pourra définir au gré de l'auteur la somme de deux caractères

Polymorphisme

- **Polymorphisme d'héritage (ou sous-typage)**
  - La possibilité de redéfinir une méthode dans des classes héritant d'une classe de base
  - Il est possible d'appeler la méthode d'un objet sans se soucier de son type intrinsèque
  - permet de faire abstraction des détails des classes spécialisées d'une famille d'objet, en les masquant par une interface commune (qui est la classe de base)

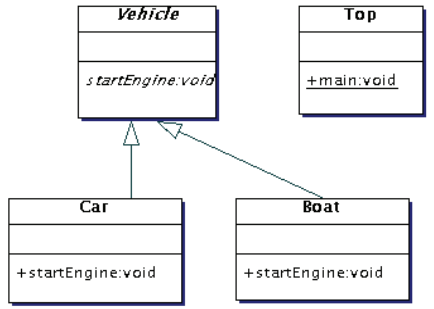
Polymorphisme

- **Polymorphisme d'héritage – Exemple**

```
Vehicle v = null;

v = new Car();
v.startEngine();

v = new Boat();
v.startEngine();
```



Polymorphisme

- **Polymorphisme d'héritage – Exemple**

```
class Forme {
public: virtual float Aire() = 0;
};
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
class Carre:public Forme {
public: virtual float Aire() {
return m_cote*m_cote;
}
private: float m_cote;
};
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
class Cercle:public Forme {
public: virtual float Aire() {
return 3.1415926535*m_rayon*m_rayon;
}
private: float m_rayon;
};
```

```
float AireTotal(Forme* tabl[], int nb) {
float s=0;
 for(int i = 0;i<nb; i++) {
 s+= tabl[i]->Aire(); // le programme sait automatiquement quelle fonction appeler
}
return s;
}
...
 Forme* tableau[3] = {new Carre, new Cercle, new Carre};
AireTotal(tableau,3);
...
```


Polymorphisme

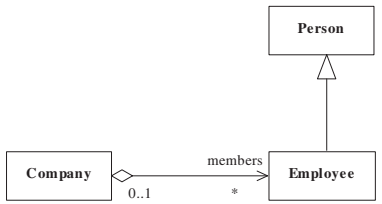
- **Polymorphisme généralisé (paramétré)**
  - correspond à la possibilité de définir des fonctions génériques et suppose la possibilité d'abstraire les types et peut être vu comme la forme ultime du polymorphisme

```
#include <iostream.h>
template <class C> void echanger(C &a, C &b){
C x;
x = a;
a = b;
b = x;
}
```


Diagramme de Classes

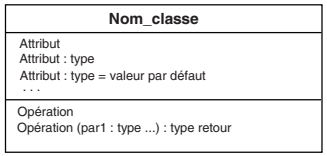
Diagramme de Classes

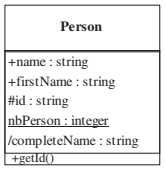
- Un diagramme de classes est un graphe d'éléments connectés par des relations
- Un diagramme de classes est une vue graphique de la structure statique d'un système



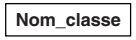
Classe

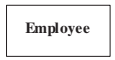
- Représentation détaillée d'une classe



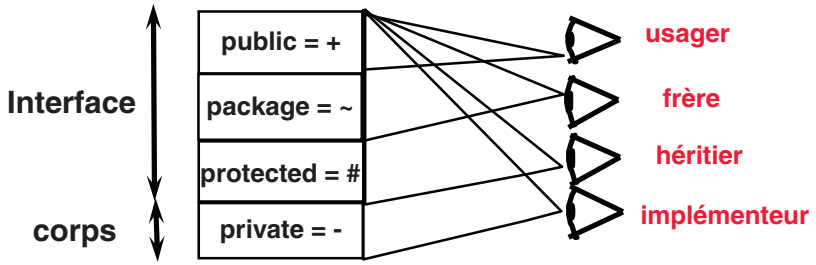


- Représentation simplifiée





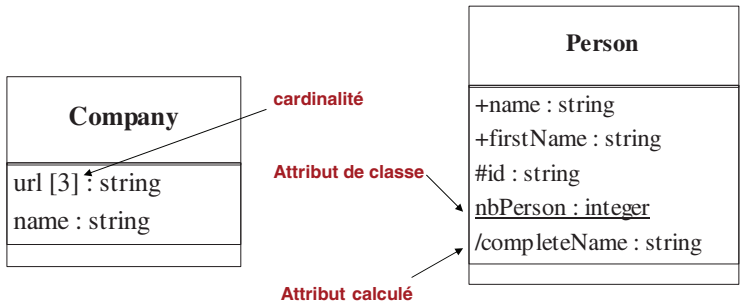
Les visibilités



Attributs

- Une classe peut contenir des attributs
- La syntaxe d'un attribut est :
  visibilité nom : type [= valeur par défaut]
- La visibilité est:
  - '+' pour public
  - '#' pour protected
  - '-' pour private
- UML définit son propre ensemble de types
  - Integer, real, string, …
- Un attribut peut être un attribut de classe, il est alors souligné.
- Un attribut peut être dérivé (calculé), il est alors préfixé par le caractère '/'

Attributs: Exemples

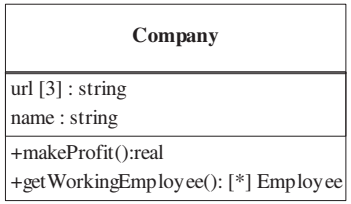


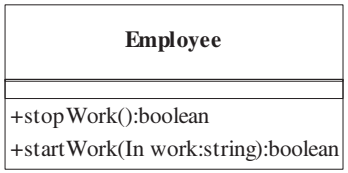
- Un attribut de classe est un attribut dont la valeur est partagée par toutes les instances de cette classe
- Un attribut dérivé (calculé), est un attribut dont la valeur est calculée à partir d'autres attributs

Opérations

- Une opération est un service qu'une instance de la classe peut exécuter
- La syntaxe d'une opération est:
  visibility name(parameter):return
- La syntaxe des paramètres est:
  kind name : type
- Le kind peut être:
  - in, out, inout

Opérations: Exemples



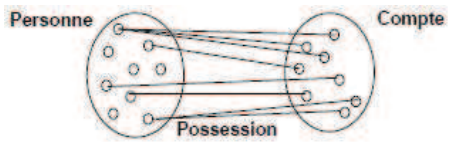


Les Associations

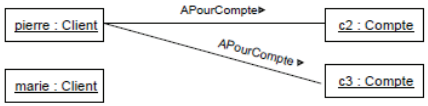
- Concept important dans UML
- Une relation met en correspondance des éléments d'ensembles
- Une relation permet la description d'un concept à l'aide d'autres concepts
- Très important: Une association est un lien stable (persistant) entre deux objets

Les Associations: Vue ensembliste

- Une association met en correspondance des éléments d'ensembles



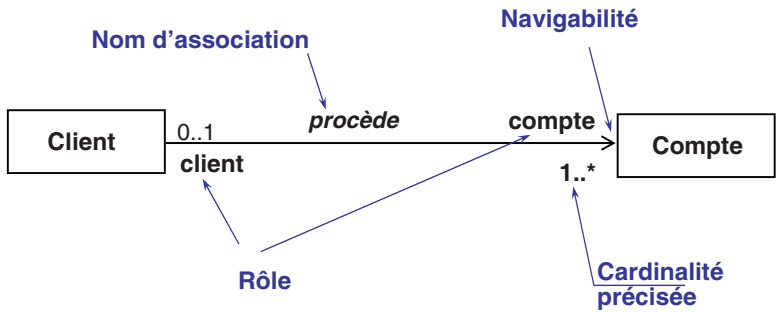
- Exemple avec un diagramme d'objet (présenté plus loin)



Les Associations

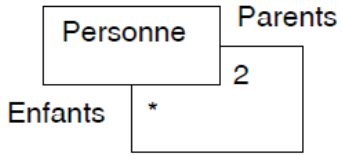
- Une association binaire est composée de deux associations ends.
- Une association end est paramétrée par:
  - Un nom (le role joué par l'entité connectée)
  - Une cardinalité (0, 1, *, 1..*, ...)
  - Un genre d'agrégation (composite, aggregation, none)
  - De plusieurs propriétés: isNavigable, isChangeable, etc.

Association: Notation



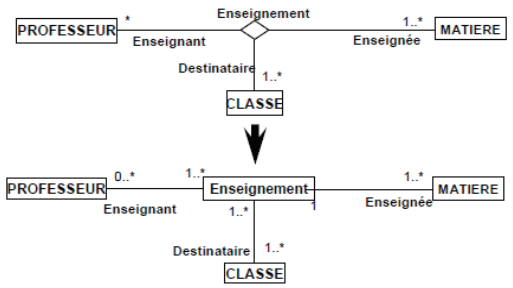
Association réflexive

- Une association réflexive lie des objets de la même classe



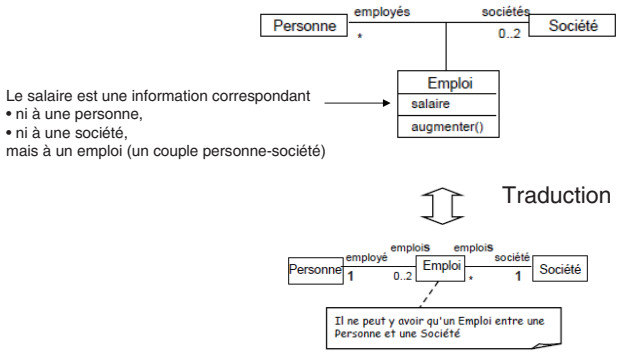
Les Associations N-aires

- Relation entre plus de deux classes
- Peuvent toujours être représentées autrement



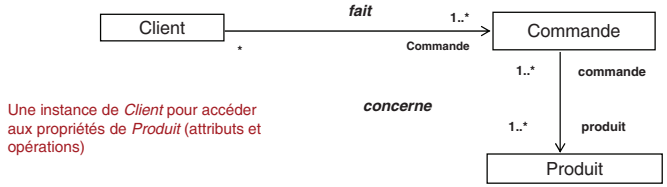
Classe d'association

- Quand l'association contient de l'information



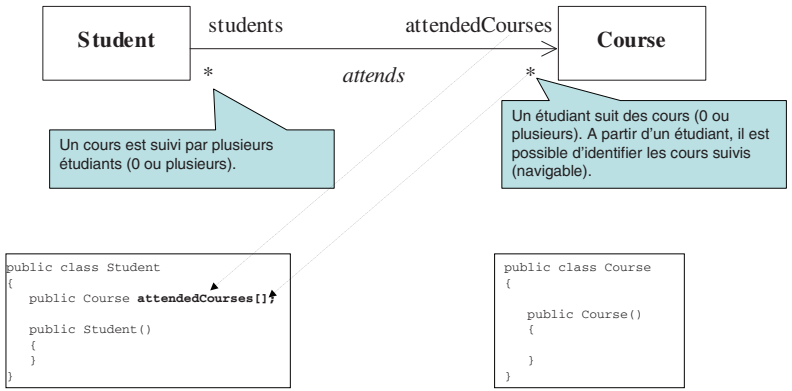
Association : Navigabilité

- Le moyen d'accéder aux propriétés (attributs et opérations) d'autres objets à travers le graph d'objets représentant l 'application
- Représentée par une flèche
  - Attention à la notation en cas de navigabilité dans les 2 sens



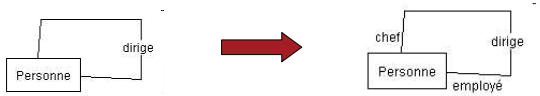
Association : Navigabilité

- Impact de la navigabilité et des noms de rôles sur la génération de code

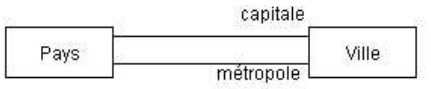


Associations: Rôles Indispensables

- Dans le cas d'une association réflexive

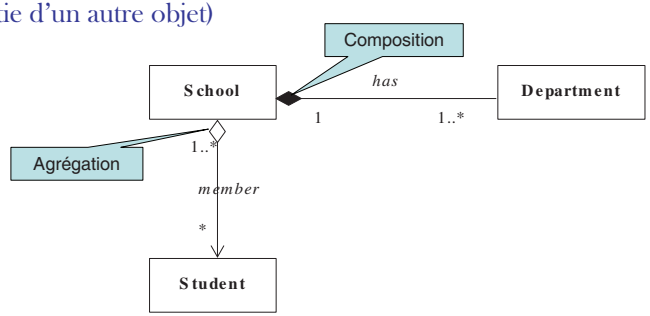


- Cas de plusieurs associations entre deux mêmes classes



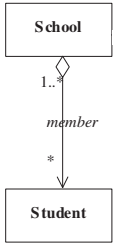
Associations: Agrégation & Composition

- Notion de « est composé de », « contient » « est construit à partir de », « est formé de », ...
- Renforce la sémantique d'association (un ensemble d'objets qui fait partie d'un autre objet)



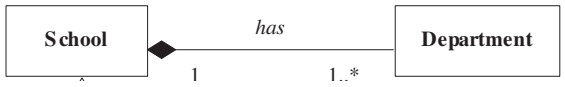
Associations: Agrégation

- L'agrégation exprime une composition faible(partagée)
- Un composant peut faire partie de plusieurs composites
  - La multiplicité coté composite peut être (1...*)
- Le cycle de vie du composant n'est pas lié à celui du composite
  - Si le composite est détruit, les composants restent



Associations: Composition

- Exprime une composition forte (n'est pas partagée)
- Le composant ne fait partie qu'un d'un seul composite
  - Multiplicité coté composite égale à 1
- Le cycle de vie du composant est lié à celui du composite
  - Si le composite est détruit, les composant n'ont plus raison d'être



Associations: Agrégation & Composition

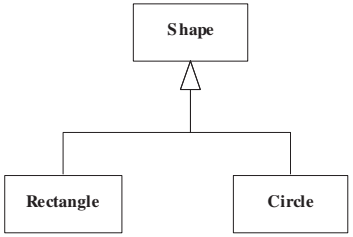
- Attention aux abus d'utilisation de ces types d'associations
- L'agrégation n'est pas trop utilisée vu qu'elle présente pas mal de similitudes dans la sémantique avec une association simple
  - Pas de contraintes sur les cardinalités
  - Juste sémantiquement un peu plus forte (le but initial dans le standard) et interdiction des cycles
  - Utilisation toujours ambiguë dans la communauté
- **Eviter de construire vos diagrammes de classe en vous posant les questions du genre, si je détruis cette classe est-ce que l'autre doit disparaître aussi, cette classe est forcement composée de cette classe, etc.**
  - Ces types d'associations doivent être des cas spéciaux et non la généralité

Généralisation (Héritage)

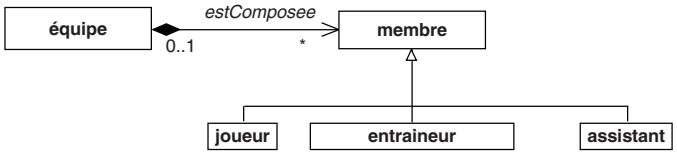
- L'héritage est un type de relation dans UML
  - et non pas un type d'association, elle aussi est une relation
- L'héritage permet de partager les points communs (attributs, opérations et associations), et de préserver les différences
- Peut être simple ou multiple
  - En Java l'héritage n'est que simple
- Peut être spécifié entre classes, packages, Use case, etc.
- Identifiable avec des phrase du genre « est une sorte de »

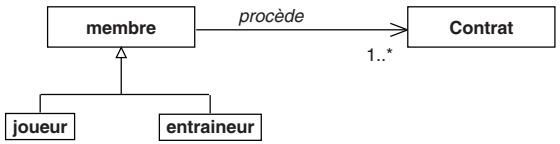
Généralisation: Notation

- On parle de Généralisation, Spécialisation
- Classe mère, classes filles
- Super classe, sous-classes



Généralisation: Héritage des associations

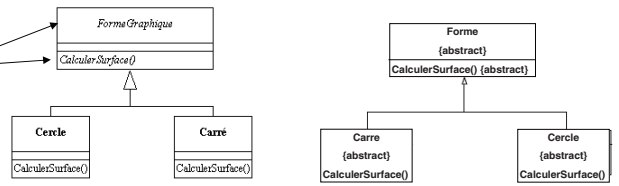




Les Classes et Opérations Abstraites

- Une classe abstraite est une classe qui contient au moins une opération abstraite
  - Capture des comportements communs
  - Servent à structurer le système
  - Ne peut pas être instanciée
- Une opération abstraite est une opération dont l'implémentation est laissée aux classes filles

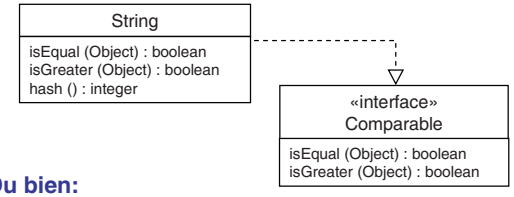
Attention: dans UML1.X, classe et opération abstraite étaient en italique. Ce n'est plus obligatoire dans UML2.0. Rajouter {abstract}



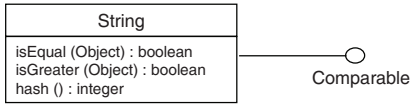
Les Interfaces

- La vitrine de la classe
- Un ensemble d'opérations sans corps
  - Juste la signature
  - Peut être vue comme une classe abstraite dont toutes les opérations sont abstraites
- Un puissant moyen de typage
- Une Classe peut réaliser une ou plusieurs interfaces
- Un contrat qui engage la classe qui réalise l'interface à implémenter ses opérations

Les Interfaces: Notation

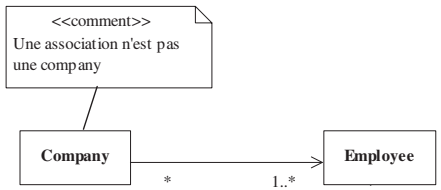


Ou bien:



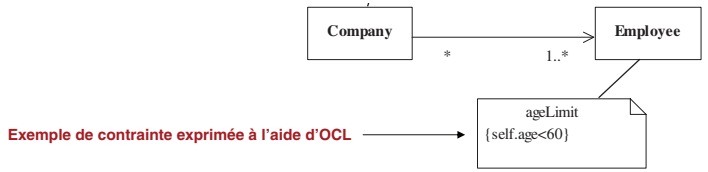
Les Notes (commentaires)

- Il est possible de rattacher à n'importe quel élément de digrammes UML des notes ou des commentaires
  - Pour mieux expliquer certains aspects, rajouter plus de précision
  - Certains outils s'en servent pour y mettre le code des opérations (en Java par ex.) afin d'aller vers une génération 100% du code à partir du modèle
- Notation graphique



Les Contraintes

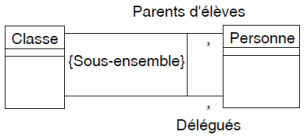
- Ça peut être des règles métiers, de bonne structuration du modèle, etc.
- Peuvent être exprimé en langage naturel, en utilisant les quelques contraintes définies par UML ({ordred}, {frozen}, etc.)
- Peuvent être formalisées à l'aide d'OCL (Object Constraint Language), standard OMG (pas abordé dans ce cours)

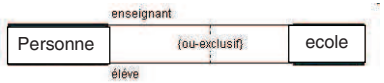


Autres Concepts

- On peut ajouter une contrainte {ordonné} (ordred) ou {trié} (sorted) pour transformer un "ensemble d'objets« en "liste d'objets" ordonnés ou triés.

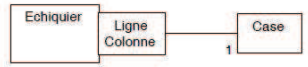
- La notion de sous-ensemble indique qu'une collection fait partie d'une autre collection. Le ou-exclusif ne le permet pas

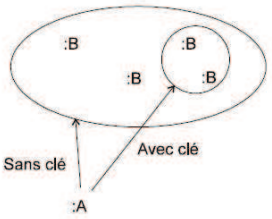




Autres Concepts

- Qualification ou Qualifieur permet la restriction des associations

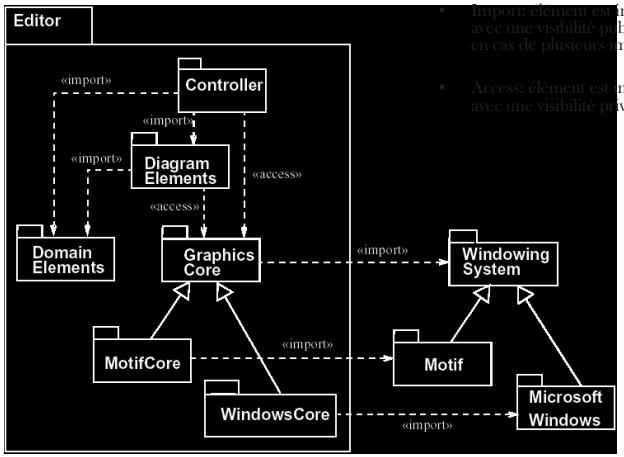




Les Packages

- Un package permet de grouper des éléments
  - Classes, use case, etc.
- Un package sert d'espace de désignation (nommage)
  - Deux classes portant le même nom ne peuvent être dans le même package
  - Précéder du nom de package si deux classes ont le même nom mais appartiennent à deux packages différents
- Un package peut inclure d'autres packages
- Un package peut importer ou utiliser d'autres packages
- L'héritage entre package est possible

Les Packages: Exemple



- Import: élément est importé dans le package avec une visibilité public. l'import est transitif en cas de plusieurs imports
- Access: élément est importé dans le package avec une visibilité private. Pas de transitivité

Diagramme d'Objets

Diagramme d'Objets

- Un diagramme d'objet représente la vue statique d'un ensemble d'instance de classes
- On parle d'objets et de liens et non pas de classes et d'associations
- Les rôles sont facultatifs
- Servent à valider le diagramme de classe, à donner des exemples
- Il est peu utilisé

Diagramme d'Objets: Exemple

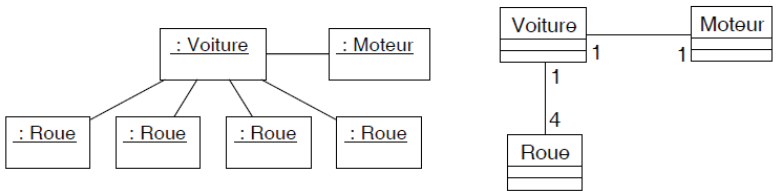
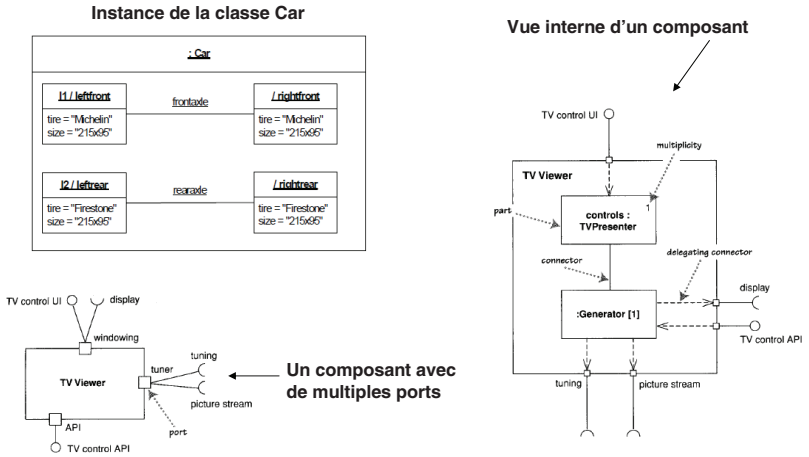


Diagramme de Structure Composite

Diagramme de Structure Composite

- Nouveauté dans UML 2.0
- Expose la structure interne d'une classe ainsi que les *collaborations* que cette dernière rend possible
  - Notion d'interfaces fournies et requises
  - Notion de parties: un *rôle* joué par une instance d'une classe ou un ensemble d'instances à l'exécution. La partie peut inclure une cardinalité
  - Notion de port: regroupe les interfaces fournies et requises selon des interactions logiques que le composant aura avec le monde extérieur ou avec ses parties
- Encore trop tôt pour voir s'il est utile/utilisé

Diagramme de Structure Composite: Exemples



UML: Point de vue Dynamique

-Diagramme de Séquence
-Diagramme de Collaboration
-Diagramme d'État/Transition
-Diagramme d'Activité

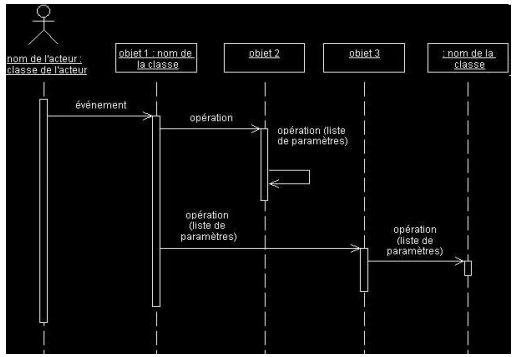
Les Diagrammes de Séquence

Diagrammes de Séquence

- **Objectif:** représenter des exemples (instances) d'interaction entre objets dans la réalisation de processus de l'application
  - **Interaction:** un ensemble de messages échangés par une partie du système
- Un diagramme de séquence capture typiquement le comportement d'un seul scénario
  - Possibilité de faire des IF dans le même D.S pour gérer les alternatives
- Très utilisés!
  - Peuvent aussi servir pour les Tests (abordés plus loin)

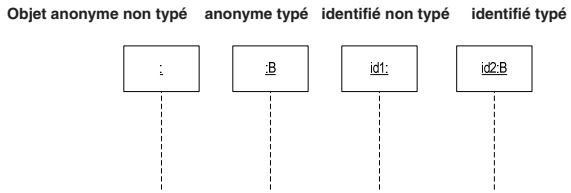
Exemple

- Un diagramme de séquence a deux dimensions :
  - L'axe horizontal représente les différents objets
  - L'axe vertical représente le temps



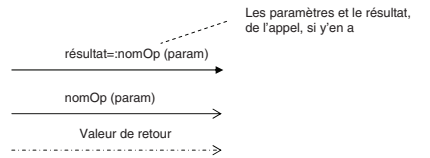
Diagrammes de Séquence: Constituants

- **Objets:** instances de classes qui participent à l'interaction
- Les objets sont représentés avec une ligne de vie

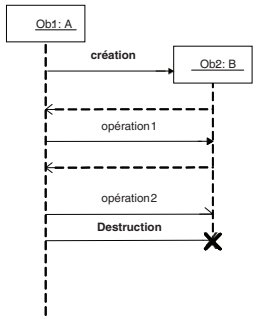


Diagrammes de Séquence: Constituants

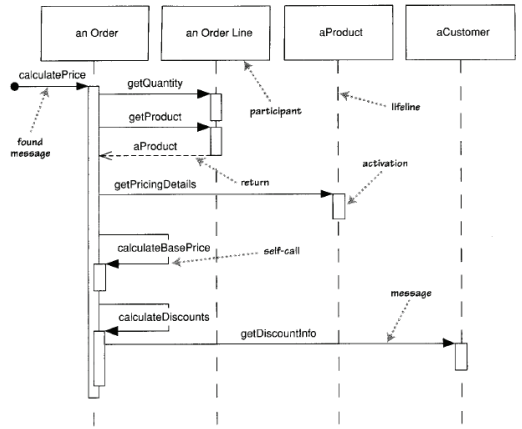
- **Messages:** un message est une communication entre objets pour communiquer une information ou bien déclencher une action
  - Message Simple (un Call)
  - Message Asynchrone
  - Message de Retour
    - Peut être omis



Diagrammes de Séquence: Messages de Création et de Destruction

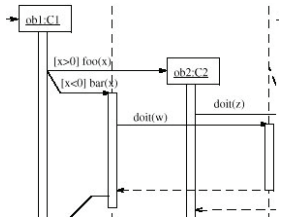


Un autre exemple, d'autres concepts



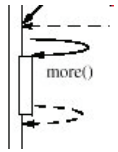
Diagrammes de Séquence: La bande d'activation

- Permet de voir quand l'objet est actif dans l'interaction



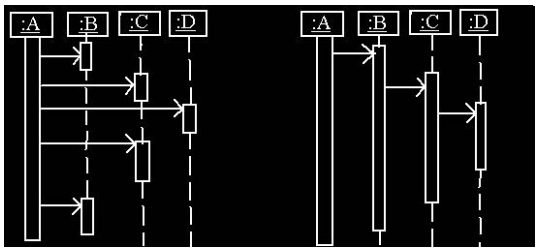
Diagrammes de Séquence: L'appel récursif

- L'objet peut appeler une opération chez lui (interaction interne)
  - Voir aussi la bande d'activation



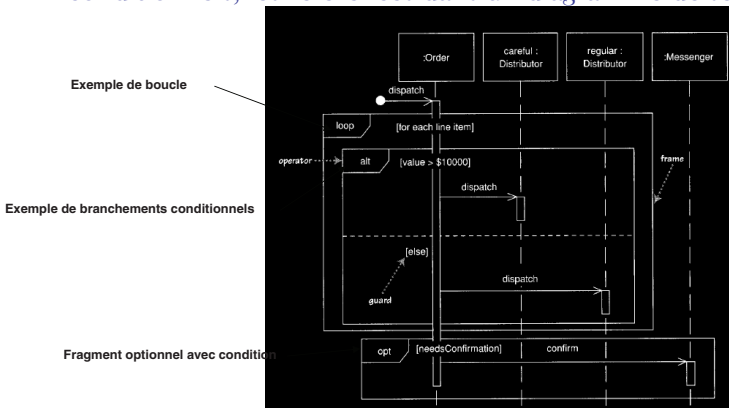
Diagrammes de Séquence: Le type de contrôle

- Les diagrammes de séquence permettent d'identifier aisément (visuellement) le type de contrôle
  - Contrôle centralisé Vs. Contrôle distribué



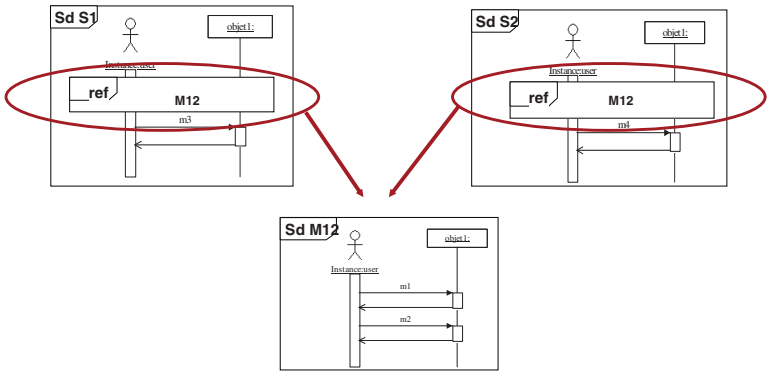
Diagrammes de Séquence: les fragments combinés

- Permettent de représenter les boucles, les branchements conditionnels, les références dans un diagramme de séquence



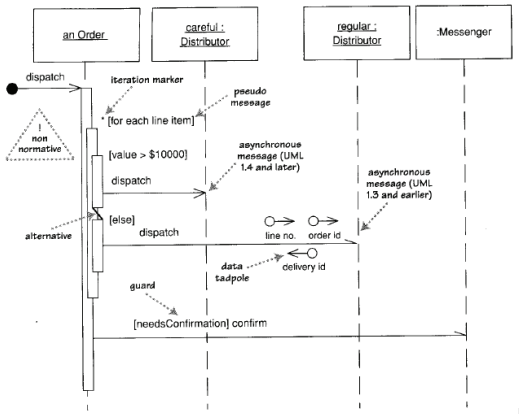
Diagrammes de Séquence: les fragments combinés

- Exemple de l'opérateur **ref : permet de référencer un diagramme de séquence à partir d'autres D.S**



Diagrammes de Séquences: L'ancienne notation UML 1.x

- Principalement, la notation concernant les boucles, les If, et les messages asynchrones



Diagrammes de Séquence: Conclusions

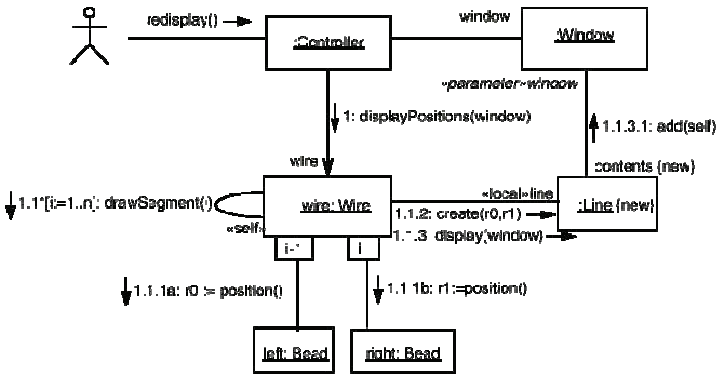
- Permettent de décrire la dynamique d'un système
- De formaliser les scénarios
- Mettent l'accent sur la dimension « Temps » dans l'interaction
- Permettent de réfléchir sur le type de contrôle + objets impliqués dans l'interaction (moins bien que le diagramme de collaboration)
- Permettent de faire le lien entre les diagrammes de cas d'utilisation et les diagrammes de classes

Diagramme de Collaboration

Diagrammes de Collaboration: En bref

- Les mêmes constituants qu'un diagramme de séquence
- L'accent est plus mis sur les objets impliqués dans l'interaction que sur l'aspect temporel des échanges de messages
- Peu utilisé
- Dans certains outils, ils sont générés automatiquement à partir des diagrammes de séquence

Diagrammes de Collaboration: Exemple

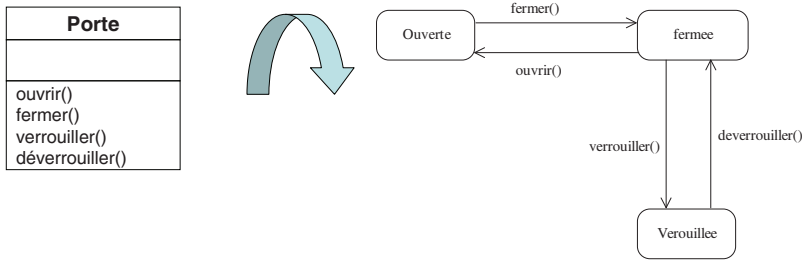


Diagrammes d'États/Transitions

Diagrammes d'États/Transitions

- Attachés à une classe
  - Généralisation des scénarios (tous les cas de figures)
  - Description systématique des réactions d'un objet aux changements de son environnement
  - Peuvent être utilisés pour décrire les états d'un système entier (lors des 1ères phases de dév. où les classes ne sont pas encore identifiées)
- Décrivent les séquences d'états d'un objet ou d'une opération :
  - En réponse aux «stimulis» reçus
  - En utilisant ses propres actions (transitions déclenchées)
- Réseau d'états et de transitions
  - Automates étendus
  - Essentiellement *Diagrammes de Harel (idem OMT)*

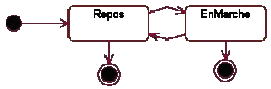
Diagrammes d'États/Transitions: Exemple



Diagrammes d'États/Transitions: Constituants

- **État:** Un état est caractérisé par une notion de durée et de stabilité. Il porte un nom

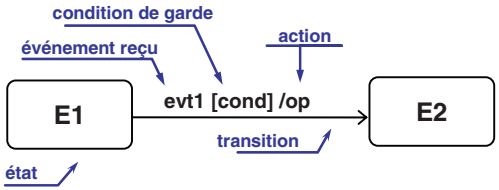
- Un automate comporte un et un seul état initial et un nombre quelconque d'états finaux



- **Transition**: Ce qui provoque le passage instantané de l'objet d'un état à un autre
  - Connexion unidirectionnelle
  - Déclenchée par un événement

Diagrammes d'États/Transitions: Syntaxe & Notation

- L'événement : "événement"
- déclenche l'action : "action"
- si la condition : "garde" est vérifiée.



Diagrammes d'États/Transitions: Événements

- Stimulis auxquels réagissent les objets
  - Occurrence déclenchant une transition d'état
- Abstraction d'une information instantanée échangée entre des objets et des acteurs
  - Un événement est instantané
  - Un événement correspond à une communication unidirectionnelle
  - Un objet peut réagir à certains événements lorsqu'il est dans certains états.
  - Un événement appartient à une *classe d'événements* (classe stéréotypée «signal»).

Diagrammes d'États/Transitions: Événements (types)

- Réalisation d'une condition arbitraire
  - transcrit par une condition de garde sur la transition
- Réception d'un signal issu d'un autre objet
  - transcrit en un événement déclenchant sur la transition
- Réception d'un appel d'opération par un objet
  - transcrit comme un événement déclenchant sur la transition
- Période de temps écoulée
  - transcrit comme une expression du temps sur la transition

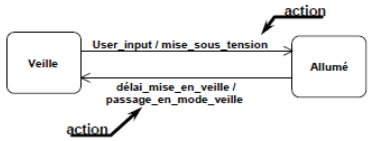
Diagrammes d'États/Transitions: Événements (Exemples)

- nomEvenement(listeParametre)
- nomParametre1 : Type, nomParametre2 : Type, ...
- Pour une durée : after(2 secondes)
- Pour une condition : when(var = 3)

Diagrammes d'États/Transitions: Action

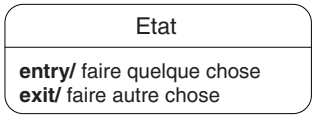
- **Action :** opération *instantanée* (conceptuellement) et *atomique* (ne peut être interrompue)

- Déclenchée par un événement
  - Traitement associé à la transition
  - Ou à l 'entrée dans un état ou à la sortie de cet état



Diagrammes d'États/Transitions: Action

- Actions à l'entrée et sortie de l'état
  - entry/ indique une action exécutée à chaque entrée dans l'état.
  - exit/ indique une action exécutée à chaque sortie de l'état.

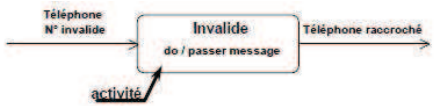


Diagrammes d'États/Transitions: Activité Interne

- Activité : opération se déroulant continuellement tant que l'objet est dans l 'état associé
  - do/ *action*

- Lien directe entre diagramme d'E/T et le diagramme d'activités

- Une activité peut être interrompue par un événement.

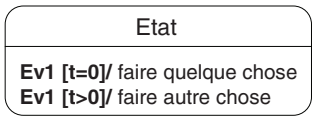


Diagrammes d'États/Transitions: Transition Interne

- L'objet ne change pas d'état

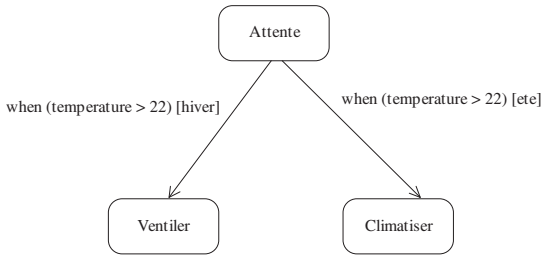
- Une transition interne est déclenchée à chaque occurrence de l'événement spécifié (dans l'état)

- Elle est différente d'une transition réflexive
  - dans une transition réflexive, il y a sortie puis entrée dans l'état



Diagrammes d'États/Transitions: Garde

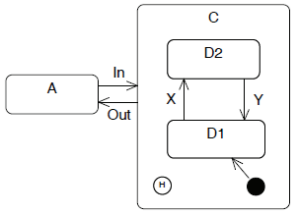
- Une condition qui valide ou non le déclenchement d'une transition lors de l'occurrence d'un événement



Diagrammes d'États/Transitions: Concepts avancés

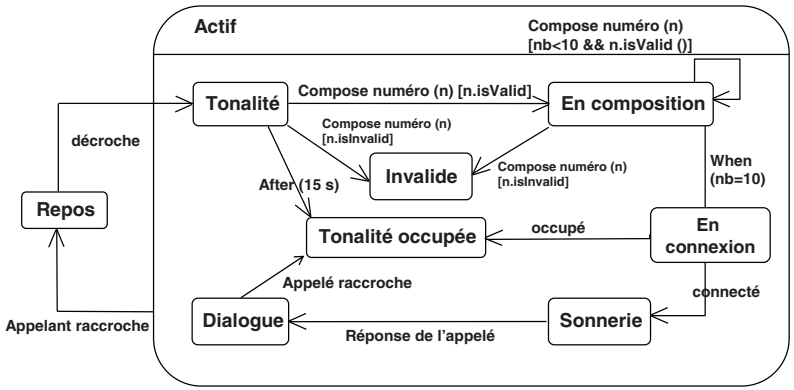
Diagrammes d'États/Transitions: État composé

- Un état peut être décomposé en sous-états
  - Pour simplifier la compréhension du problème
  - Factoriser des transitions



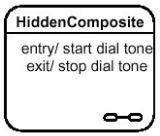
Diagrammes d'États/Transitions: État composé

- L'exemple du téléphone



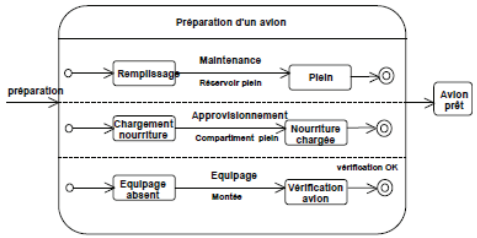
Diagrammes d'États/Transitions: État composé

- Notion de souche: masque les détails



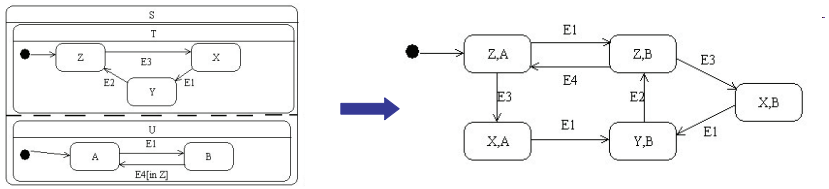
Diagrammes d'États/Transitions: État composé & Concurrents

- Utilisation de sous-états concurrents pour ne pas à avoir à expliciter le produit cartésien d 'automates
  - si 2 ou plus aspects de l 'état d 'un objet sont indépendants
  - Activités parallèles
- Sous-états concurrents séparés par pointillés
  - « *swim lanes* »



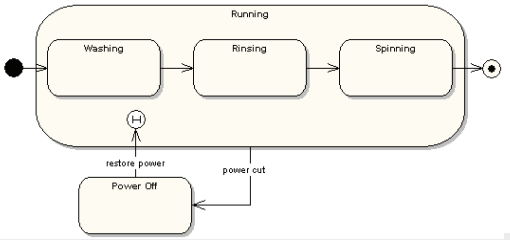
Diagrammes d'États/Transitions: État composé & Concurrents

- Autre exemple



Diagrammes d'États/Transitions: état historique

- Par défaut, un état composite ne retient pas le sous-état dans lequel il était à sa dernière sortie
- **Avec un indicateur d'historique, on peut revenir dans le sous-état de la sortie précédente**
- En ajoutant une "*" à l'indicateur d'historique, on applique l'indicateur aux états imbriqués



Diagrammes d'États/Transitions: Transition Composée

- Le choix de la transition est fait avant d'atteindre le point de branchement

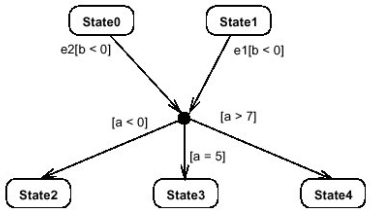


Diagramme d'Activités

Diagramme d'Activités

- **Souvent utilisés pour modéliser les Workflow ou Business Process**
- Pour modéliser les traitements effectués par une opération (corps)
  - Description d'un flot de contrôle procédural
- Attachés à
  - une classe,
  - une opération,
  - ou un *use-case (workflow)*
- L'un des diagrammes les moins utilisés
- Changement radical entre UML1.x et UML2.0

Diagramme d'Activités: Concepts de base

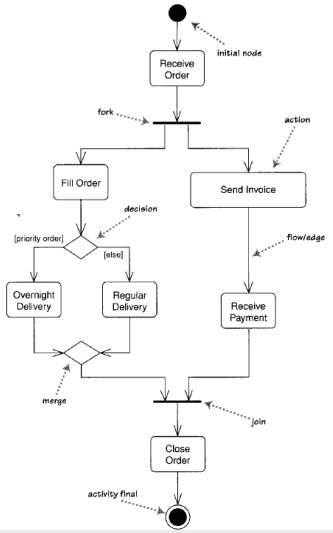


Diagramme d'Activités

- Attention dans UML 2.0 on ne parle plus d'activité comme unité à grain fin mais d'Action (une action est atomique)
- Une Activité est composée:
  - d'actions,
  - De nœuds de contrôle (les fork, merge, decision, join),
  - De nœuds d'objets (input et output pins, object nodes, ActivityParameterNode)
  - De flèches (Control Flow et Object Flow)
- Le principe de parcours du graph d'activités et de la gestion de ressources est basé sur la production et la consommation de Token (comme dans réseaux de Petri)
- Les transitions sont automatiques

Diagramme d'Activités: Des Activités et des Paramètres

- Comme pour les opérations, une activité pour avoir des paramètres en entrée et des paramètres en sortie

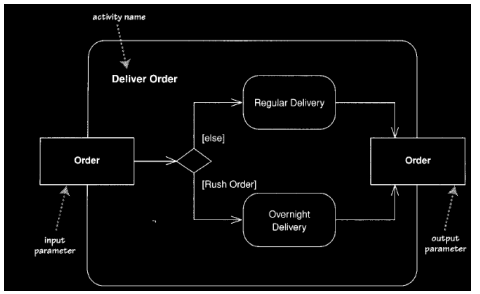


Diagramme d'Activités: Les Call Actions

- Une activité peut appeler une autre activité (CallBehaviorAction)
- Une activité peut appeler une opération (CallOperationAction)

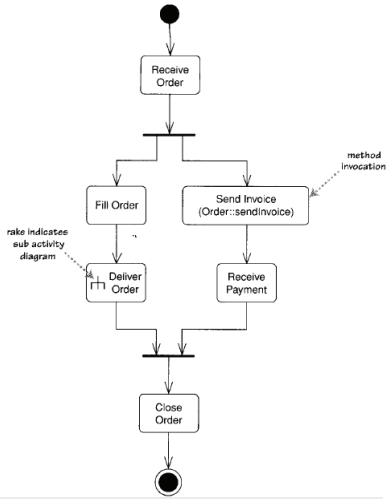


Diagramme d'Activités: Les Signaux

- Ça peut être suite à l'épuisement d'une durée
- À l'envoi explicite d'un signal
  - Catché par une AcceptEventAction

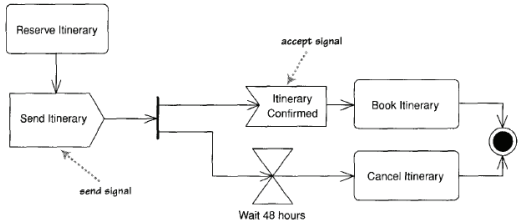


Diagramme d'Activités: Les Swim Lanes

- Possibilité d'utiliser les swim lanes comme dans UML 1.x

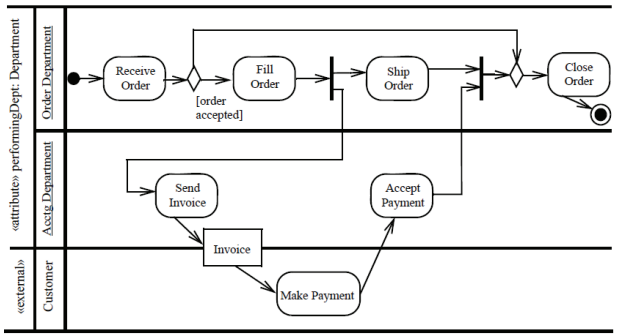


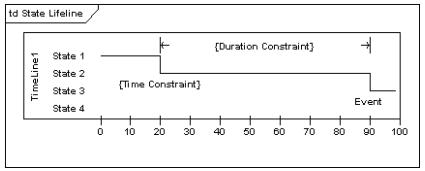
Diagramme d'Activités: Pas plus loin

- Les diagrammes d'activités dans la nouvelle version du standard sont très complets et très riches
- Une spécification large et complexe qui ne sera pas abordée dans ce cours
  - Le standard reste le document le plus complet concernant les DA
- Sensé être utilisé massivement dans l'approche MDE pour modéliser le corps des méthodes
  - Malheureusement trop complexe
  - Trop d'ambigüités dans la spéc.
  - Aucun outilleur n'arrive à fournir une implémentation complète de la spéc.
  - Le code reste plus lisible qu'un énorme diagramme d'activité

Diagramme de Temps

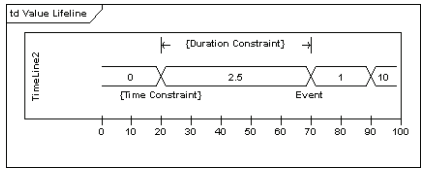
Diagrammes de Temps

- Nouveauté dans UML 2.0
- Les diagrammes temps d'UML sont utilisés pour afficher le changement d'état ou de valeur d'un ou plusieurs éléments à travers le temps
- Exemple de changement d'état à travers le temps/événement



Diagrammes de Temps

- Exemple de changement de valeur à travers le temps (x)



UML: Point de vue Déploiement

-Diagramme de Composants
-Diagramme de Déploiement

Diagramme de Composants

Diagramme de Composants

- « A component is a self contained unit that encapsulates the state and behavior of a number of classifiers » [UML 2.0, OMG]
- Débat controversé de ce qu'est un composant

"Components are not a technology. Technology people seem to find this hard to understand. Components are about how customers want to relate to software. They want to be able to buy their software a piece at a time, and to be able to upgrade it just like they can upgrade their stereo . They want new pieces to work seamlessly with their old pieces, and to be able to upgrade an their own schedule, not the manufacturer's schedule . They want to be able to mix and match pieces from various manufacturers. This is a very reasonable requirement. It is just hard to satisfy". Ralph Johnson

- Le diagramme de composants sert à montrer le lien entre les différents composants d'une application

Diagramme de Composants: Notation

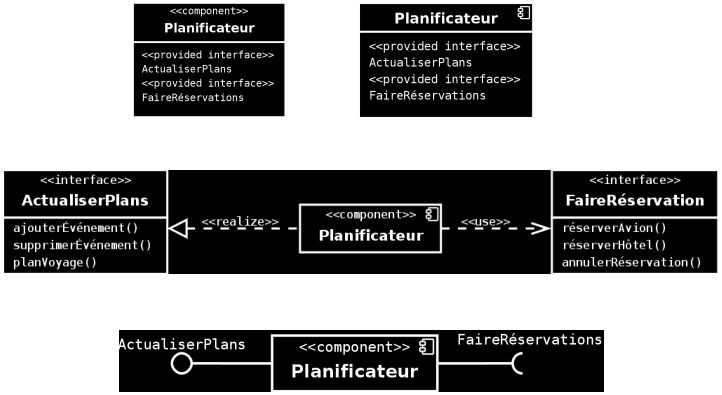
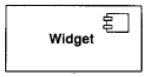
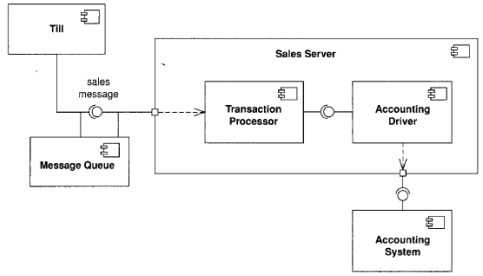


Diagramme de Composants: Exemple



Notation UML 2.0



Un Exemple de diagramme de Composants

Diagramme de Déploiement

Diagramme de Déploiement

- Montrent la configuration des éléments intervenants à l'exécution
  - Eléments physiques
  - Eléments logiciels
- Utile pour réfléchir sur l'architecture physique (distributions, moyens, performances, etc.)

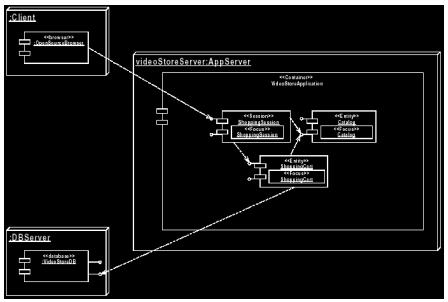
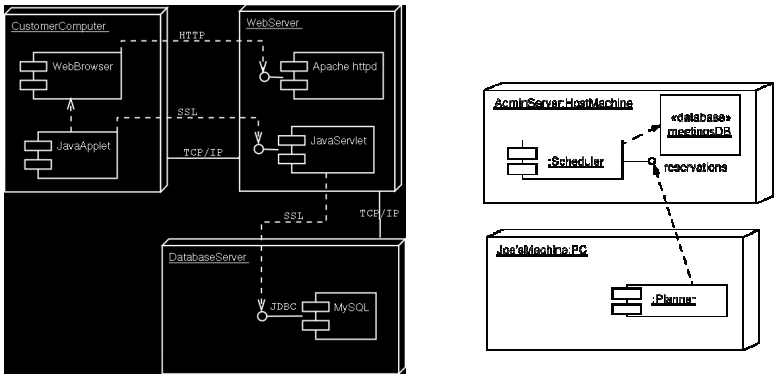
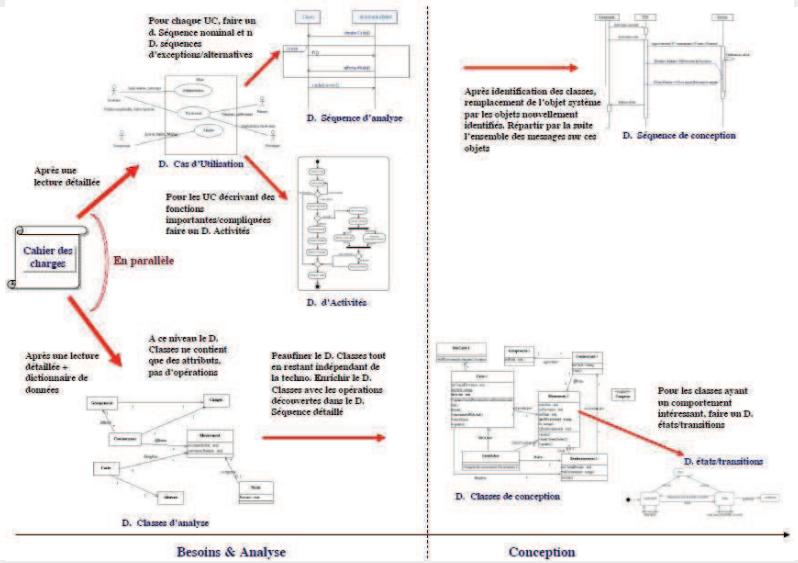


Diagramme de Déploiement : Exemples



Méthode de Développement UML: UML-P6

Processus & diagrammes



Processus & diagrammes

- Le processus adopté dans ce cours est celui présenté avant
- Ça ne représente qu'un processus parmi d'autres
- Ne couvre pas toutes les phases du développement
  - Réalisation: génération de code (prévu plus loin dans ce cours)
  - Validation: nous présenterons une approche avec des jeux de tests en utilisant les diagrammes de séquences et Junit (prévu plus loin dans ce cours)
  - Déploiement: (utilisation du diagramme de déploiement et du diagramme de composants)
  - Maintenance et amélioration du design: (prévu plus loin dans ce cours)
    - Design Pattern
    - Gestion des dépendances

Lectures

- Software Engineering,
  - Ian Sommerville, Addison Wesley; 8 edition (15 Jun 2006), ISBN-10: 0321313798
- The Mythical Man-Month
  - Frederick P. Brooks JR., Addison-Wesley, 1995
- Cours de Software Engineering du Prof. Bertrand Meyer à cette @:
  - http://se.ethz.ch/teaching/ss2007/252-0204-00/lecture.html
- Cours d'Antoine Beugnard à cette @:
  - http://public.enst-bretagne.fr/~beugnard/

- UML Distilled 3rd édition, a brief guide to the standard object modeling language
  - Martin Fowler, Addison-Wesley Object Technology Series, 2003, ISBN-10: 0321193687
- UML2 pour les développeurs, cours avec exercices et corrigés
  - Xavier Blanc, Isabelle Mounier et Cédric Besse, Edition Eyrolles, 2006, ISBN-2-212-12029-X
- UML 2 par la pratique, études de cas et exercices corrigés,
  - Pascal Roques, 6ème édition, Edition Eyrolles, 2008
- Cours très intéressant du Prof. Jean-Marc Jézéquel à cette @:
  - http://www.irisa.fr/prive/jezequel/enseignement/PolyUML/poly.pdf
- La page de l'OMG dédiée à UML: http://www.uml.org/
- Cours de Laurent Audibert sur http://laurent-audibert.developpez.com/Cours-UML/html/Cours-UML.html

---

- Design patterns. Catalogue des modèles de conception réutilisables
  - Richard Helm (Auteur), Ralph Johnson (Auteur), John Vlissides (Auteur), Eric Gamma (Auteur), Vuibert informatique (5 juillet 1999), ISBN-10: 2711786447