

Plan du cours

- ❑ Introduction au Génie Logiciel:
 - Quelques chiffres et faits
 - Modèles de développement des logiciels (*cycle de vie*)
- ❑ Analyse avec UML
 - Diagrammes de classes et autres diagrammes
 - Compléments aux diagrammes
 - Sémantique formelle en OCL et JML
- ❑ De l'analyse à la conception (abstraite, en UML)
 - Passage du modèle aux interfaces de classes Java
- ❑ Test de programmes
- ❑ Preuve de programmes: quand le test ne suffit pas !

Mise en œuvre associée

- ❑ 10 cours, 11 TD
- ❑ Contrôle des connaissances
 - 1 partiel + 1 examen final
- ❑ **TER associé** : réalisation d'un logiciel
 - En équipe, en parallèle avec le reste de vos activités
Avec un découpage en phases défini à l'avance et imposé,
Avec un ensemble de documents à fournir
Notation basée sur le logiciel final, **mais aussi** sur
 - le respect des phases,
 - la qualité (syntaxique et sémantique) des documents,
 - le respect des délais et des rendez-vous,
 - la qualité de l'interaction avec le « client »
 - **Une note à mi-parcours + une note finale**
 - **Les notes sont individualisées !**

Bibliographie et Weberies



- ❑ **UML 2.0**, Martin Fowler, Campus Press, 2004
- ❑ **Developing Applications with Java and UML**, Paul R. Reed Jr., Addison Wesley, 2002
- ❑ **UML 2 et les Design Patterns**, G. Larman, Campus Press, 2005
- ❑ **UML 2 en action**, P. Roques, F. Vallée, Eyrolles 2004
- ❑ **The Unified Modeling Language User Guide**, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison-Wesley, 2005
- ❑ **Précis de Génie Logiciel**, M.-C. Gaudel, B. Marre, F. Schlienger et G. Bernot, Masson, 1996
- ❑ **The Science of Programming**, D. Gries, Springer Verlag, 1981

http://www.omg.org/gettingstarted/what_is_uml.htm

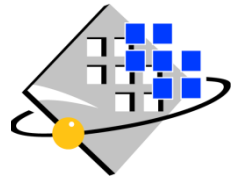
<http://www.eecs.ucf.edu/~leavens/JML/>

<http://www.junit.org/>

Remerciements à: M.-C. Gaudel, F. Schlienger, F. Fichot, S. Cohen-Boulakia



UNIVERSITÉ
PARIS-SUD 11



Partie I.1 : Introduction au Génie Logiciel

Pourquoi le Génie Logiciel ?

Une approche empirique du développement ne suffit pas :

- Logiciels de plus en plus complexes, volumineux
 - Contraintes de fiabilité, de sûreté, de sécurité,
 - Mélange matériel, logiciel, automatismes
 - Informatique, omniprésente et « diffuse »
- Aspects techniques
- *Présence d'un **client**, aux besoins en partie imprécis et changeants, avec qui il faut **pouvoir communiquer***
 - Développement en équipe, dans la durée
 - Logiciels avec une longue durée de vie, donc besoins en *maintenance => **documents de référence***
- Aspects managériaux

Mauvaise image de marque, manque de crédibilité
=> frein au développement

Des enjeux de plus en plus importants:

- ❑ Fiabilité, Sûreté et Sécurité des logiciels :
 - Transports terrestres (RER, Métros, TGV), aériens, espace,
 - Contrôle de processus industriels, nucléaire, armement, ...
 - Médical : télé-chirurgie, imagerie médicale, appareillage...
 - Téléphonie, commerce électronique, carte « Vitale » et autres cartes à puces
- ❑ Économique: coût et délai de développement de logiciels **fiables, répondant aux besoins**, acceptés par la tutelle et le public...
- ❑ Techniques et outils évoluent :
en 20 ans: gain d'un facteur 100 (?) sur la densité de fautes résiduelles
... mais la complexité des problèmes, les impératifs de disponibilité, de sûreté et sécurité aussi !
- ❑ **Aspects techniques, méthodologiques, managériaux**, comme pour toute activité industrielle

Quelques paniques célèbres

- **Mariner 1** (Venus, 1962) : mauvaise transcription d'une formule ⇒ corrections erratiques de trajectoire au lancement (destruction en vol)
- **1981** : Ajournement du premier lancement de la navette spatiale (problème de synchronisation des résultats de calculateurs redondants)
- **1985 - 1987** : Surdosage en radiothérapie (Therac-25, 5 décès). **Idem 2007**
- **1992** : Crash du système d'appel des ambulances londoniennes
- **1996** : Echec du vol 501 de Ariane 5 (problème de « validation » du système)
- **Juillet 1997** : Blocage de noms de domaines .com
- **Septembre 1999** : Perte de Mars Climate Orbiter après 9 mois de voyage. Coût : 120 M\$ (confusion d'unités entre deux équipes distinctes)
- **2004** : panne du système de réservation de billets SNCF aux guichets,
- **2004** : déni d'accès aux réseaux chez Bouygues Télécom, France Telecom

Des interrogations ?

Exemple (récent) d'interrogation : Contrôle de vitesse Renault Laguna (2005)

Est-ce qu'il y a un bogue informatique ou pas ?

Problème d'interaction matériel/logiciel/environnement ?

☹ *Même s'il n'y en a pas, personne n'est entièrement convaincu !*

- ☞ être capable de **justifier/contrôler son « processus de fabrication »**
- ☞ être capable de parler de la **qualité** des tests effectués et du produit final

Ubiquité du logiciel ?

Peugeot 607 = 2 Mo de logiciel embarqué, 40 calculateurs, 20 % coût total

« Besoins » (ABS, ESP, pédales électroniques, capteurs, ...) supplémentaires

- Interaction de services difficile à maîtriser. Trafic/Interférences sur les bus...
- Validation du système avec toutes les combinaisons d'options ?

Future automatisation des lignes de métro (RATP – ligne 1: 2010)

- Quel processus de validation ?
- Responsabilités entre RATP et industriels (SIEMENS)
- Qui autorise in fine la mise en service ? Au vu de quelle certitude ?

Quelques chiffres

❑ Taille des logiciels ?

- Windows « 90 » : 10 M. LOC source, Win2000: 30 M. LOC
- Noyau RedHat 7.1 (2002) : ~2.4 M. LOC, XWindow ~1.8 M., Mozilla ~2.1 M.
- Navette spatiale (et son environnement) : ~50 MLOC
- Station Columbus (abandonnée) : 980 M Lignes compilées ??

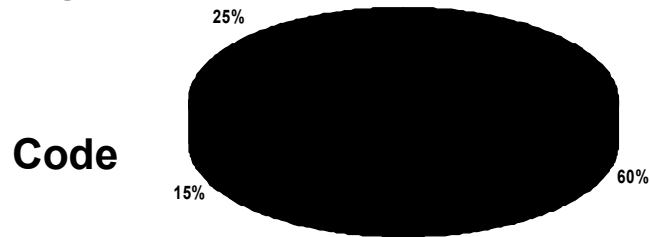
❑ Coût du développement ?

- **Proportion de la partie « Codage » : 15 à 20 %**
Codage de plus en plus automatisé (outils *CASE*)
- Proportion de la partie Validation et Vérification ? ~ 40%
- Proportion de la partie Spécification et Conception ? ~ 40%

❑ Estimation (Boehm) $MH = 3,5 KISL^{1,2}$ (très approximatif !)

Des chiffres sur les « défauts logiciels »

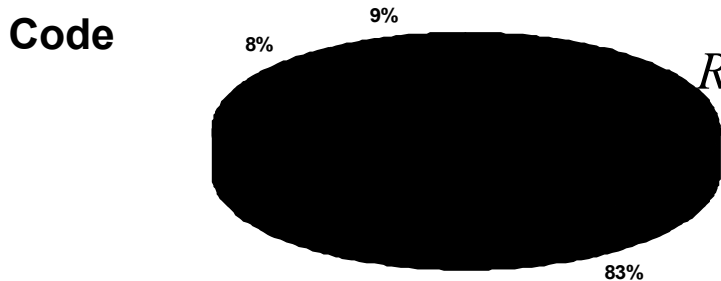
Algorithme



**Spécification et
conception**

Répartition du nombre de défauts

Algorithme



Répartition du coût de réparation des défauts

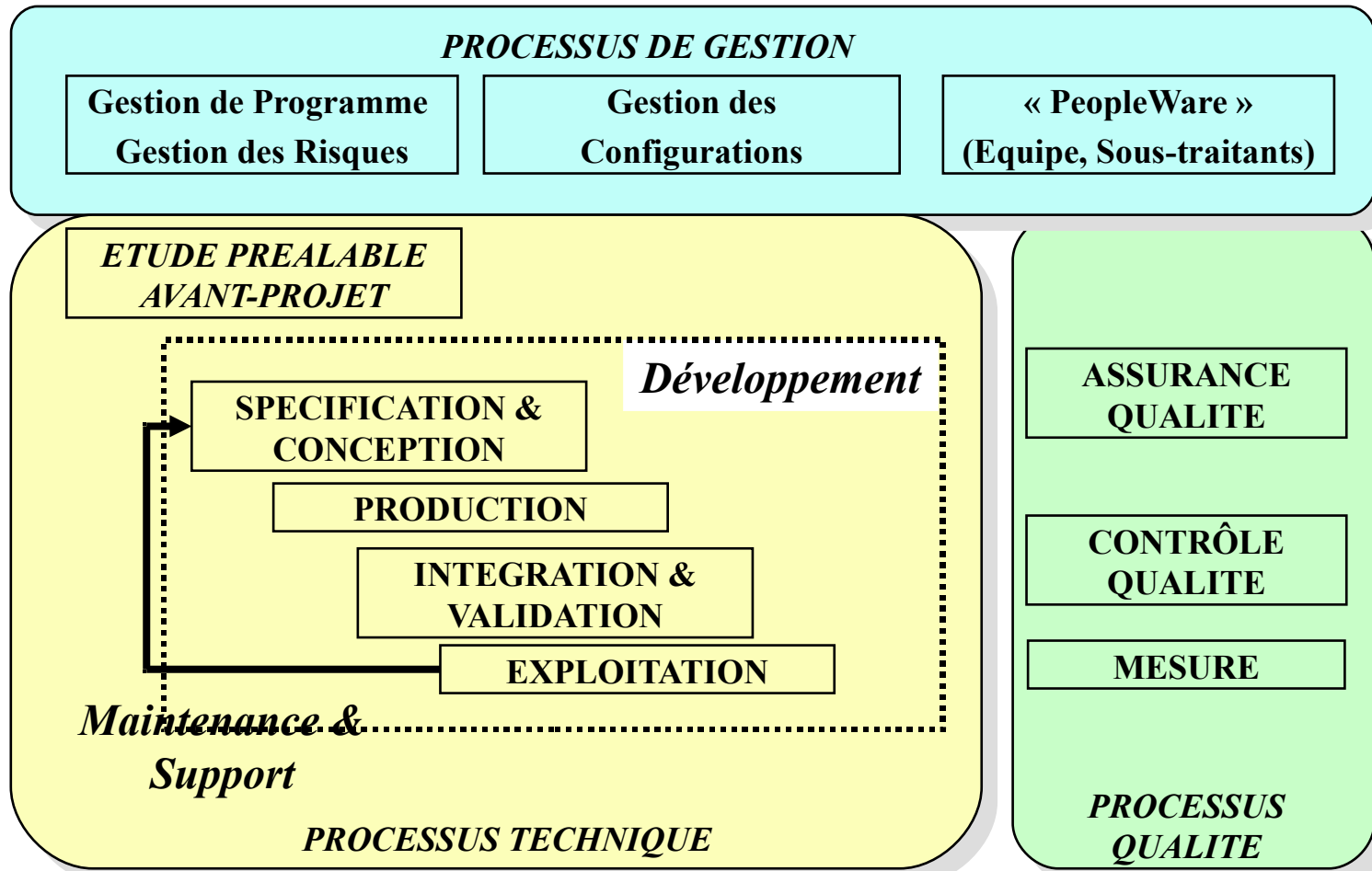
Spécification et conception

Source: F. Fichot, Cours de Conduite de Projets, IFIPS

Spécificités du logiciel

- ❑ Complexités différentes selon les applications :
 - Synchronisation, répartition des processus (parallélisme)
 - Nature ou volume des données, problèmes algorithmiques
 - Imbrication matériel et logiciel,
 - Performance ou « temps de réponse » (temps réel, systèmes réactifs)
 - ❑ Pas de « fabrication », pas de difficulté de duplication
 - ❑ Facilité apparente de modification (versatilité) mais difficulté à limiter les impacts d'une modification
 - ❑ Faible prédiction des taux de défaillances :
 - Pas de « modèle physique » des causes de pannes comme pour le matériel
 - Pas de « modèle logique » consensuel pour les fautes logicielles
 - ❑ Faible visibilité du processus de développement pour le management et le client
-
- ☞ *Importance des phases d'analyse des besoins, conception, validation*
 - ☞ *Besoin d'un processus de fabrication « industriel », traçable*
 - ☞ *Basé sur des modèles et des documents (dans une vision classique du G.L.)*

Les différents aspects du Génie Logiciel



Aspects organisationnels: la conduite de projets

Prévoir, organiser, faire le point, réagir !

- ❑ Estimation des coûts et des délais
- ❑ Planification (diagrammes PERT et GANTT)
- ❑ Suivi: gestion de l'équipe, des échéances et fournitures à remettre au client
- ❑ Gestion des risques (mesures préventives ou correctives)
- ❑ Gestion des configurations (gestion des sources, des tests et des documents associés aux différentes versions)
- ❑ Gestion des documents associés aux différentes phases de la production du logiciel
- ❑ ...

*☞ le tout dans le respect de normes qualité
(documents types, circuits de validation, règles de nommage)*

Le processus de développement logiciel

- ❑ Suite de descriptions de plus en plus précises et exécutables et documentations associées
 - Cahier des charges informel -> modèle(s) d'analyse -> modèle(s) de conception -> codage -> validation et vérification
- ❑ Développement en « étapes »
 - Chaque étape se termine par la production de documents qui sont vérifiés et validés avant le passage à l'étape suivante.
- ❑ Chaque étape contribue à un certain nombre d'« activités », certaines activités s'étalent sur plusieurs étapes
 - Documentation (manuel utilisateur, manuel de référence, manuel d'installation,...) : constituée ou précisée à différentes étapes.
 - Activités de Validation/Vérification : à chaque étape !

Activités et étapes (1)

- ❑ Étape de **capture des besoins**
Étude de l'environnement et du contexte d'utilisation, analyse des besoins et des contraintes (performances, ergonomie, portabilité, existant,...)
Rédaction du **cahier des charges**
- ❑ Étape de **analyse et spécification**
On décrit ce que le logiciel va faire (et pas comment il va le faire)
Écriture de différents *modèles*
- ❑ Étape de **conception architecturale**
Décomposition du logiciel en « composants »
Prévision de la manière et l'ordre d'intégration de ces différents composants (incréments); conception des tests associés à l'intégration
- ❑ Étape de **conception détaillée**
Choix des algorithmes, des structures de données
Définition de l'interface des classes à écrire
Conception des tests qui valideront les implémentations

Activités et étapes (2)

- ❑ Étape de **codage**
- ❑ Étape de **test unitaire**
Exécution contrôlée des tests de chaque procédure/méthode tels que définis préalablement
- ❑ Étape **d'intégration** et de **test d'intégration**
Exécution des tests prévus pour l'intégration
- ❑ **Test d'acceptation** et **systemes** (par le fournisseur)
Test global, en conditions réelles d'exploitation
- ❑ **Recette** (par le client)
Tests, inspection des documents requis, éventuellement vérification des normes qualités imposées

Au-delà des étapes...

- ❑ Importance **des** documentations :
publics distincts : utilisateur, concepteurs, exploitants,...
- ❑ Distinction entre deux activités complémentaires :
Validation: *vérifier qu'on construit le **bon** système (correction par rapport à la demande du client; cohérence « externe »)*
Vérification: *vérifie qu'on construit **bien** le système (correction par rapport à la spécification du système; cohérence « interne »)*
- ❑ Importance de la **traçabilité** !
 - **Documenter** et archiver les **choix** effectués
 - Garantir la cohérence entre les modèles et le produit construit (environnements de « *forward/reverse engineering* »)
- ❑ Importance de **l'archivage** et **gestion des configurations** (gestion des versions de documents, gestion des code sources)
 - Garantir la cohérence des documents avec les versions

Et après la mise en service ?

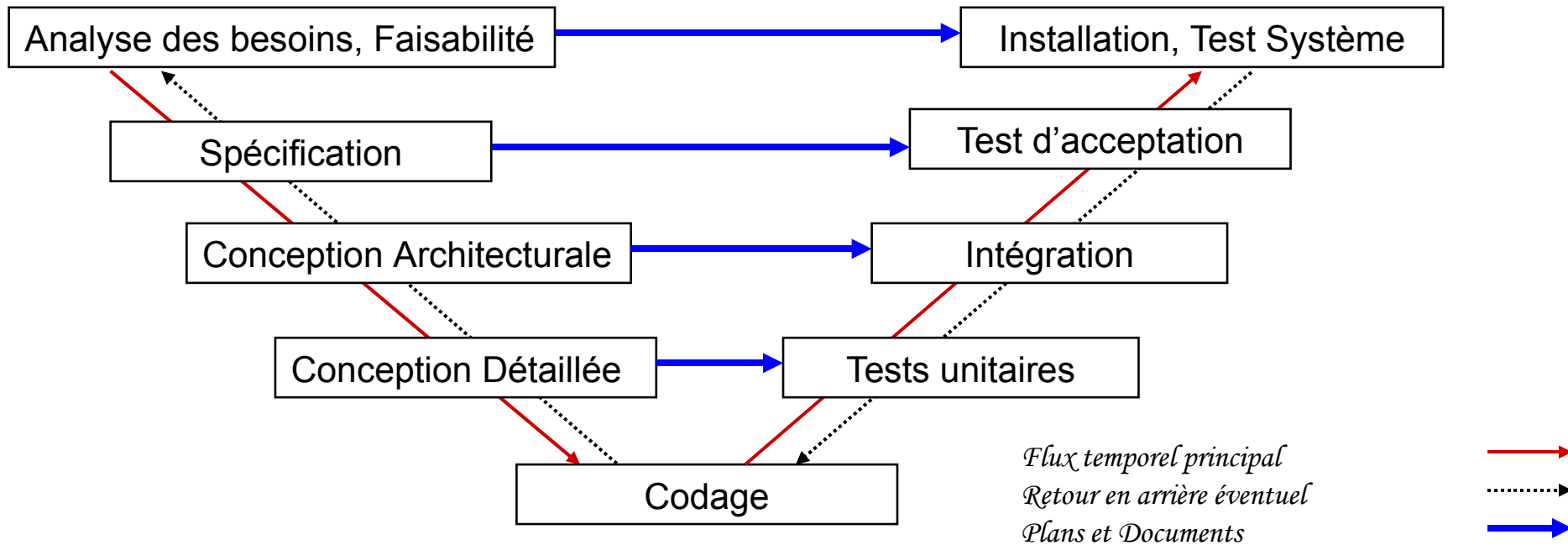
- ❑ Maintenance douloureuse et coûteuse !
 - Maintenance *corrective* : correction des bogues
 - Maintenance *adaptative*: version d'OS, de matériel à supporter simultanément, problèmes de performances
 - Maintenance *évolutive*: évolutions des fonctionnalités
- D'autant plus douloureux qu'il peut y avoir un ensemble de versions à maintenir simultanément !
- ❑ Coût Maintenance: **2 à 4 fois** le coût de développement ?
- ❑ Besoin de **test de non-régression** d'une version à l'autre :
 - Description précise des tests (entrées, sorties attendues, contexte d'exécution)
 - Besoin d'automatisation des tests (exécution, dépouillement, archivage) et donc d'outils

Modèles de développement du logiciel...

- ❑ Organisation des étapes et des documents associés
 - Structurer le processus et améliorer sa visibilité,
 - Prévoir des jalons, des documents associés
 - Minimiser les risques
 - Aspect non linéaire du développement (modification dans les besoins, environnement technologique mouvant,...) ?

- ❑ *Méthodologies adaptées à des tendances générales ?*
 - *Aspects « répartis » de l'informatique*
 - *Souci d'interopérabilité (SOAP, WebServices, XML, ...)*
 - *Approches « Composants » (réutilisation, « COTS »)*
 - *Séparation logique métier / services techniques
(Normes autour des spécifications J2EE et .Net, WebServices)*

Le Modèle en V...

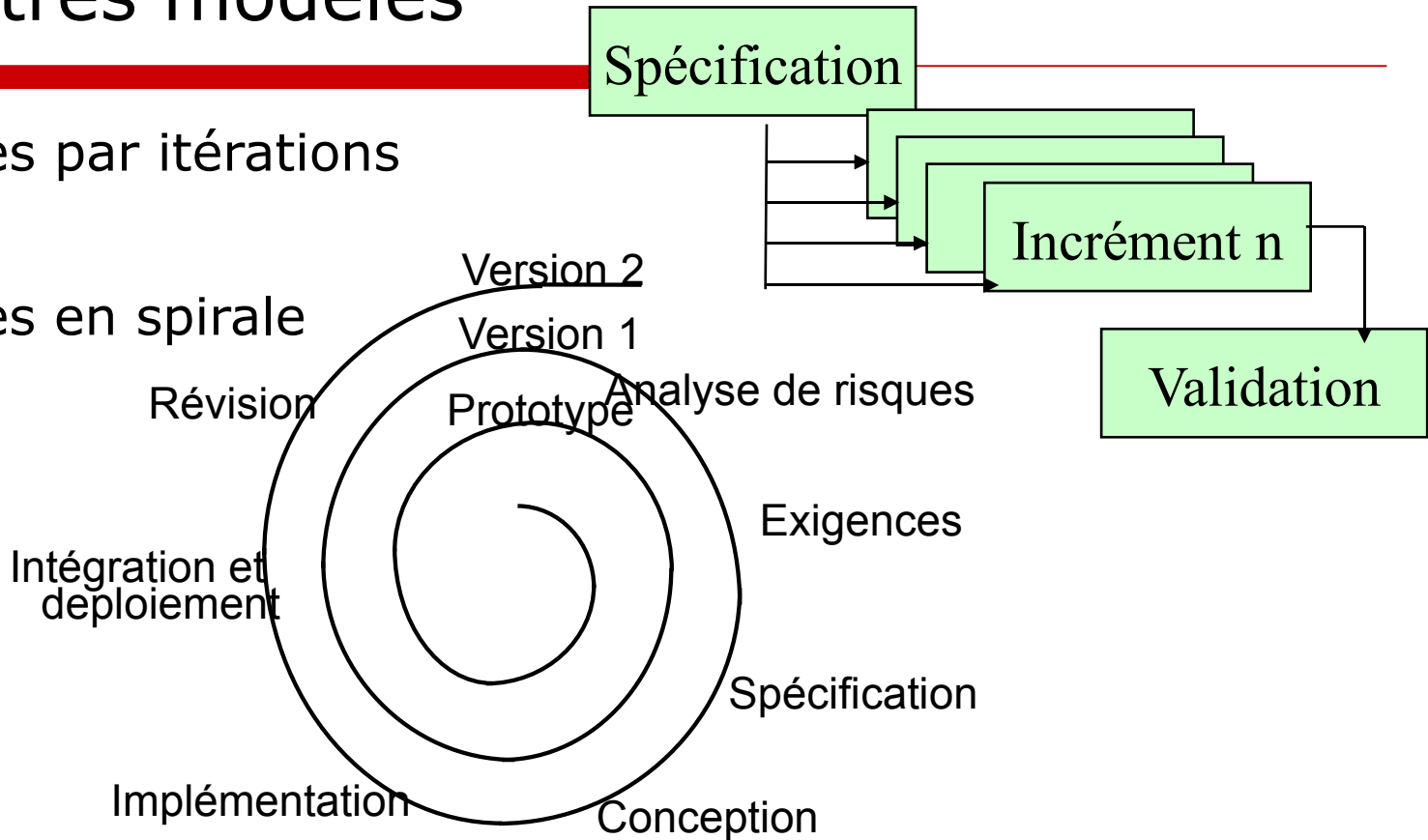


- ❑ Historiquement le plus connu, le plus pédagogique !
- ❑ Le flux descendant produit les guides pour les phases montantes
- ❑ Pédagogique, mais pas toujours adapté...(global, linéaire)

D'autres modèles

❑ Modèles par itérations

❑ Modèles en spirale



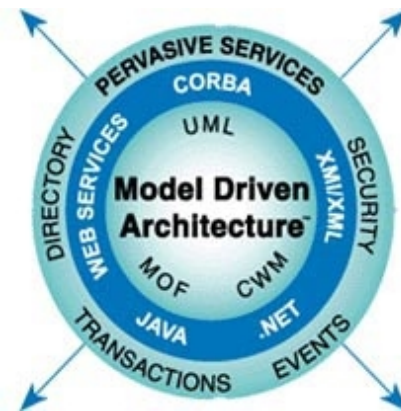
Principes généraux:

Diminuer les risques, meilleure visibilité, retours utilisateur plus rapides,

Intégration et validation progressive des fonctionnalités

Les modèles en vogue

- ❑ Xtreme Programming: (http://www.en.wikipedia.org/wiki/Extreme_programming)
 - Développement dirigé par les tests
 - Restructuration permanente du code, codage par « pairs »
 - Présence « permanente » du client
- ❑ **Rational Unified Process** (RUP): itératif, utilise UML.
<http://www.idbconsulting.com/francais/incIBMRationalRUP.cfm>
http://en.wikipedia.org/wiki/Rational_Unified_Process
- ❑ Architecture MDA
(<http://www.omg.org/mda/>)
Approche par « transformation de modèles »



Partie I.2 : U.M.L. en un clin d'œil...

Pourquoi UML ?

Survol d'UML en quelques transparents

Environnements UML



Pourquoi UML ?

- ❑ les méthodes orientées objets sont actuellement en vogue
UML est bien adapté (classes, interfaces, héritage, ...)
- ❑ Pas forcément associé à la programmation objet, peut servir dans un contexte plus large:
*Description des aspects **statique** (représentation des données) et **dynamiques** (comportements + évènements)*
- ❑ Notation devenue un **standard de fait**
Synthèse de notations antérieures (OMT, OOA/OOD, OOSE, ...)

Définie et maintenue par l'OMG (<http://www.omg.org>)

- Consortium de spécification (IBM, Sun, Oracle, HP, Boeing, BEA,...)
- À la base de nombreux standards (CORBA, SOAP, MDA, ...)

UML: Version 1.1 en 1997, version 2.0 en 2005.

UML = Unified Modeling Language

- ❑ Permet d'établir divers **modèles** du système à réaliser
- ❑ Fournit des **notations** (semi-graphiques) : diagrammes...
 - Les diagrammes peuvent être enrichis par des textes
- ❑ UML ne fournit pas le mode d'emploi ☹
- ❑ Quel usage ?
 - Graphique ⇒ outil de communication possible avec les clients
 - Modèles partagés par l'équipe de développement
 - Plusieurs modèles selon les étapes, dans la même notation
- ❑ UML se veut
 - Universel (limité à une notion discrète du temps), donc gros ;-(
 - Extensible (via des « stéréotypes » et des « profiles »)
 - Couvre le développement de l'analyse à la conception
- ❑ UML a une sémantique officielle



Qu'est-ce que UML ?

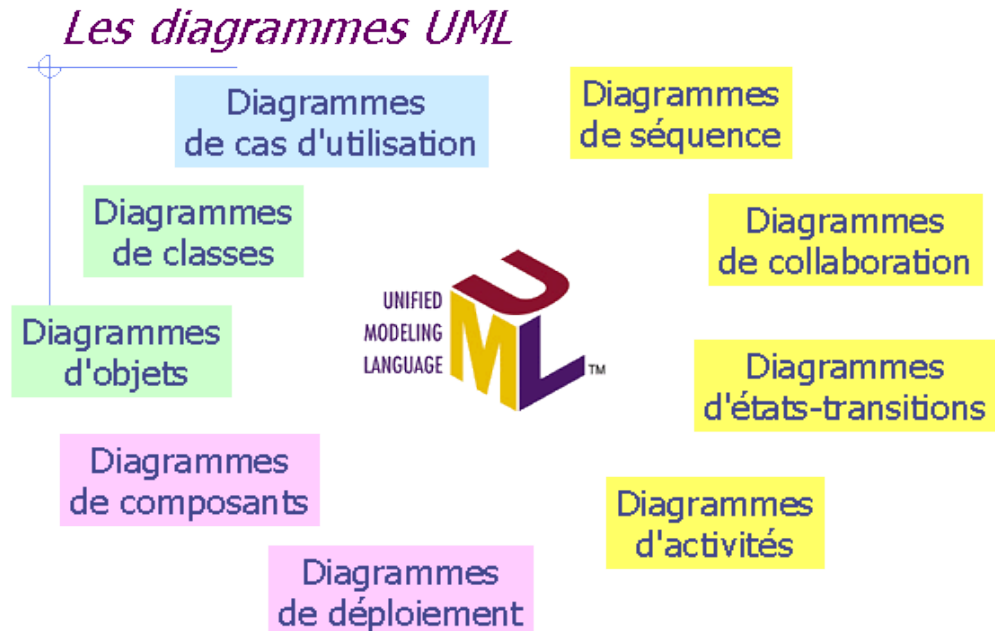
- ❑ UML, Version 1.1 : 9 types de diagrammes

3 vues différentes :

Architecturale

Statique

Dynamique



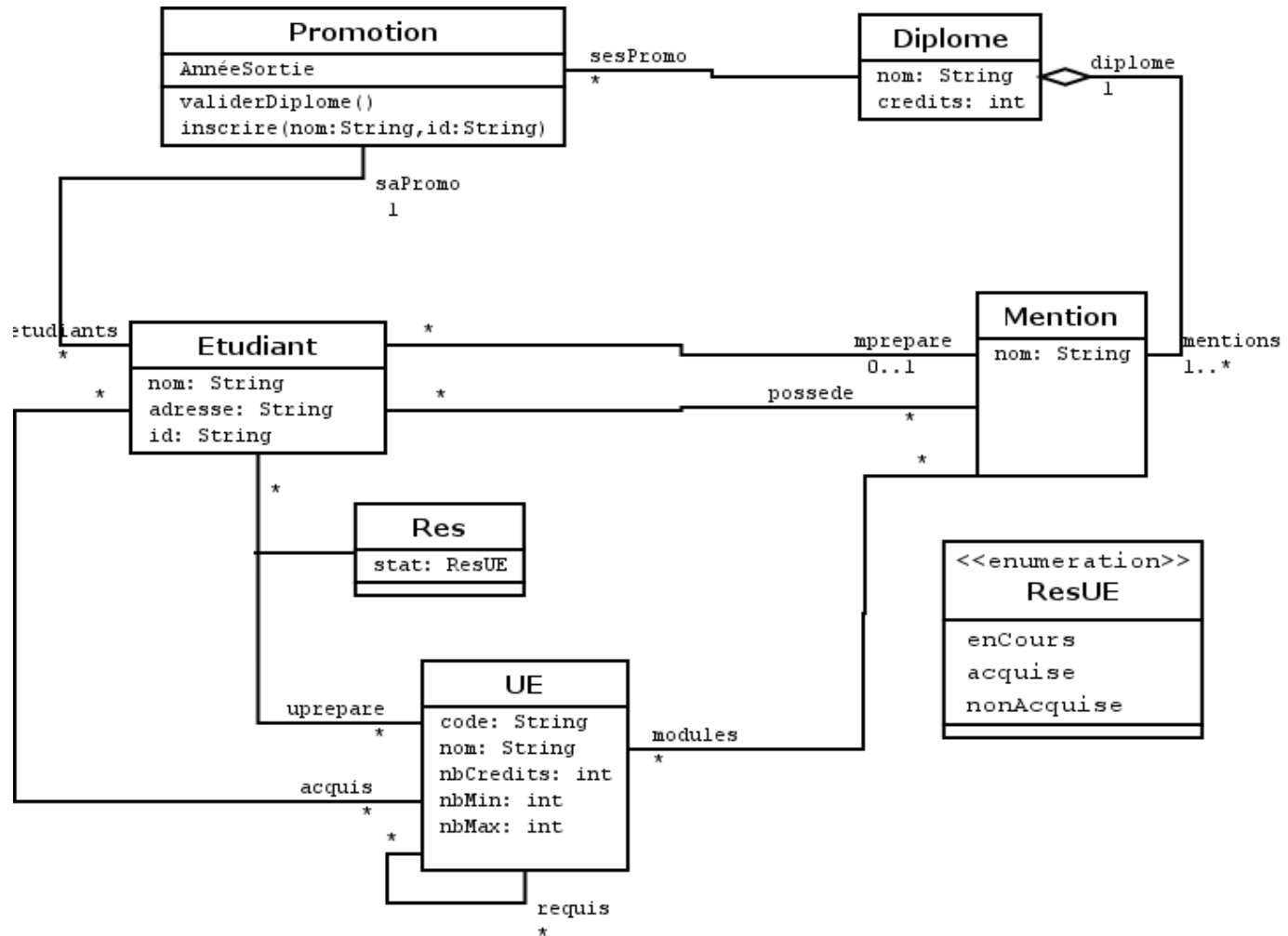
- ❑ UML Version 2.0 ajoute 4 types de diagrammes :
Diagrammes de structure composite, de communication,
de paquetages, de contraintes temporelles (« timing »)

Les principaux diagrammes UML (1)

- ❑ *Diagrammes de classes* : la **structure statique** du système
 - les classes d'intérêt : ce qu'on représente dans le système
 - les relations entre classes
 - Les attributs et les méthodes
 - les types, interfaces requises/définies ...

- ❑ *Diagrammes des cas d'utilisation* : la **partie dynamique** (les fonctions) du système
 - Les interactions du système avec le monde extérieur (agents extérieurs qui communiquent avec le système)
 - Les cas principaux, les alternatives, les extensions

Un exemple de diagramme de classes

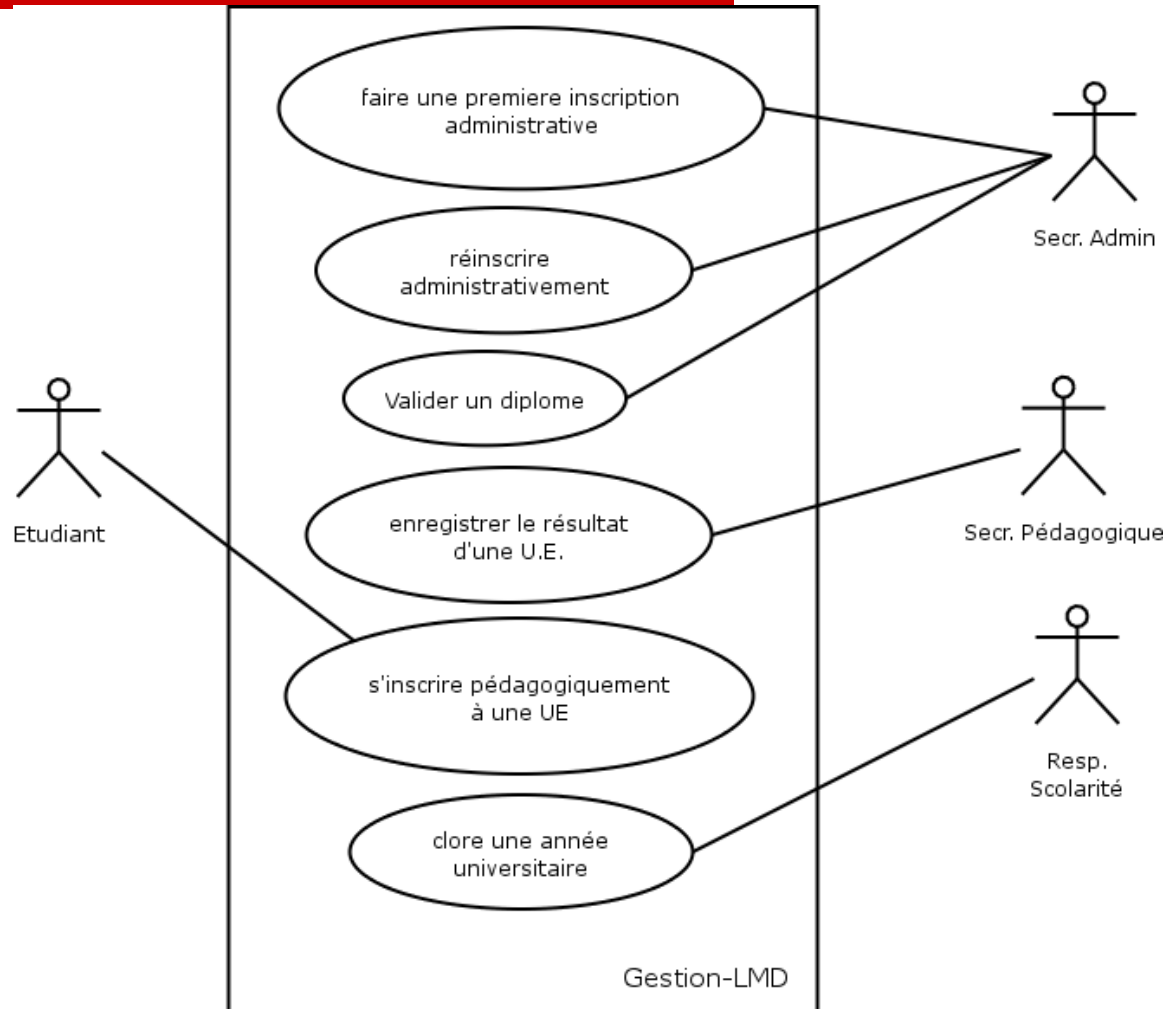


A propos de ce diagramme de classes

- ❑ Il existe souvent des contraintes du domaine qu'on ne peut pas exprimer dans le diagramme (« règles métiers » ou « invariants »)
 - Les numéros étudiants sont tous distincts
 - Un étudiant ne peut pas suivre un module qu'il a déjà acquis
 - Un module ne peut pas faire partie, directement ou non, de ses pré-requis
 - Un étudiant ne peut suivre un module que s'il possède les modules pré-requis
 - Il existe des règles qui limitent sur la variation du nombre de crédits de chaque module d'une même filière
 - etc.

- ☞ Il faudra d'autres mécanismes pour décrire ces contraintes !

Un exemple de « Diagramme des Cas d'Utilisation »

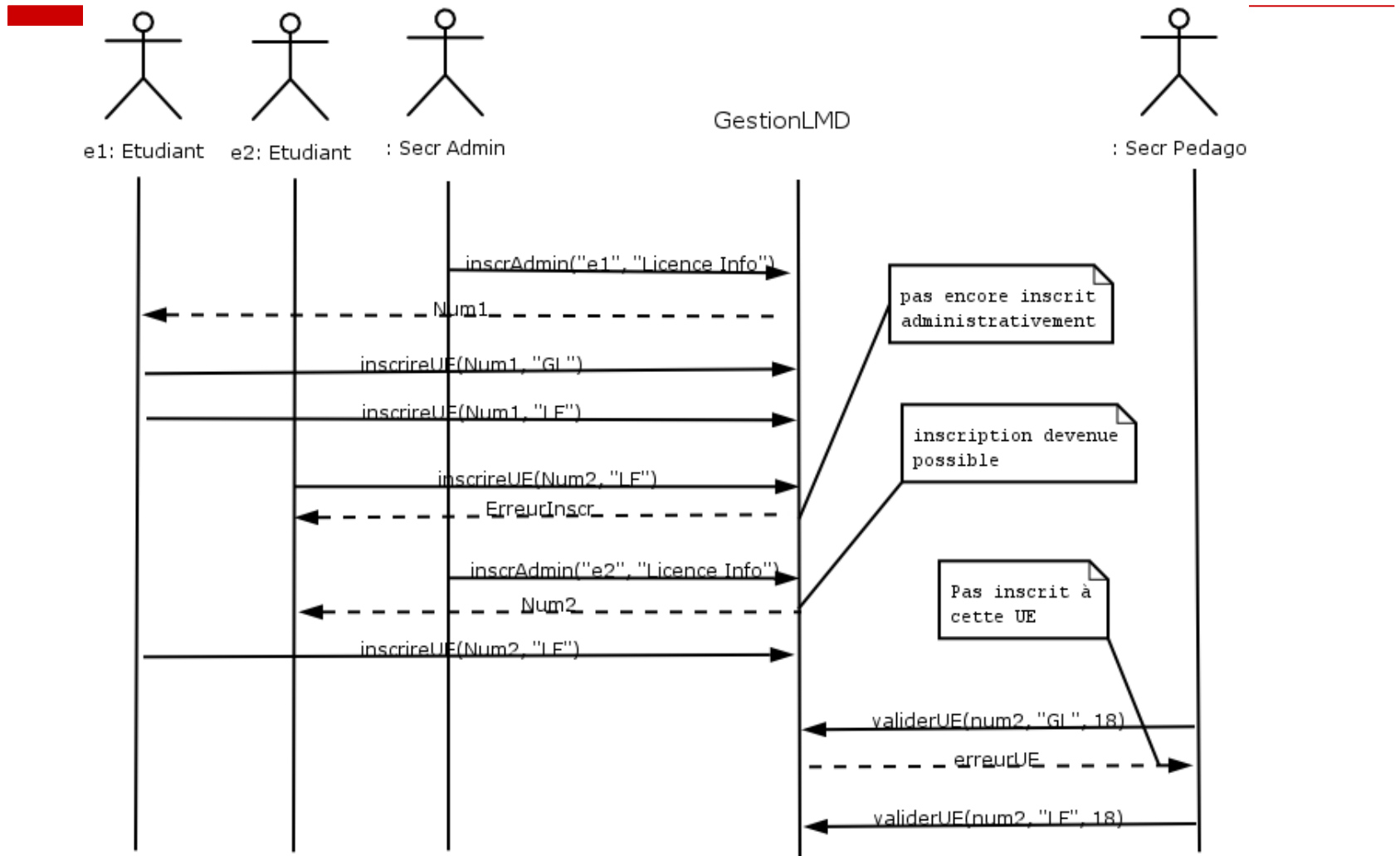


Les principaux diagrammes UML (2)

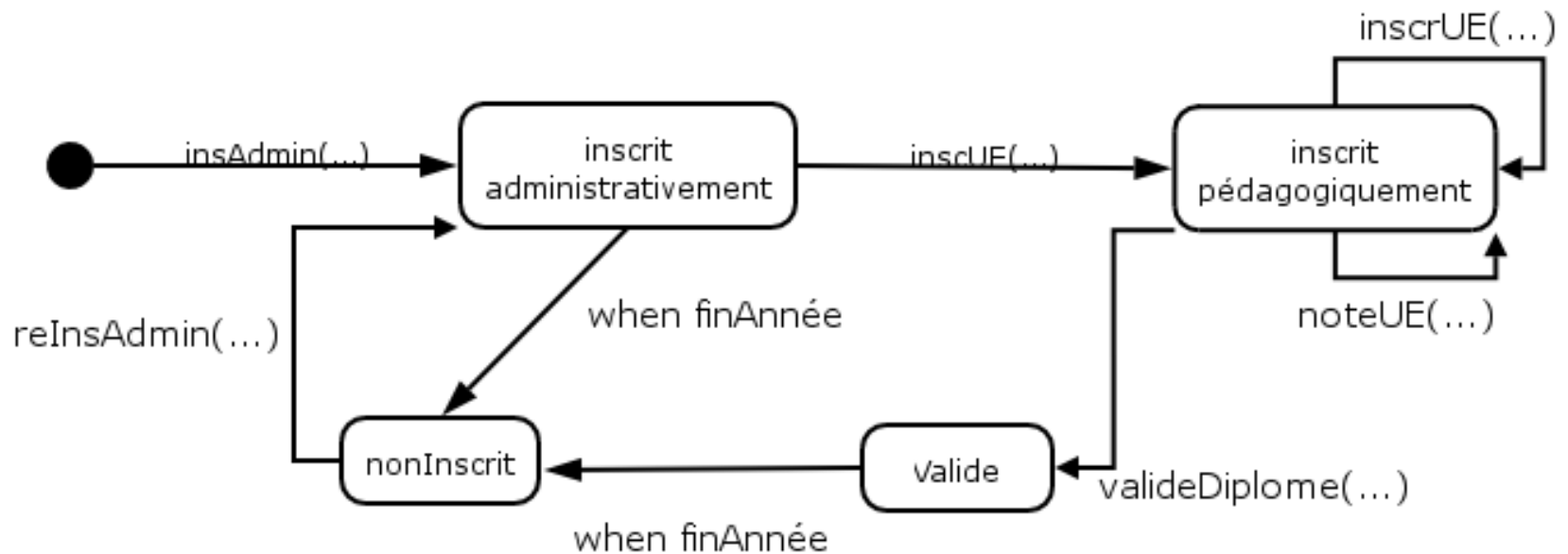
- ❑ *Diagrammes d'interaction*: les **interactions** entre objets pour réaliser une fonctionnalité
 - *Diagramme de séquence*: privilégie le déroulement temporel des échanges. Notions de *scénarios*.
 - *Diagramme de collaboration*: centré sur les objets, privilégie les collaborations entre objets

 - ❑ *Diagramme d'états-transitions* ou « *machines à états* »: le **comportement** d'une instance isolée d'une classe en termes d'états
 - Intéressant si un objet réagit à des « événements » (asynchrones ou non) de l'extérieur
 - Ou si l'objet considéré à un **cycle de vie** intéressant (passage par un ensemble d'états bien caractérisés, avec transitions explicites)
- C'est une extension de la notion d'automates finis (sorties, actions)

Un exemple de diagramme de séquence



Un exemple (simplifié) de machine à états



Règle (imaginaire): Un étudiant doit s'inscrire administrativement avant de s'inscrire pédagogiquement. Les notes des UE sont enregistrées au fur et à mesure. Lorsque toutes ses notes sont connues, on valide ou non son diplôme. En fin d'année universitaire, les étudiants sont désinscrits.

On doit statuer sur le diplôme de tout étudiant inscrit à au moins une UE.

On pourra ajouter des conditions (« gardes ») et des actions aux transitions

Quels diagrammes UML ?

Suivant les applications (ou les étapes), on ne donne que les diagrammes qui sont pertinents...

- ❑ Pour des applications de taille limitée, sans problèmes de synchronisation, ou de temps, on se limite souvent à
 - Diagrammes des cas d'utilisation (« use-cases »)
 - Diagramme de classes
 - Quelques diagrammes de séquence
 - Éventuellement quelques machines à états
- ❑ Pour un système réactif: plus de machines à états
- ❑ Pour un système « temps réel »: des diagrammes de temps en plus
- ❑ Les *diagrammes d'activité* permettent de modéliser des « workflows » compliqués
 - mise en parallèle d'activités et synchronisation, branchements, flot de données ... Ressemble assez à un organigramme !

Ce qu'on ne verra pas...

Conception « Architecturale » d'un système complexe

- Décomposition en grands blocs fonctionnels et leur articulation
- Structuration en composants, connectés via leurs interfaces
- Placement des composants sur des ressources physiques

Composant ?

- Unité logique dont la frontière est définie par des interfaces fournies/requises
- Peut être dupliqué, physiquement réparti, ...
- Peut être remplacé par toute autre implémentation respectant les mêmes interfaces (ex. dans un autre langage...)

Composant ?

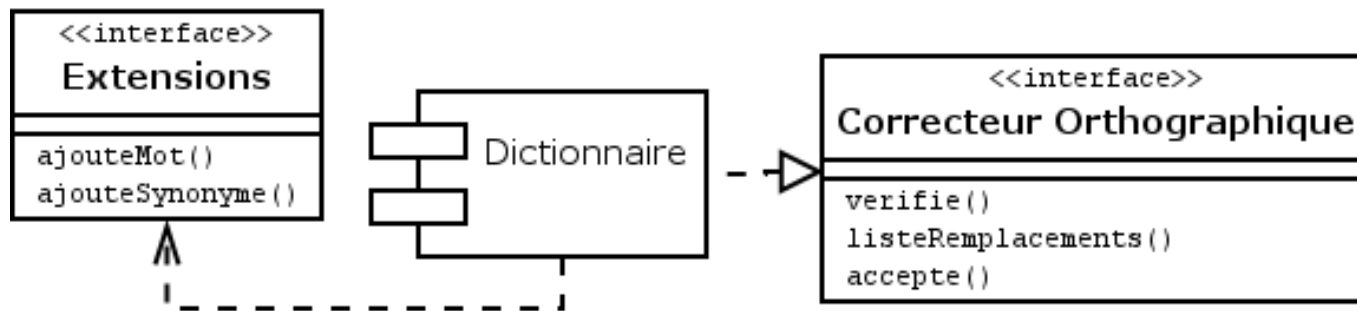
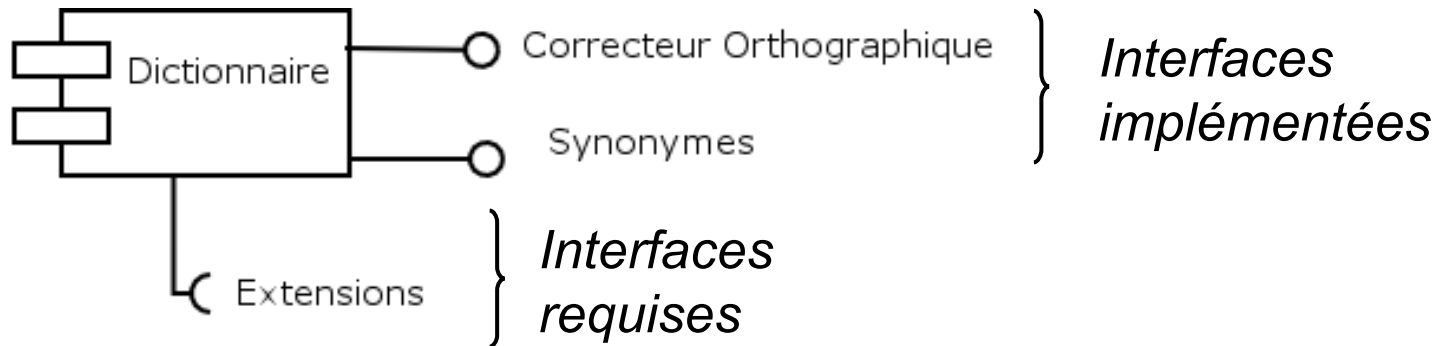
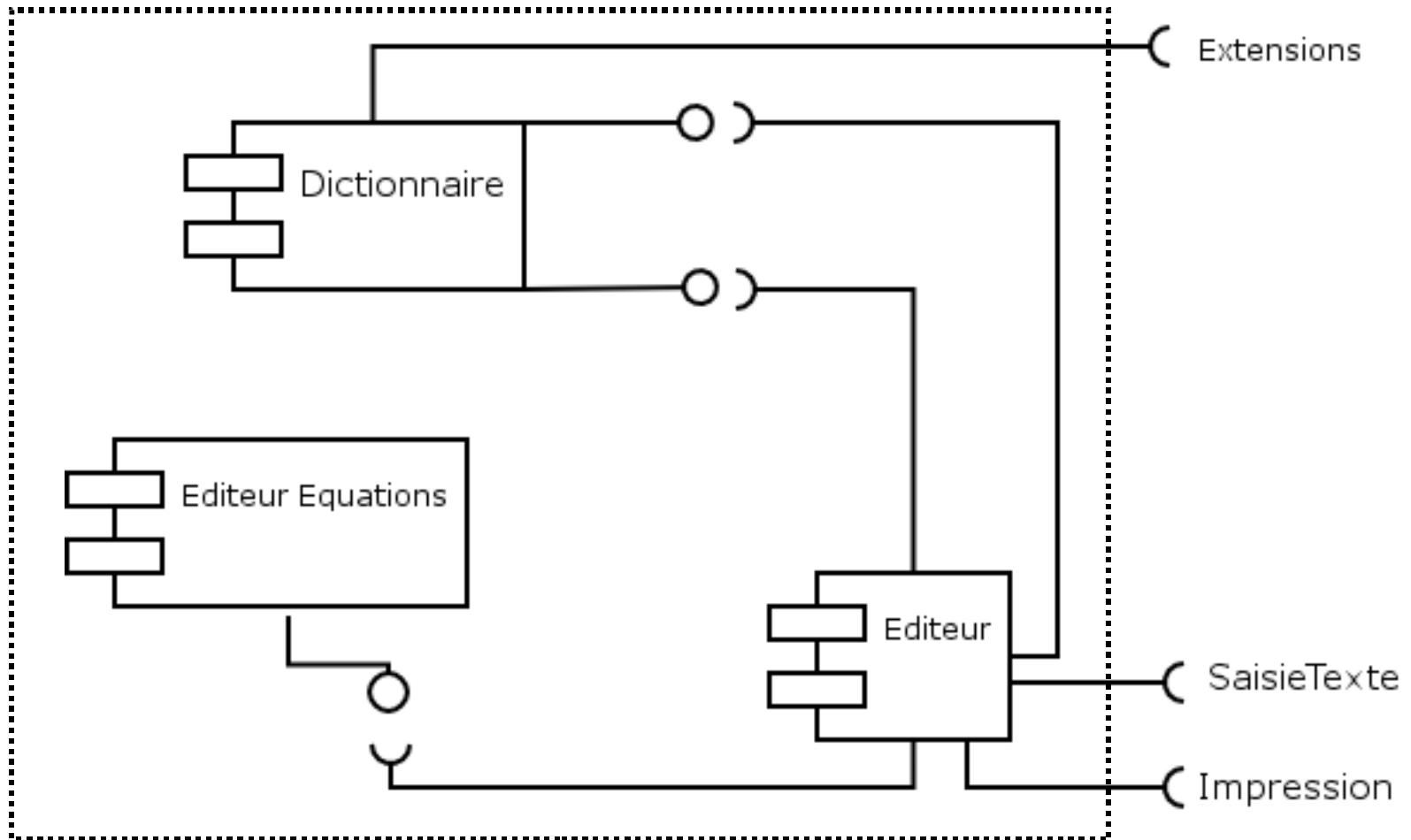


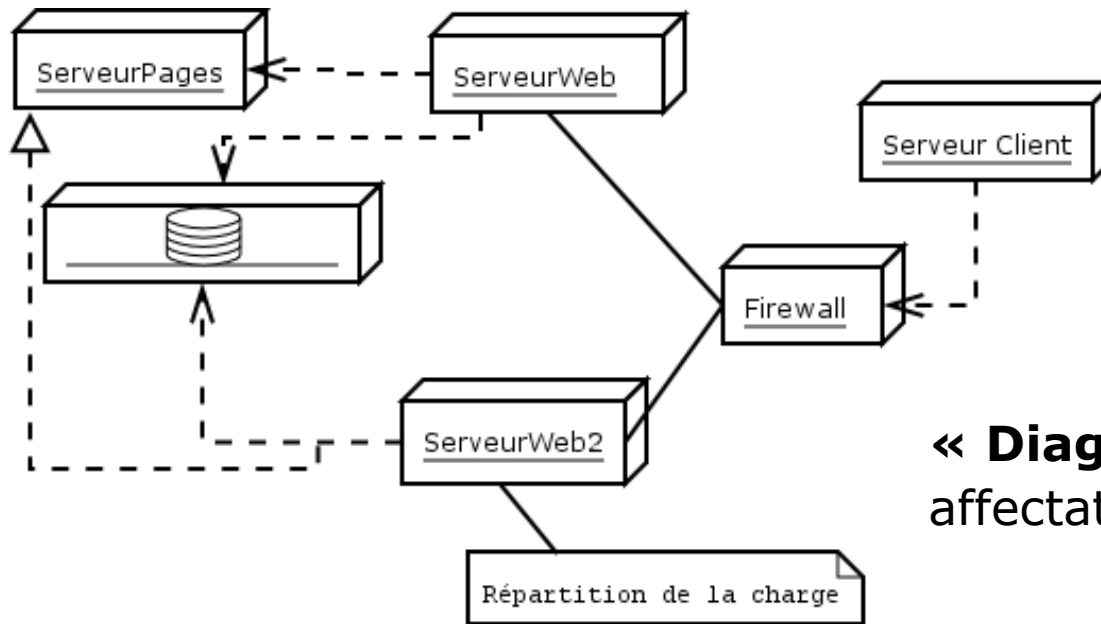
Diagramme de composants



Ce qu'on ne verra pas... (2)

Implémentation « physique »

- « Artefact » : **unité physique** de réalisation d'un système: exécutable, archive jar, DLL Windows, serveur d'authentification
- « Nœud »: ressource physique de traitement, reliés à d'autres nœuds (réseau local, WIFI, WAN ?)



« **Diagramme de déploiement** » : affectation d'artefacts à des nœuds...

Environnements UML

- ❑ Il existe de nombreux environnements UML, dont :
 - Rational Rose, Objectteering, Poseidon, Together
 - « Plug-In » Eclipse (Odomo, adapté à UML 2)
 - plus de nombreux outils limités au dessin de diagrammes
- ❑ Intérêt ?
 - Faciliter la réalisation des modèles
 - Gérer la cohérence entre les diagrammes
 - Production assistée du code, de la doc. (***forward engineering***)
 - Remontée d'informations à partir du code (***reverse engineering***)
- ❑ Inconvénient ?
 - Parfois avec des limitations d'usage ou de notation UML
 - Parfois des notations graphiques non standard
 - Parfois plus orienté conception qu'analyse (biais vers la génération de code)

Partie I.3 : Analyse avec UML

Syntaxe et sémantique des éléments de diagramme

Diagramme des cas d'utilisation

Diagramme de classes: classes et associations

Diagramme de séquence et scénarios

via un exemple

Modèles d'opérations

Contraintes en OCL

Comment faire une analyse



En guise de cahier des charges...

- ❑ Une **station service ATTOL** a plusieurs **postes** de distribution
 - soit *automatiques* (par carte bancaire) ouverts 24h/24
 - soit *manuels* (utilisables seulement si la station est ouverte)Chaque poste est identifié par un numéro.
- ❑ Chaque poste peut délivrer 3 sortes de carburant. A chaque instant, il en délivre au plus une sorte.
- ❑ Chaque poste est muni de 3 compteurs
 - la quantité de carburant servie
 - le prix au litre
 - le prix à payer (qui ne change pas pendant la journée)Les compteurs affichent 0 s'il n'y a pas de distribution en cours.
- ❑ La station dispose de 3 **citernes** (une par type de carburant) avec
 - le niveau courant de carburant,
 - un niveau d'alerte pour prévenir le gérant qu'elle va être vide
 - un niveau minimal qui, une fois atteint, ne permet plus de délivrer du carburant.Les niveaux d'alerte et minimaux sont les mêmes pour toutes les citernes.
- ❑ Le système doit garder une trace de tous les **achats** effectués depuis la dernière ouverture de la station.

En guise de cahier des charges... (2)

- ❑ Un opérateur **ouvre** la station quand il arrive et la **ferme** quand il part
- ❑ Pour **utiliser un poste automatique** :
 - Un client insère sa carte bancaire et tape son code
 - Le système authentifie la carte auprès du service bancaire
 - En cas de succès, le client choisit un type de carburant et se sert d'un certain nombre de litres.
 - Le système enregistre l'achat, envoie un ordre de débit au service bancaire et remet à zéro les compteurs du poste.
- ❑ Pour utiliser un **poste manuel** :
 - le client choisit son carburant et se sert.
 - Les caractéristiques de l'achat sont envoyées à l'opérateur.
 - Lorsque le client a payé en indiquant son numéro de pompe, le pompiste **enregistre** l'achat et remet à zéro les compteurs du poste

Les postes restent bloqués tant que les compteurs n'ont pas été remis à 0.

- ❑ Quand une **livraison de carburant** a eu lieu, l'opérateur met à jour le niveau de la citerne. Une livraison fait toujours repasser au-dessus de l'alarme.

Limitations de (la manière de traiter) l'exemple

- ❑ On ne modélise pas le début et la fin de la distribution, ni la mise à jour des compteurs pendant la distribution
- ❑ On ne décrit pas comment le client choisit le carburant, ni l'authentification de la carte bancaire
- ❑ Il n'y a pas de « simultanéité » d'évènements
- ❑ On suppose que le niveau minimal est suffisant pour assurer toute distribution
- ❑ Le réapprovisionnement livre toujours assez de carburant
- ❑ ...

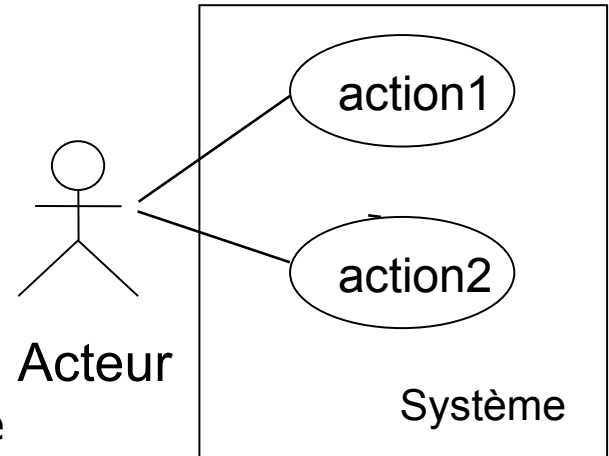
- ☞ On modélise en supposant qu'on connaît au début de l'opération le volume et le type de carburant acheté!
La distribution apparaît ici comme « atomique »

Les « cas d'utilisation »

- les frontières du système
- les fonctionnalités attendues
- les utilisateurs et les partenaires

Les « cas d'utilisation » en UML 2

- ❑ Un « cas d'utilisation » :
 - une **fonctionnalité** du système
 - déclenchée par un **acteur extérieur**
- ❑ **Acteur** : tiers qui joue un rôle
 - En interagissant avec le système :
Il envoie des données / signaux au système et/ou en reçoit des informations
 - Il est extérieur au système

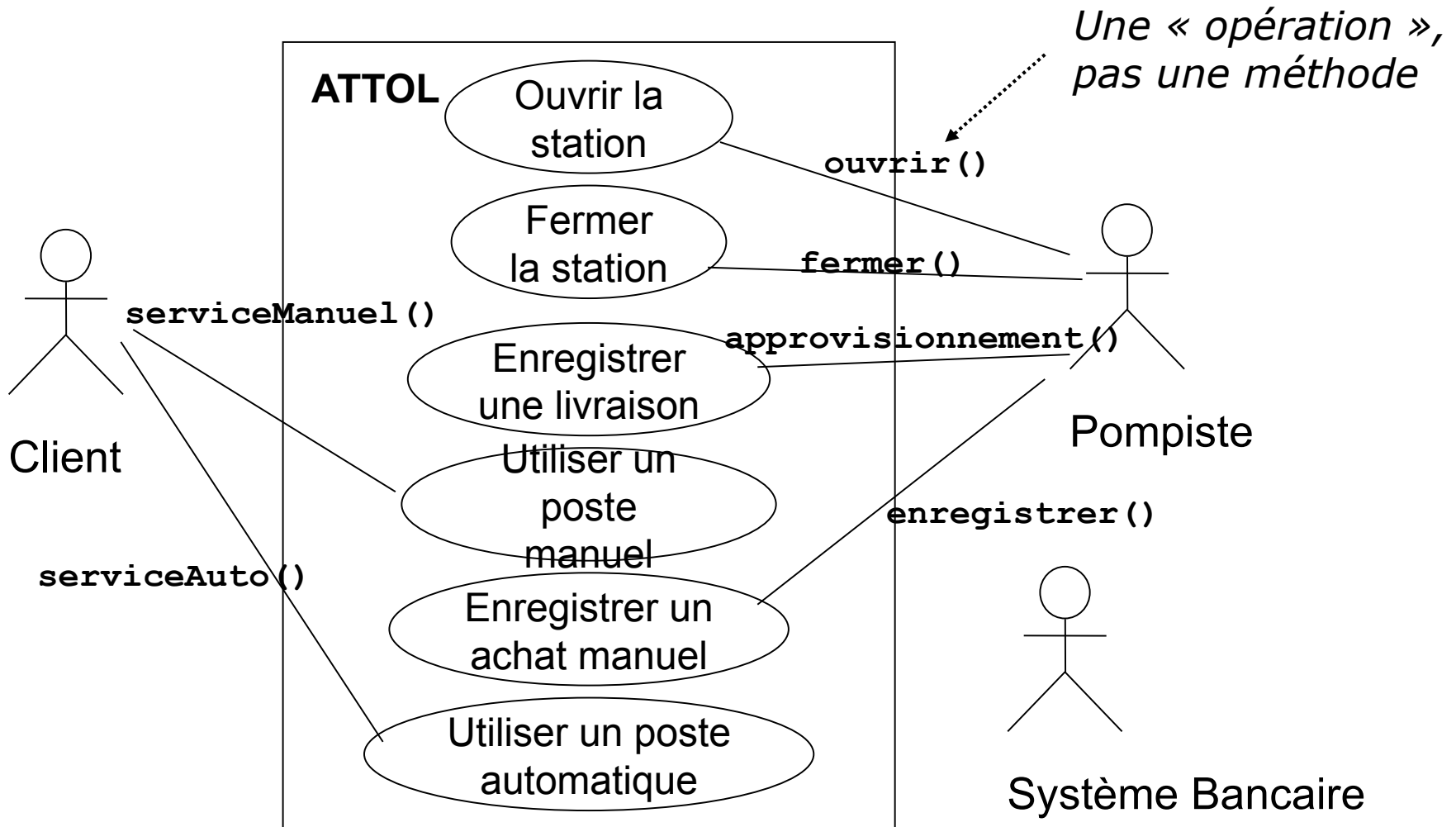


- ☞ *Dans le système, on a parfois une **représentation** de l'acteur (ex. informations sur le client)*
- ☞ *Une même personne peut jouer le rôle de plusieurs acteurs (opérateur, client) dans le temps.*
- ☞ *Plusieurs personnes peuvent avoir le même rôle: vu du système il s'agit du même acteur (les « instances » d'un acteur sont anonymes)*
Un acteur est une « personne logique »

Les cas d'utilisation en UML 2 (2)

- ❑ Les cas d'utilisation décrivent le **comportement** du système, les interactions qu'il a avec l'extérieur.
- ❑ Un système ne fait pas des choses spontanément, mais en réaction à une sollicitation initiale par un « acteur » extérieur
- ❑ On doit préciser un cas d'utilisation
 - en décrivant les flots d'événements à l'aide d'un **texte** ;
 - À l'aide de **diagrammes de séquences** pour préciser graphiquement ces flots.
- ❑ Il faut prendre en compte
 - Les cas normaux, leurs variantes éventuelles
 - Les cas « d'erreurs »
- ❑ Il faudra aussi mettre en évidence les interactions **entre** fonctionnalités des cas d'utilisation (diagrammes de séquence)

Diagramme des cas d'utilisation ATOLL



Diagrammes de séquence en analyse

- ❑ Ils correspondent à des « diagrammes système »
 - le système est vu comme une « boîte noire »
 - Échanges de messages entre les acteurs et le système
 - ☞ jamais entre acteurs (pas du ressort du système !)
 - ☞ ni entre éléments internes au système (conception)
 - Représentation de la chronologie des échanges
 - Pas de boucle, branchement !
 - ☞ Un chemin principal, des alternatives,
 - ☞ des cas d'exception, des commentaires

- ❑ Usages :
 - Illustrer **chaque cas isolément** (diagramme très simple)
 - Illustrer les **interactions entre cas** d'utilisation (scenarii)

- ❑ D'autres types de diagrammes de séquence seront vus en conception pour illustrer le déroulement interne d'une opération

Diagramme de séquence (détails d'interaction)

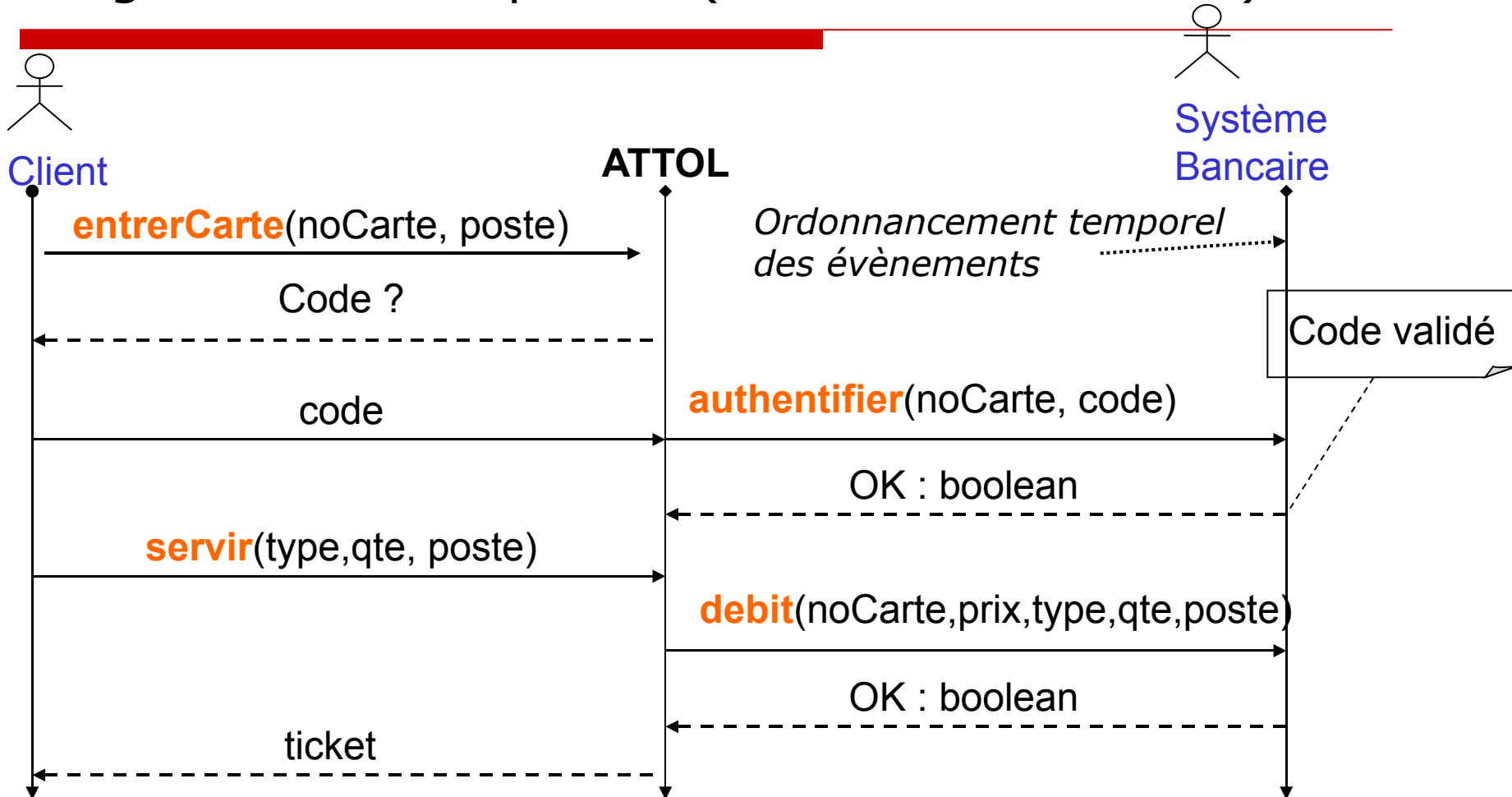
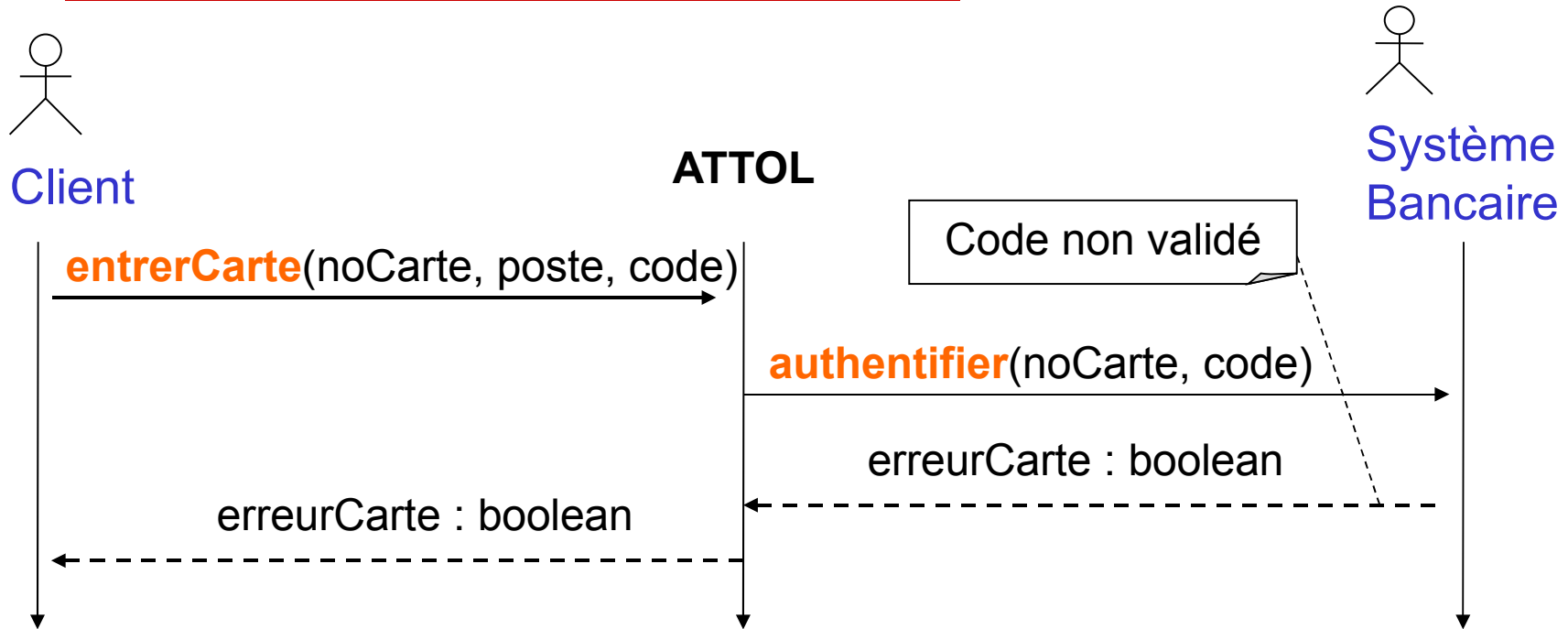


Diagramme de séquence d'un cas d'utilisation



« utiliser un poste automatique » (2)

- 👉 Erreurs ou signaux renvoyés
- 👉 Ici, on a fait le choix de ne pas détailler les interactions

Exemple de synthèse d'un cas d'utilisation

Utiliser un poste automatique	
Cas principal	Un client utilise un poste automatique après avoir authentifié sa carte bancaire et sélectionné un carburant. Le niveau de ce carburant reste au-dessus du niveau d'alerte .
Cas alternatif	Après utilisation d'un poste automatique par un client, le seuil d'alarme du carburant est franchi.
	Après utilisation d'un poste automatique par un client, le seuil minimal du carburant est franchi.
Cas d'exceptions	Echec d'authentification de la carte bancaire d'un client d'un poste automatique
Remarques	L'interface du système garantit qu'en cas de niveau de carburant inférieur au minimum, la sélection du carburant est impossible sur tous les postes. Ce niveau est tel que toute distribution commencée peut être achevée correctement

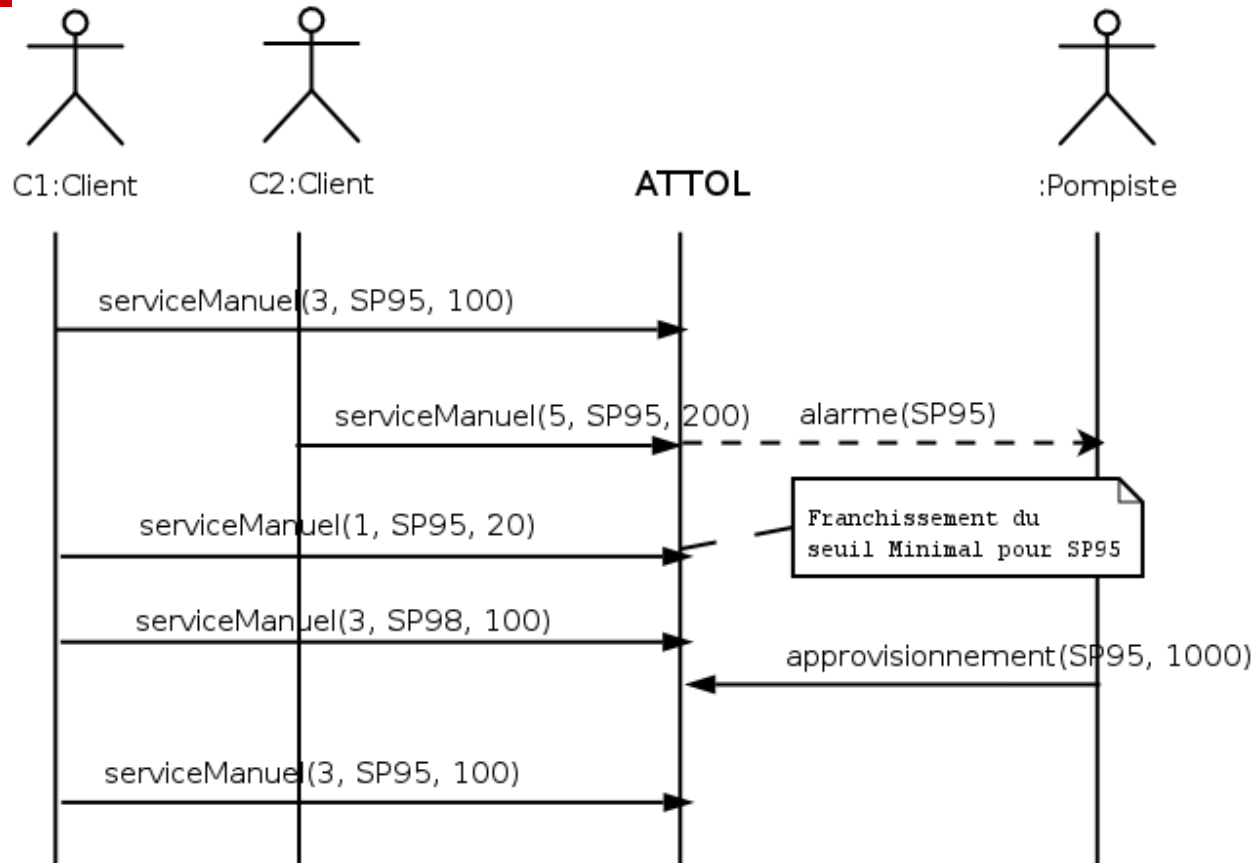
*Description inspirée de « Developing Applications with Java and UML »,
P. R. Reed, Addison-Wesley*

Description textuelle d'un cas d'utilisation

Nom	<i>nom de l'opération</i>
Objectif	<i>en une ou deux phrases claires</i>
Acteur principal	<i>initiateur de l'opération</i>
Acteurs secondaires	<i>autres acteurs impliqués</i>
<i>Pour chaque cas d'utilisation...</i>	
Statut	<i>(principal, alternatif, d'exceptions)</i>
Etapes principales	<i>Enumération des étapes principales et des conditions de déclenchement</i>
<i>pour chaque extension...</i>	
Extension	<i>nom de l'extension</i>
	<i>point de rattachement de l'extension</i>
	<i>Etapes principales de l'extension</i>

*Description inspirée de « Developing Applications with Java and UML »,
P. R. Reed, Addison-Wesley*

Un diagramme de séquence **inter** cas d'utilisation



Ce scénario n'est pertinent que s'il permet d'illustrer un fonctionnement global du système. Quels sont les « évènements » mis en valeur ?

Retour sur le diagramme précédent

- ❑ Ce qu'on arrive à exprimer sur cet exemple :
 - Les carburants ne sont pas associés à un poste précis
 - Si le niveau est compris entre le niveau d'alarme et le niveau minimal, on peut continuer à se servir de ce carburant

...

Penser aux scénarii pertinents, qui illustrent l'usage attendu du système, l'interaction entre opérations

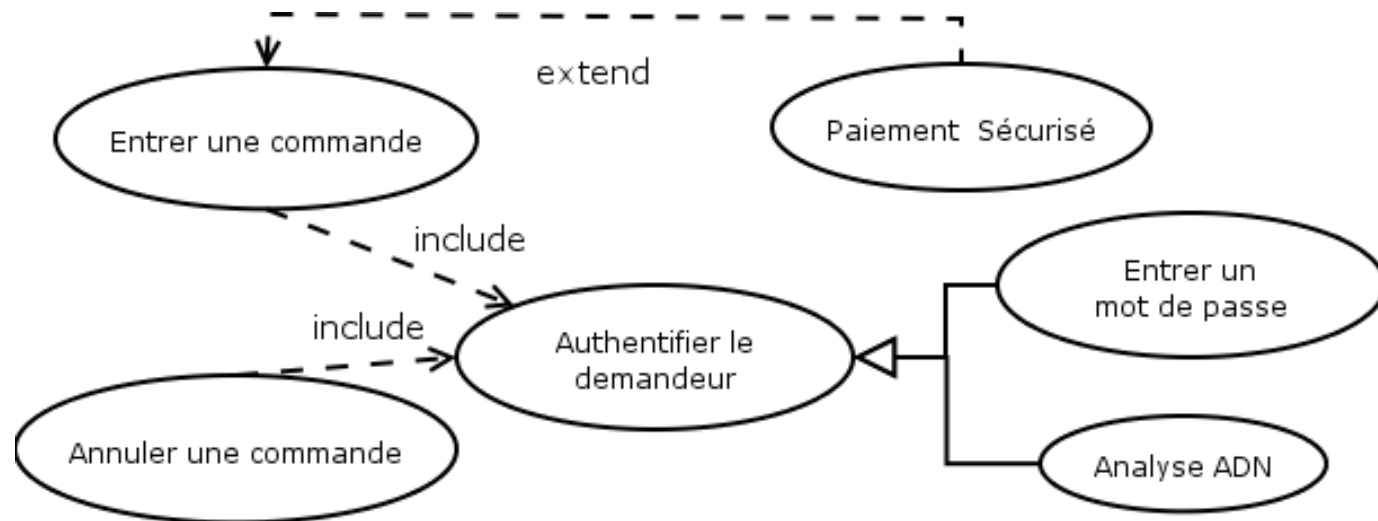
- ❑ Ce qu'on n'arrive **pas** à exprimer :
Quand le niveau minimal est atteint, on ne peut plus se servir d'un carburant : l'opération est inhibée (non exprimable !) et il n'y a pas « d'observateur » du niveau d'une citerne.

On ne peut pas illustrer ce cas par un scénario (et on ne pourra pas le tester !)

[👉 Identification d'un manque « d'observateurs » dans ce système !](#)

Relations entre cas d'utilisation

*A utiliser
avec modération !*



extend:

- Le cas de base a prévu une liste de points d'extensions, ainsi que des conditions d'activation

include:

- Le cas d'utilisation n'existe pas isolément
- Le cas de base mentionne explicitement les points d'inclusion

généralisation: on évitera, sauf si cela apporte vraiment quelque chose

Quelques règles sur les cas d'utilisation

- ❑ Entre 3 et 6 fonctionnalités par diagrammes de cas d'utilisation. Si plus, introduire un cas d'utilisation plus abstrait qu'on raffindra ensuite dans un nouveau diagramme
- ❑ Il est plus utile d'avoir une bonne description textuelle qu'un diagramme sophistiqué avec de nombreuses relations
- ❑ Illustrez votre diagramme par des scénarii pertinents illustrant les aspects dynamiques du système, les acteurs impliqués, les cas d'erreur.

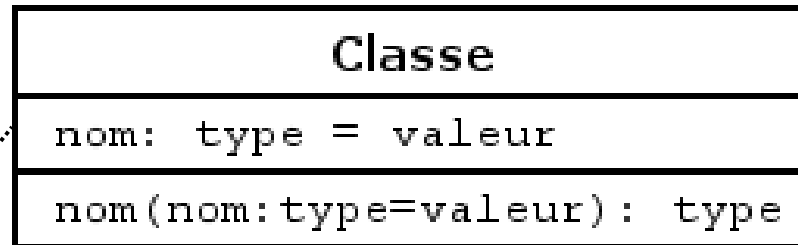
Le diagramme de classes :

- les entités concernées
- leurs relations

Syntaxe des Classes et Instances en UML 2

Marqueurs:

+: publique
 - : privé
 #: protected
 : paquetage
 / : dérivé



Modificateurs:

statique
abstrait

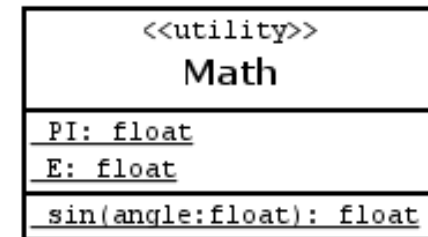
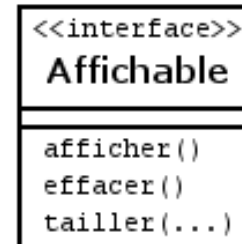
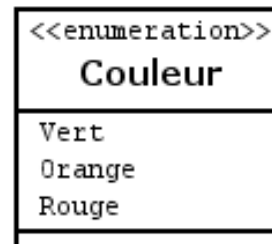
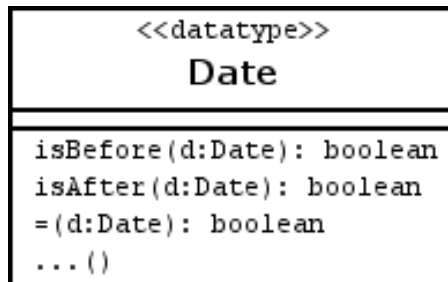
Mode d'un paramètre:

in (par défaut)
 out
 in out

Instances:



Stéréotypes:



Les classes en phase d'analyse

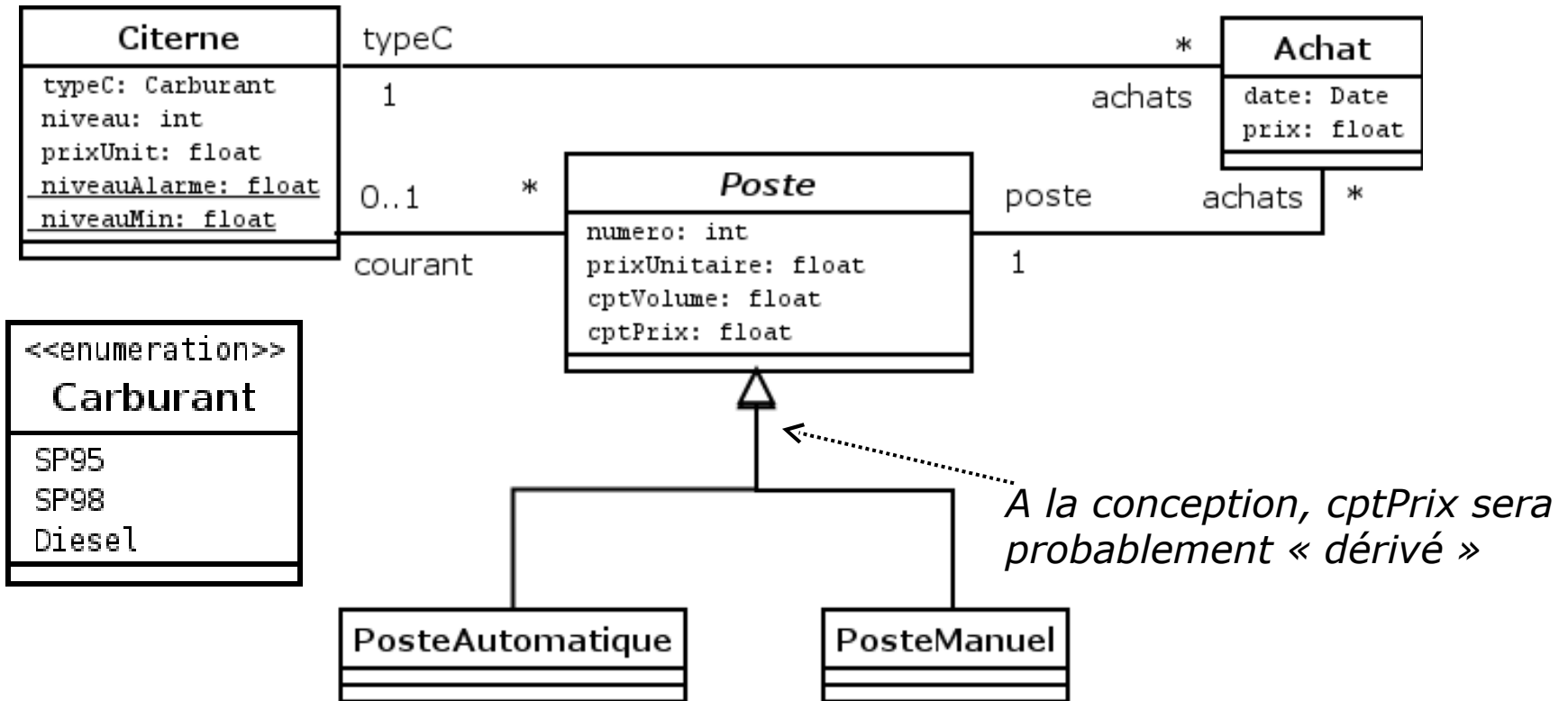
□ Les attributs

- De type simple (int, boolean, ...) ou primitif (Date) uniquement !
- Les liens avec les autres classes d'intérêt sont représentés par les **associations**
- Dans le modèle d'analyse, on les considère comme **public** (on raffinerà à la conception, ainsi que les méthodes d'accès)
- On ne distingue pas les attributs « dérivés » des autres

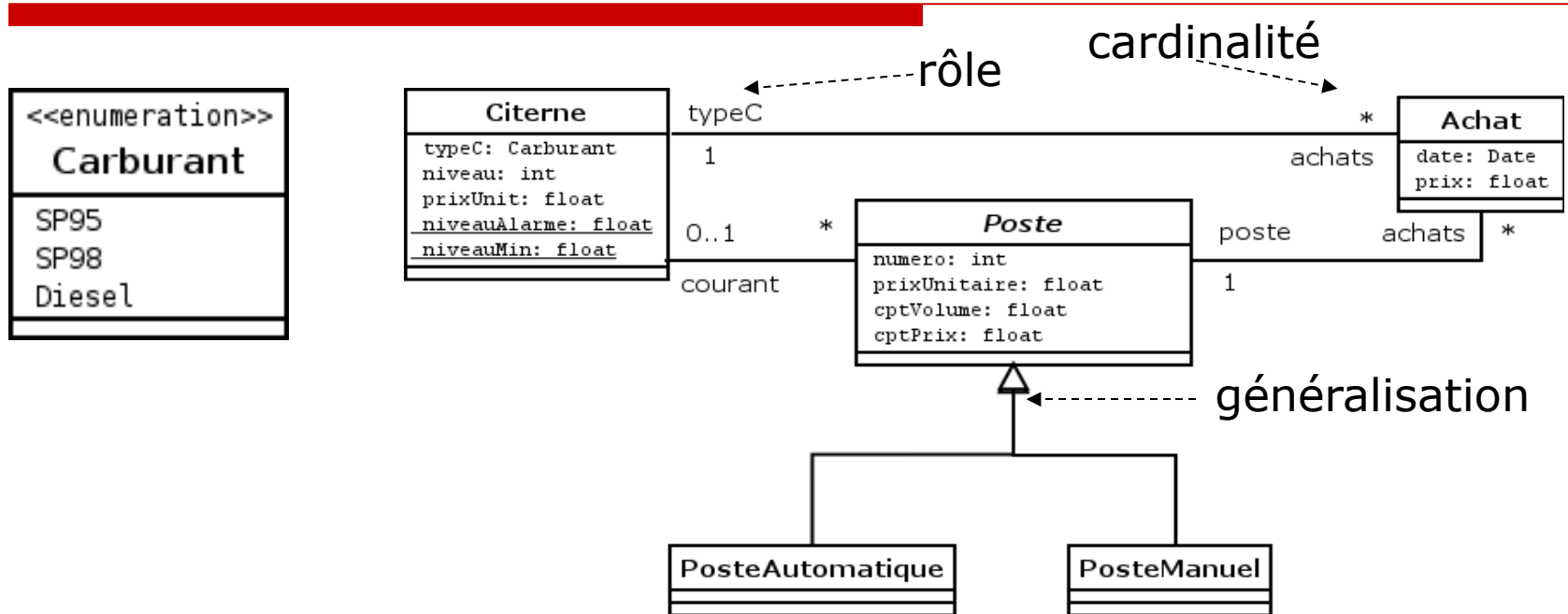
□ Les opérations

- dans un premier temps, on ne considère que les opérations principales (liées aux cas d'utilisation)
- on ne les rattache pas à une classe particulière, donc on ne se préoccupe pas des méthodes à ce niveau (quels paramètres ?)

Une première version



Les associations en UML 2



Pour a: Achat, a.poste correspond à **une** instance de Poste.

Pour c: Citerne, c.achats correspond à une **collection** d'instances de Achat

Pour p: Poste, p.courant correspond à une **collection de 0 ou 1** Citerne.

Les rôles permettent de naviguer à travers les associations

Le nom de classe peut servir de rôle par défaut (si pas d'ambiguïté)

Cardinalité/Multiplicité

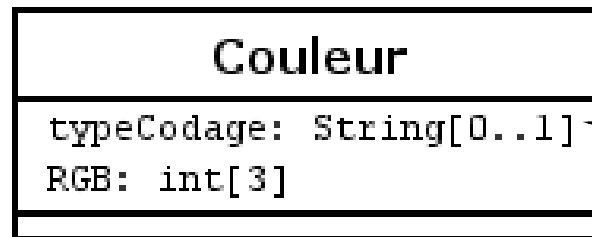
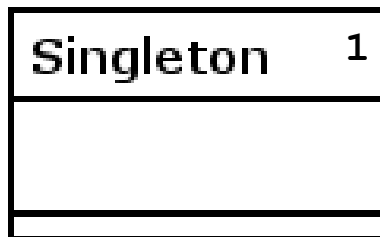
Cardinalités dans les associations: de la forme

- 1, 2, ou un nombre entier précis (**pas** une expression !)
- * : un nombre quelconque, éventuellement nul
- un intervalle comme 1..*, 0..1, 1..3, (**pas** 1..N)
- ☞ *on donnera systématiquement les cardinalités*
- ☞ *Attention à la différence: une instance (1), au plus une instance (0..1), une collection d'instances (* ou 1..*),*

Sémantique d'une association : *des tuples*

On peut associer une multiplicité aux attributs et classes

☞ *À utiliser avec précaution !*

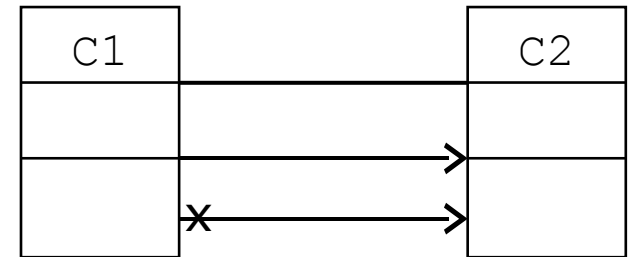


*0 ou 1 chaîne,
pas une chaîne
de taille 0 ou 1*

Traversée d'associations

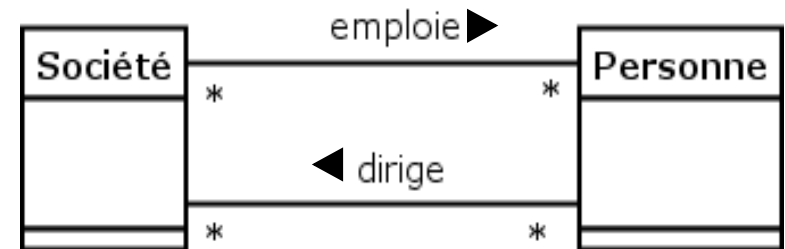
Navigabilité: 3 possibilités

- ➔ Dans la phase d'analyse, on se limite à des associations implicitement navigables dans les deux sens
- ➔ A la conception, on pourra choisir de ne conserver qu'un sens
- ➔ En pratique, la navigabilité « réelle » se verra en fonction des rôles nécessaires pour exprimer les invariants



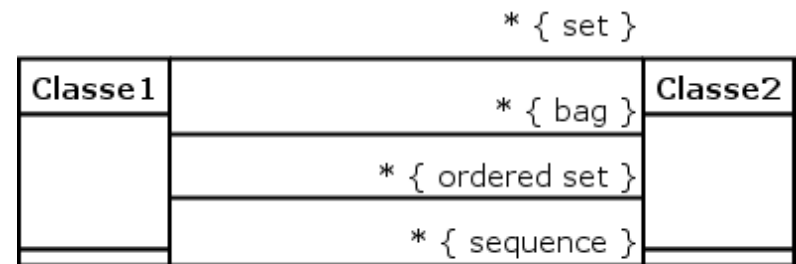
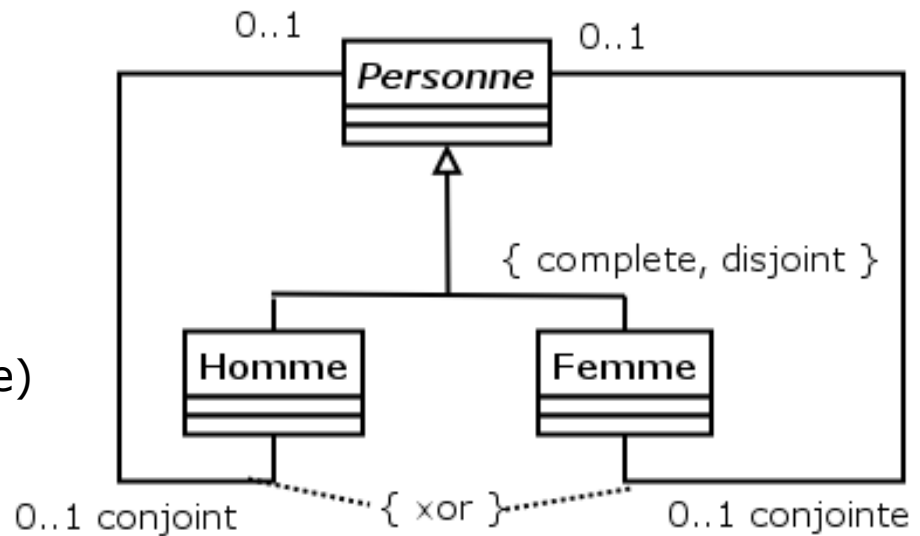
Nommage : on peut nommer une association

- ➔ *on privilégie plutôt les noms de rôles...*
- ➔ *on donnera un rôle si on en a besoin dans une contrainte*

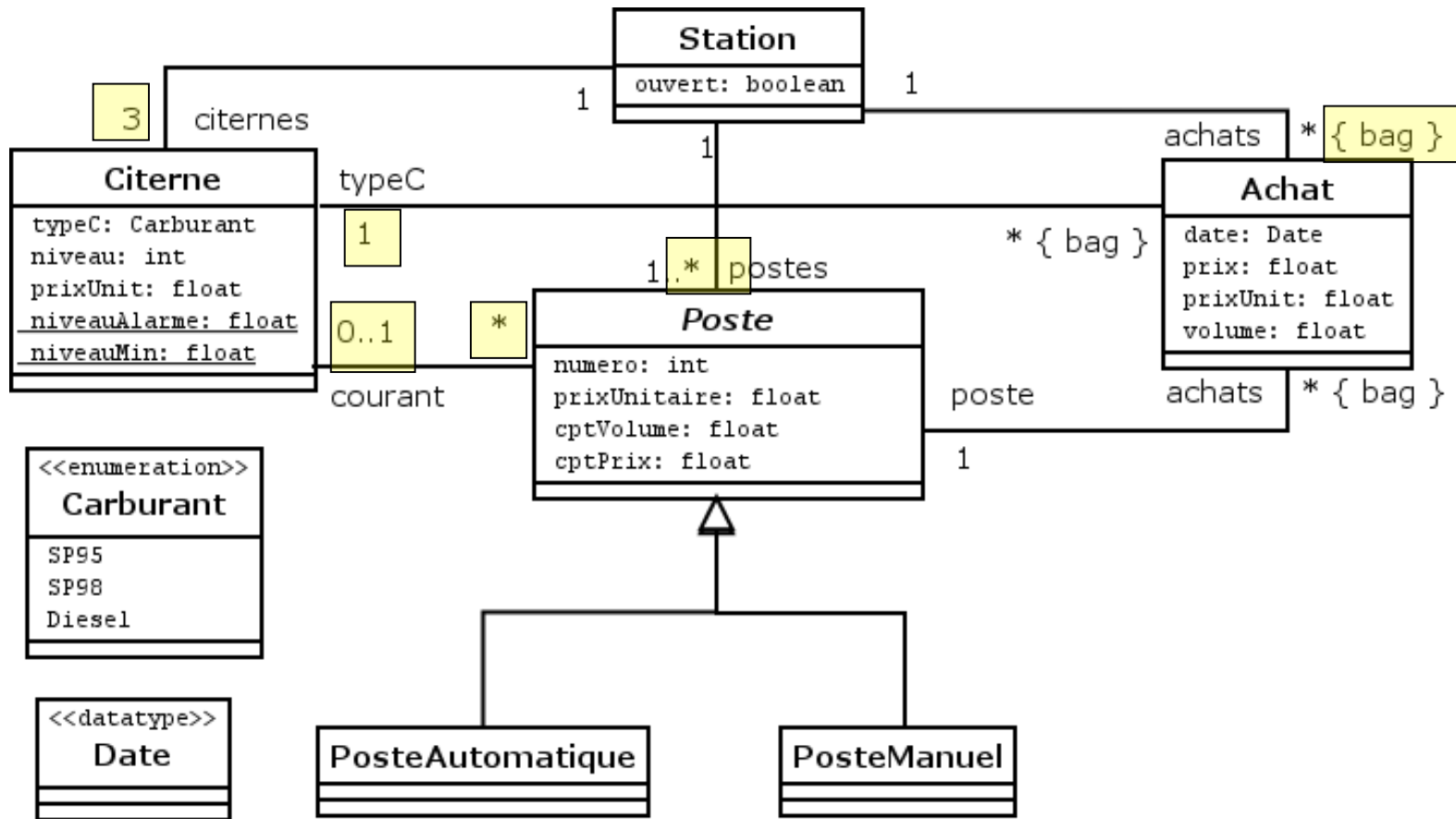


Contraintes posées sur une association

- ❑ Pour la généralisation:
 - complete, incomplete
 - disjoint, overlapping
- ❑ Entre associations
 - xor (souvent ambigu, on lui préférera une contrainte explicite)
- ❑ Immuabilité d'une extrémité de lien (readonly)
- ❑ Contraintes associées à la multiplicité *:
 - Ordonnée ou non?
 - Avec duplication des éléments ?



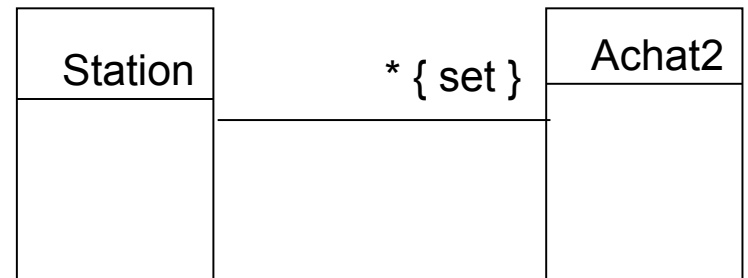
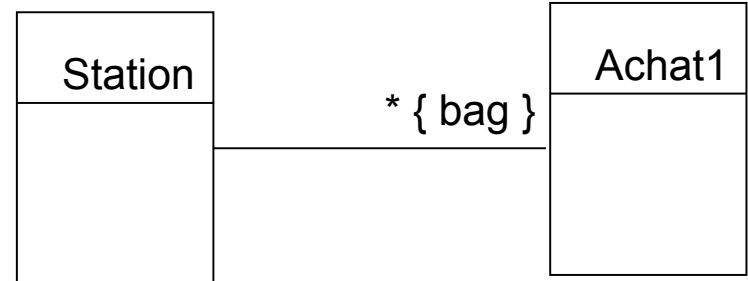
Les associations pour ATTOL



Bag ou Set: vaut mieux y regarder à deux fois...

```
public class Achat1 {
public boolean equals(Object o) {
    if (o == null) return false;
    try { Achat1 a = (Achat1) o;
        return a.date.equals(date)
            && a.prix == prix && ...
    } catch (ClassCastException e)
        { return false; }
}

public class Achat2 {
    public boolean equals(Object o) {
        return this == o;
    }
}
```



Vous préférez un bag{Achat1} ou un set{Achat2} ?

Ça dépend aussi de la précision associée à la notion de Date !

ATTOL: au-delà du diagramme de classes

Le diagramme n'exprime pas tout !

☞ *Nécessité d'invariants informels ...*

- les citernes ont des carburants distincts
- en cours de distribution le prix affiché est le prix du carburant choisi
- s'il n'y a pas de carburant choisi, les compteurs sont à 0
- $\text{niveauMin} \leq \text{nivAlarme}$
- $\text{cptPrix} = \text{prixUnitaire} * \text{cptVolume}$

...

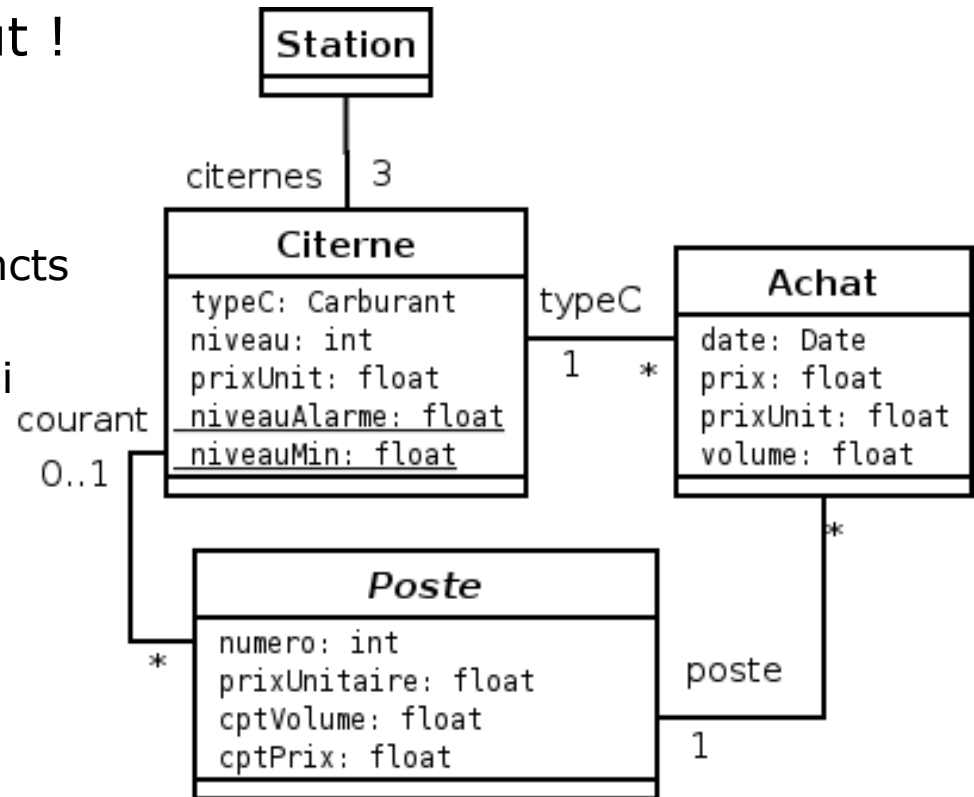
☞ *ou formels (OCL) !*

context p:Poste

inv paiement: p.cptPrix = p.cptVolume * p.prixUnitaire

inv RAZ: p.courant->empty() implies

p.cptVolume = 0 and p.prixUnitaire = 0



ATTOL: redondance d'associations

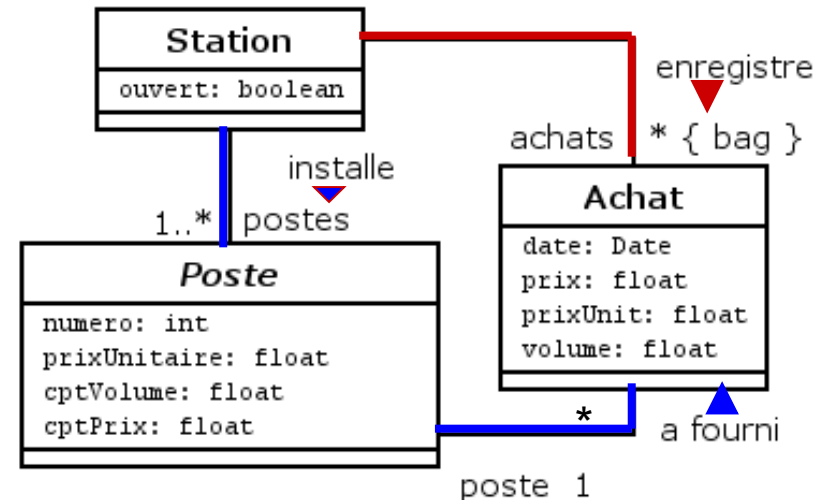
Il existe plusieurs moyens d'accéder aux achats !

- Par navigation avec **installe** et « **a fourni** »
- via **enregistre**

Sont-elles cohérentes ?

Invariant ?

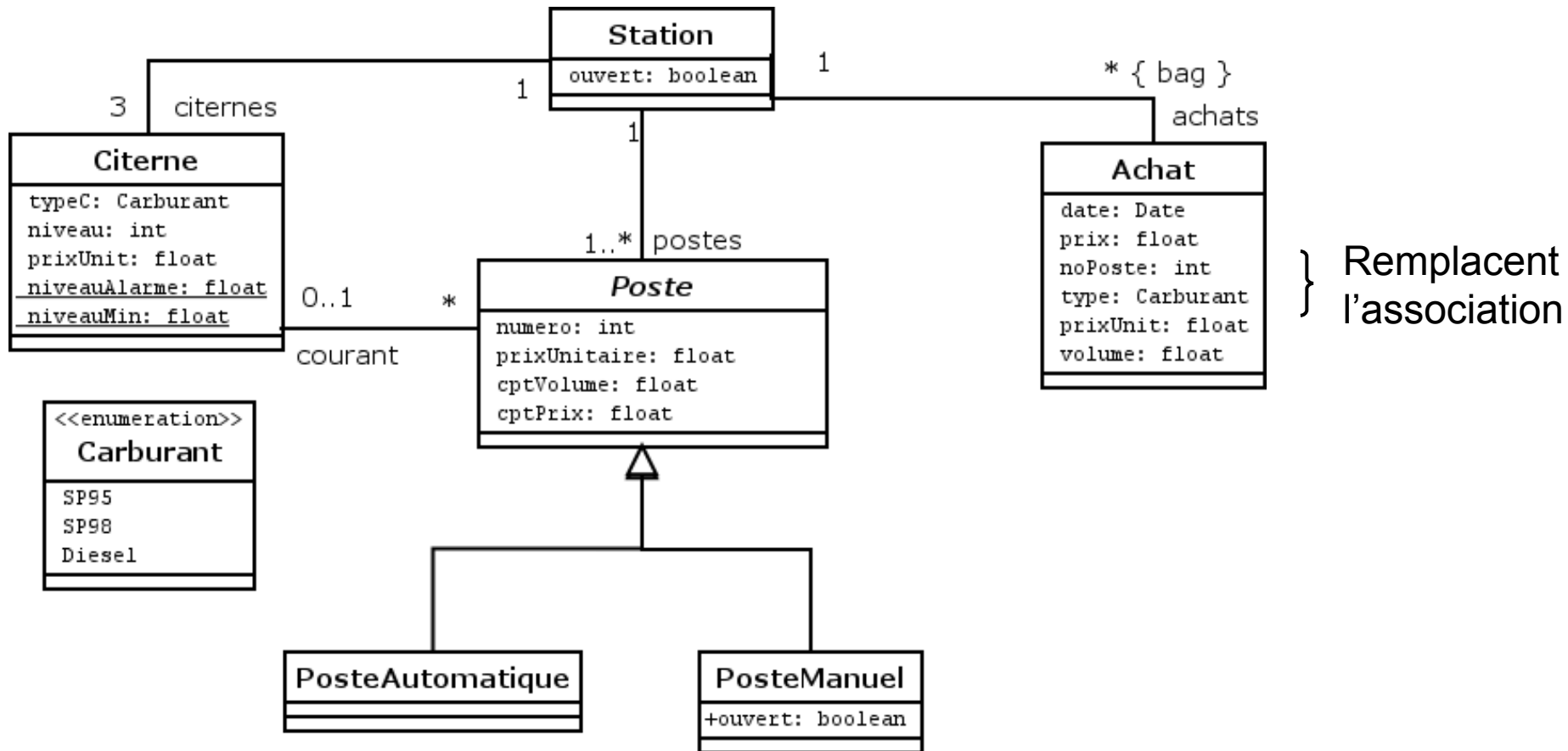
« la collection d'achats **enregistre** est la réunion, sur l'ensemble des postes **installe** de la station, des collections d'achats **a fourni** de chaque poste »



A l'analyse : il faut se demander si on garde la redondance

A la conception : Idem ! Ou comment navigue-t-on ?

ATTOL sans association redondante



👉 *On a maintenant besoin d'un invariant sur `noPoste` !*

Variations sémantiques d'associations

❑ Association simple

❑ Agrégation : « est composé de »

☞ antisymétrie et transitivité des liens

☞ Associations binaires uniquement !

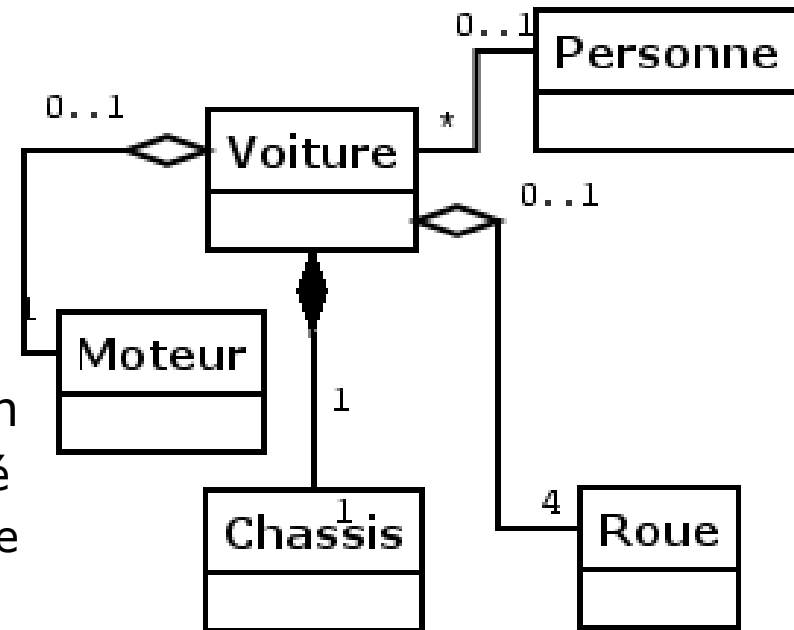
❑ Composition : restriction d'agrégation

➤ Un composant ne peut pas être partagé

➤ La durée de vie du composant est gérée par l'instance propriétaire

☞ des variations « sémantiques » !

☞ Java : même représentation pour les 3...
(C++: Sous-objet ? Pointeur ? Référence ?)



Variations sémantiques d'associations (2)

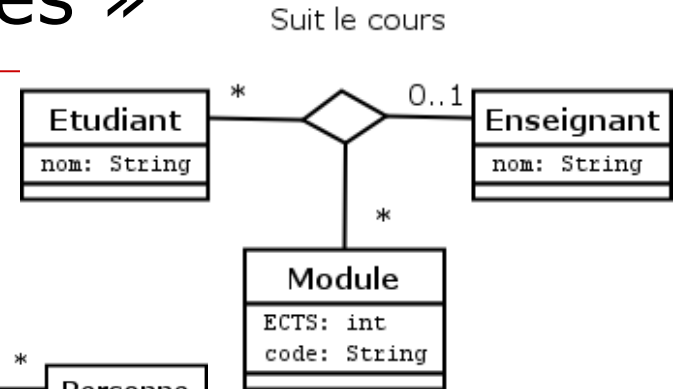
Généralisation/Spécialisation

- Pas le droit de re-déclarer un attribut
- Les méthodes redéfinies doivent coïncider sémantiquement avec la superclasse sur les aspects communs (pas de redéfinition arbitraire !)
- Pas limité à de l'héritage simple,
- ni à un typage statique (classification dynamique)
- ☞ Traduction dans un langage de programmation ?
- la sous-classe hérite des contraintes de la superclasse (invariants, pré/post-conditions des méthodes !)

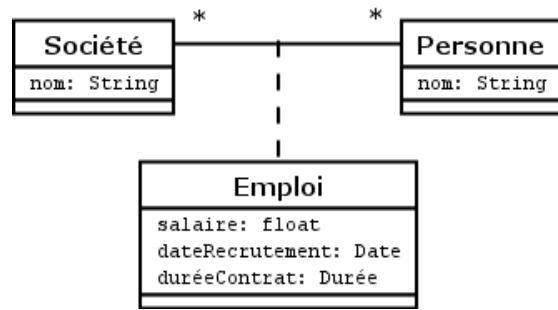
Les associations « exotiques »

Association N-aire

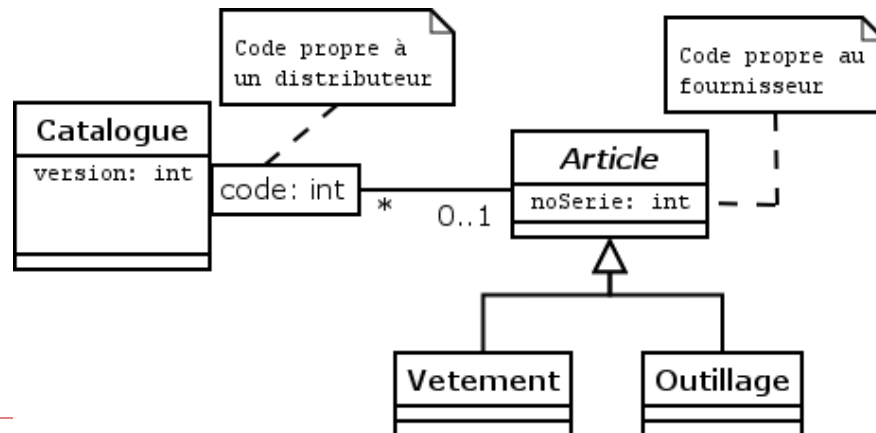
Multiplicité ? Sur une branche = pour une instance fixée à chacune des N-1 autres branches



Association avec attributs (« classe association »)

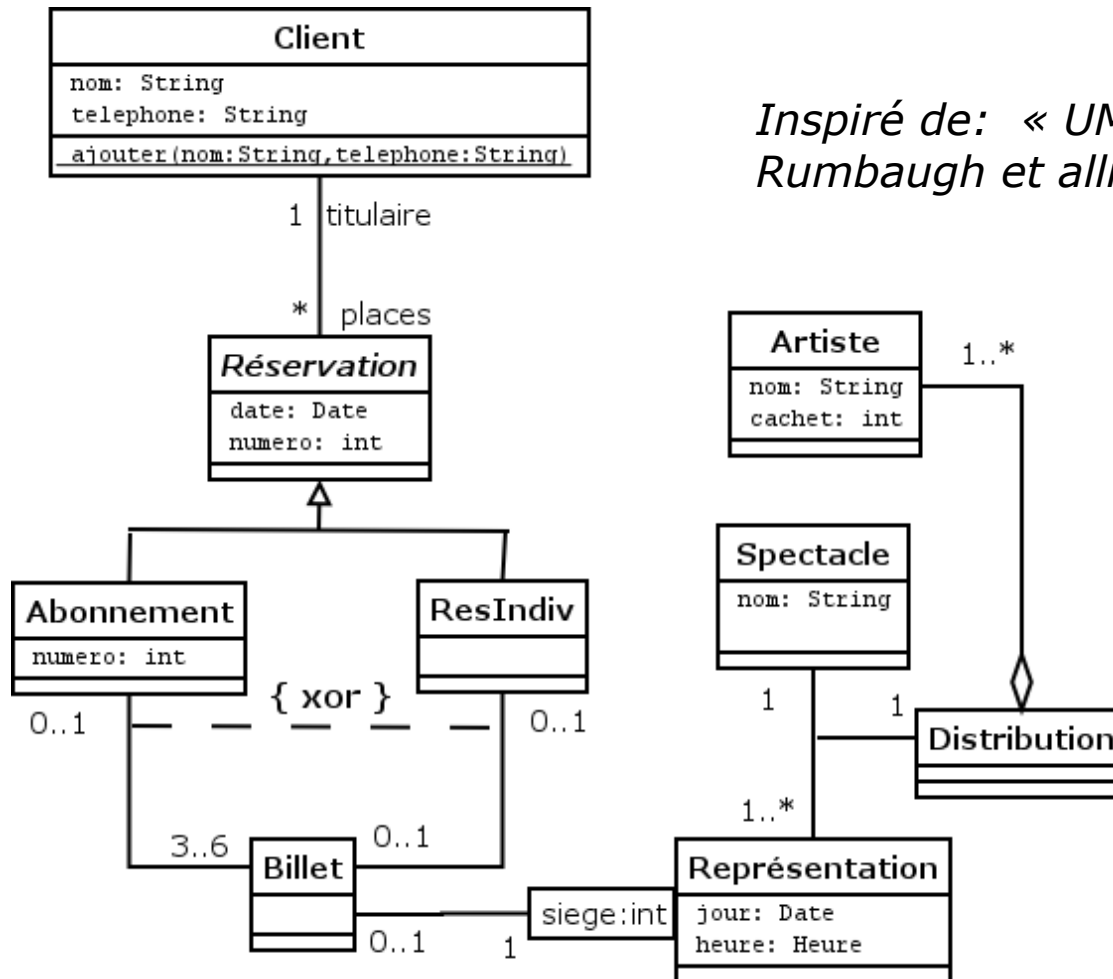


Association « qualifiée »

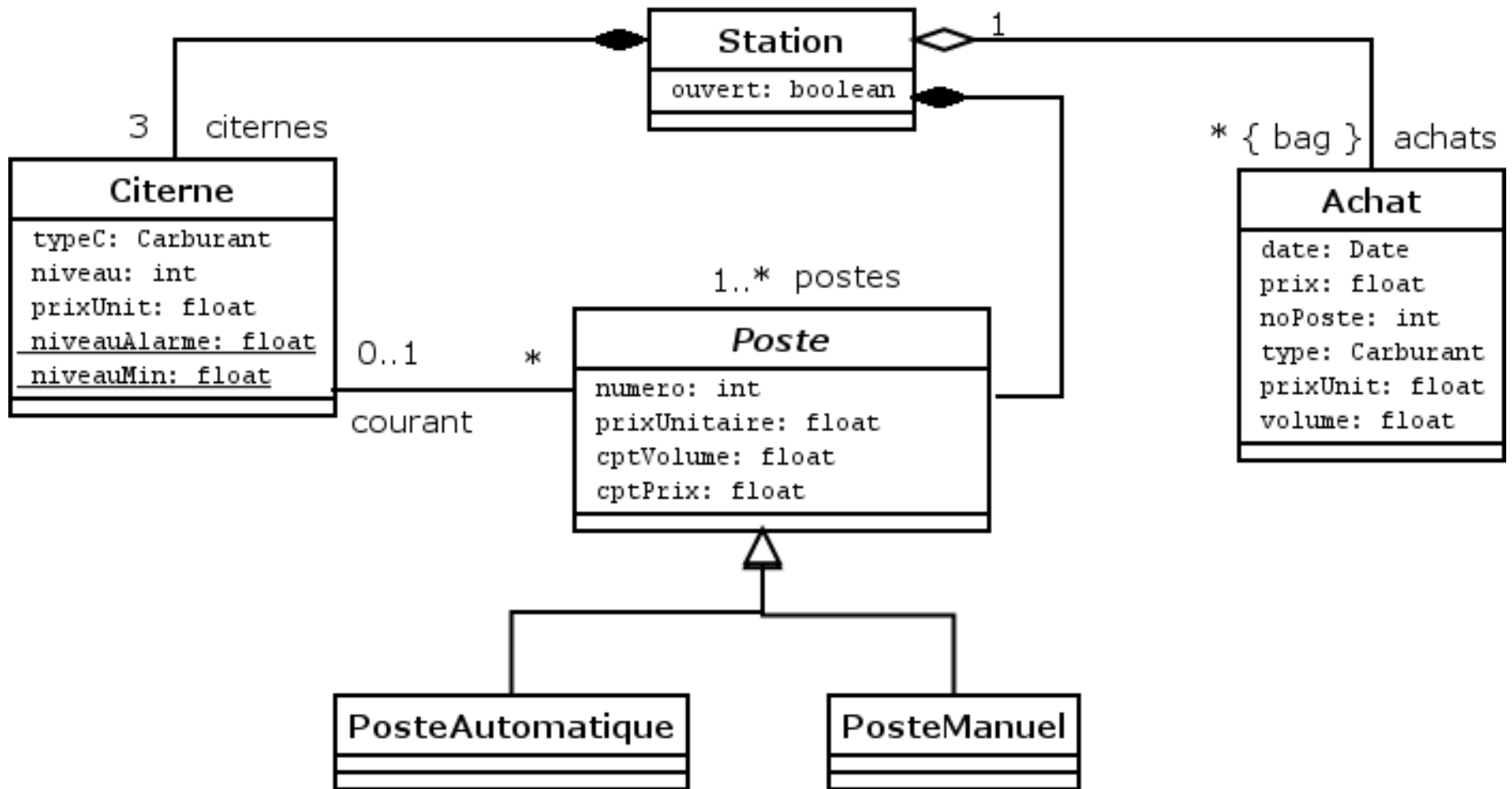


En mettant les variations d'association ensemble...

*Inspiré de: « UML 2.0 Guide de référence »,
Rumbaugh et alli., CampusPress, 2005*



Le diagramme de classe ATTOL final ?



Exercice(s)...

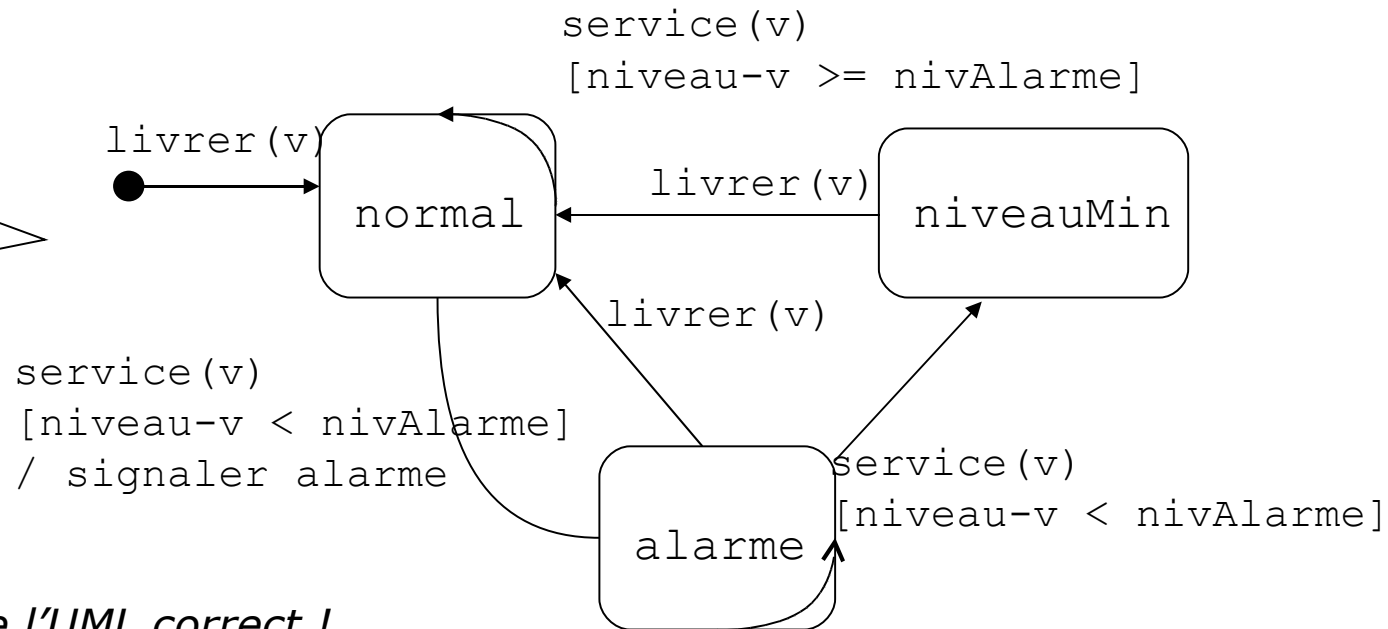
- ❑ La station dispose de plus de 3 types de carburant, mais un poste n'offre toujours que 3 carburants.
- ❑ Historiser les achats: on distingue les achats consultables en ligne des achats archivés. A la fermeture de la station les achats du jour sont archivés.
- ❑ La station a un programme de fidélisation des clients (via une carte) qui leur donne des réductions et un ensemble de stations partenaires qui acceptent la même carte.
- ❑ Les tarifs peuvent varier (même dans la journée). On veut pouvoir retrouver le tarif en vigueur pour un achat.

☞ *Adaptez le diagramme de classes en conséquence...*

Une machine à états pour ATTOL

On veut insister sur la gestion des niveaux de carburant...

Il manque une transition !



Ce n'est **pas** de l'UML correct !

☞ Ultérieurement, on formalisera les transitions (méthodes, évènements, réalisation d'une condition)

☞ Donne une vue différente d'une instance (prototypique) de la classe, qu'on devrait retrouver implicitement ou explicitement dans l'implémentation...

Modèles d'opérations

Invariants et modèles d'opérations en O.C.L.

Un mini catalogue O.C.L.

Ref: J. Warmer, A. Kleppe : « *The Object Constraint Language
(Second edition)* », Addison-Wesley, 2003

Opérations et Méthodes en UML

- ❑ **Opération** : un « service » qui peut être demandé au destinataire.
- ❑ Une **méthode** implémente une opération et est attachée à une classe.
 - Entre classe/sous-classe, une méthode peut être redéfinie pour implémenter la même opération
 - On peut envisager des méthodes surchargées (différant par les paramètres) pour la « même » opération.
- ❑ Une opération peut
 - Provoquer un changement d'état du système (d'un objet ou d'une instance d'association...)
 - Retourner une valeur à l'appelant
 - Retourner un « signal » à un acteur

Les opérations en analyse

- ❑ En phase d'analyse, les opérations ne sont pas forcément rattachées à une classe mais peuvent « flotter » :
 - On ne connaît pas encore forcément tous les détails d'interaction (quels paramètres ?)
 - En conception, on ajoutera des « classes d'interface » entre les acteurs et le cœur du système. Leurs méthodes récupéreront les paramètres et appelleront les méthodes des classes d'intérêt
 - ☞ L'opération pour le cas d'utilisation sera à destination d'une telle classe d'interface.

- ❑ Pendant la phase de conception:
 - On choisira les classes d'interface
 - On associera définitivement les opérations aux classes
 - On ajoutera les méthodes pour implémenter les opérations

Les modèles d'opération

Description des opérations associées aux cas d'utilisation

- ❑ Aspects syntaxiques: signature complète, y compris les signaux envoyés
 - Si une opération ne change pas l'état du système, on le précise

```
context Poste::calculePrixTotal(...): Float { isQuery = true }
```

- ❑ Aspects sémantiques:

- **Pré-condition:** ce qu'on suppose établi à l'entrée
- **Post-condition:** ce qu'on garantit établir en sortie

☞ Les pré- et post-conditions peuvent comporter plusieurs clauses

- Chaque clause est une expression logique qui décrit un effet (ou une condition) de l'opération

- L'ordre des clauses n'est pas significatif: ce n'est pas un algorithme !

☞ On décrit « **ce que doit faire** » la fonction, pas « **comment** » elle le fait

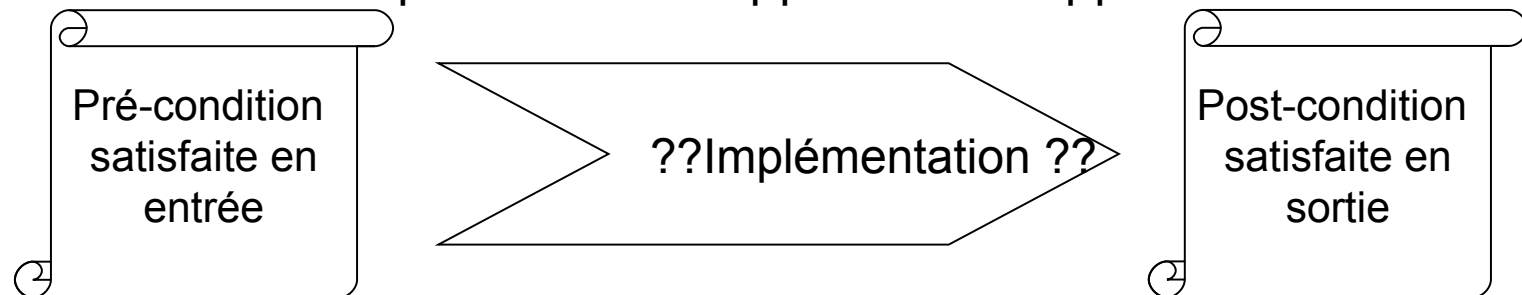
☞ Utile d'essayer de rédiger à la forme passive et/ou au passé (point de vue: quand l'opération est terminée !)

Pre/PostConditions

Basé sur la notion de « *programmation par contrat* »

- Distinction entre ce qui relève de l'opération *appelante* ... et ce qui relève de l'opération *appelée*:

☞ « Contrat » passé entre l'appelant et l'appelée



- L'implémentation de l'appelée doit garantir la terminaison et la post-condition pour l'état de sortie, si la précondition est vérifiée pour l'état d'entrée
- Si la pré-condition n'est pas vérifiée: comportement indéfini.
- Assurer la précondition relève de la responsabilité de l'appelant

Un modèle d'opération en français

context `Station::fermeture()` : boolean **signals** Echec

pre:

- la station est ouverte (sinon le bouton de fermeture est inhibé par l'interface du système)

post: -- *en fin d'opération*

- ✓ S'il existe un poste manuel en attente de paiement, Echec a été envoyé au pompiste et la station est toujours ouverte
- ✓ Si tous les postes manuels sont encaissés alors
 - le champ `ouvert` de la station a été mis à `false`
 - On a retourné `true`

Un modèle d'opération en OCL

context Station::fermeture() : boolean **signals** Echec

pre:

ouvert = true -- pour l'instance courante de Station

post: -- *en fin d'opération*

let ok: Boolean = not (postes->exists
 (p | p.oclIsTypeOf(PosteManuel) and
 p.courant->notEmpty()))

in

- ✓ (not ok) *implies* Echec.sent() and ouvert
- ✓ ok *implies* (not ouvert and result = true)

Pré/Post Condition (2)

- ❑ Du point de vue de l'appelée, la précondition doit être vérifiée en entrée et on ne se préoccupe pas dans la post-condition de ce qui se passe dans le cas contraire.
- ❑ Il n'en demeure pas moins que la pré-condition doit pouvoir être établie par tous les appelants
 - ☞ Précondition « réaliste » (pas de « vœux pieux »)
- ❑ Pré-condition ou comportement spécifié par l'appelant

context fermeture() *si final Echec*
Pre : true
Post: not ouvert@pre
 implies Echec.sent()
 ouvert@pre **implies** ...

un vrai choix à l'analyse !

context fermeture()
Pre : ouvert = true
Post:
 ouvert = false and
 postes->forall(p| not p.ouvert)

Garanti comment ?

@pre: la valeur de l'attribut ouvert de l'instance courante à l'entrée

Pre/Post-conditions (3)

- « assertions » : se retrouvent sous une forme restrictive et différente dans certains langages de programmation

Exemples : `assert` C++ et Java (JML)

« différente » : Les assertions sont vérifiées à l'exécution

« forme restrictive » = une expression du langage de programmation

- On peut vouloir exprimer des choses non exprimables dans un langage de programmation !

On vérifierait comment ?

context Commande

inv: `Commande.allInstances->isUnique(c|c.numero)`

Pre/Post-Conditions (4)

Un dernier exemple...

```
context minSeq (in s:Sequence(Integer)):int {isQuery = true}
Pre: s->notEmpty()
Post: s->includes(result)
        s->forall(e| result <= e)
```

Le résultat retourné

- 👉 Laisse la place à toute implémentation de la recherche !
- 👉 Pourtant, le résultat de `minSet` est bien spécifié...
- 👉 Pas de sur-spécification qui contraindrait l'implémentation....

$i^{\text{ème}}$ élément de `s`:
`s->at[i]`

Exercice:

Spécifier sous forme de pré/post-condition le tri d'une séquence...
Attention aux spécifications incomplètes qui autorisent des implémentations « triviales » !

OCL: langage de spécification

- ❑ **OCL (*Object Constraint Language*)**: pour exprimer formellement des contraintes sur
 - Les classes et associations (invariants)
 - Les opérations (pré/post-conditions)
 - Les transitions des machines à états (gardes)
- ❑ OCL est un **langage de spécification**, pas de programmation: pas d'aspects algorithmiques, de séquençement, etc.
- ❑ Les notations peuvent faire référence aux éléments des diagrammes du modèle:
 - Classes, Attributs, Types
 - Rôles et cardinalités des associations
 - États des machines à états
 - Signaux pour les opérations et machines à états
- ☞ Les classes du modèle correspondent à des types OCL

Les invariants en OCL

Contraintes associées aux classes:

```
context Classe          -- instance anonyme (self)  
inv nom: expression OCL avec self
```

ou (mieux)

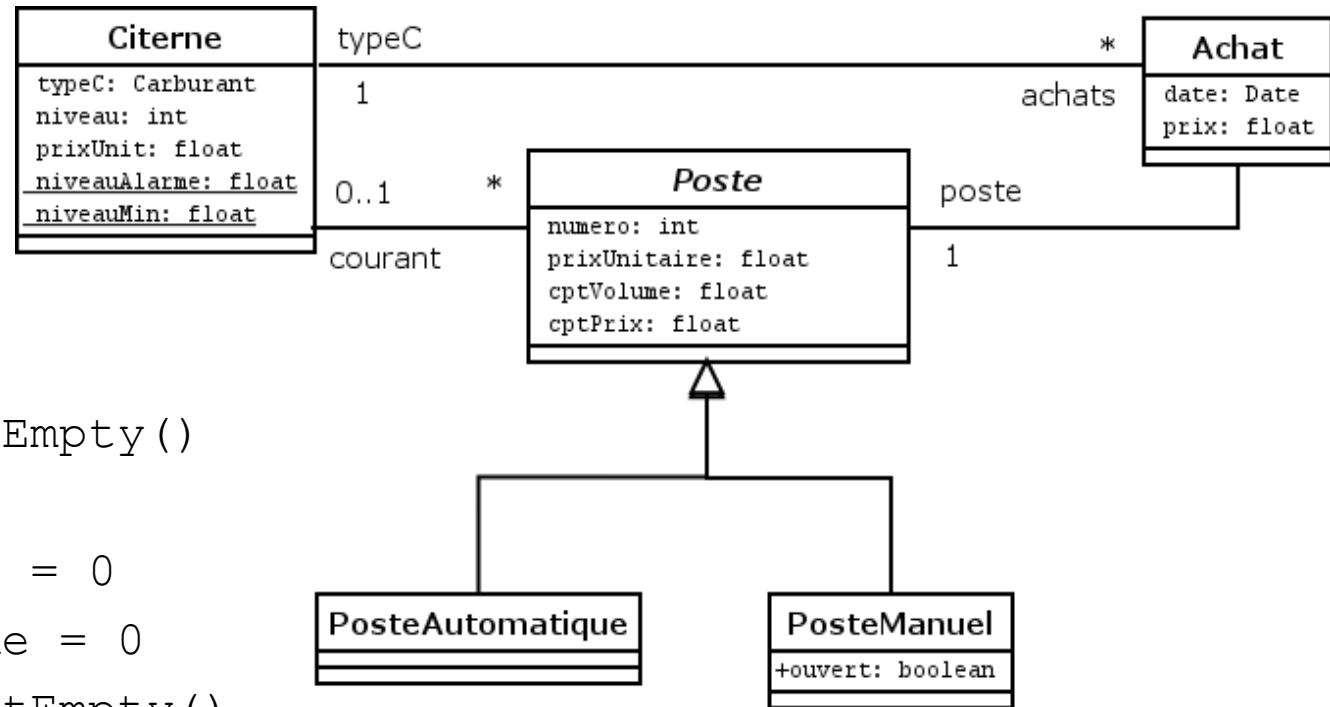
```
context c:Classe       -- instance prototypique c  
inv nom: expression OCL avec c
```

Un tel invariant est implicitement quantifié universellement
(« pour tout *c* appartenant à *Classe*)

Les expressions OCL peuvent faire référence:

- aux attributs de la classe
- aux rôles des associations auxquelles elle participe...
- aux classes atteignables via ces associations

Exemple d'invariants de classes



context p: Poste

inv: p.courant->isEmpty()

implies

p.prixUnitaire = 0

and p.cptVolume = 0

inv: p.courant->notEmpty()

implies p.prixUnitaire = p.courant.prixUnit

inv: p.cptPrix = p.cptVolume * p.prixUnitaire



p.courant.prixUnit est indéfini si p.courant->isEmpty()

Invariants de classe

- ❑ Chaque objet de la classe doit satisfaire l'invariant:
 - À sa création
 - Après chaque application d'une opération publique

- ❑ Toutes les instances doivent respecter l'invariant !

Contre-exemple:

```
context a: Achat
```

```
inv a.prix = a.poste.prix
```

n'est **pas** un invariant: c'est vrai seulement au moment de cet achat !

- ❑ De nombreux invariants sont déjà exprimés dans le diagramme
 - ☞ Une cardinalité est un invariant
 - ☞ Pareil pour les types/sous-types des attributs/associations
 - ☞ Pareil pour les contraintes associées aux associations

Propriété « collective » d'une classe

- ❑ On peut avoir besoin d'une **contrainte globale** à l'ensemble des instances (?) d'une classe

context Commande

inv: `Commande.allInstances->isUnique(c | c.numero)`

context Singleton

inv: `Singleton.allInstances->size() = 1`

context Client

inv: `Client.allInstances(c1, c2 | c1 <> c2 implies
c1.nom <> c2.nom or c1.adresse <> c2.adresse)`

Seul cas où la notation `allInstances` est autorisée !
Elle à valeurs dans la collection des instances de la classe

allInstances ou pas ?

- ❑ L'invariant pour une classe est déjà implicitement quantifié universellement
 - ☞ Pas besoin de `allInstances` pour parler d'une propriété d'une instance prise isolément !

```
context p:Poste 😊  
inv: 0 <= p.prix
```

```
context Poste ☹️  
inv: Poste.allInstances->(p | 0 <= p.prix)
```

```
context s: Station 😊  
-- la clause ci-dessous peut suffire à nos besoins !  
inv: s.postes->forall(p | 0 <= p.prix)
```

Modèles d'opérations en OCL

- ❑ La formalisation des modèles d'opérations !

```
context NomClasse:Operation(parametres): TypeRetour  
    signals SignauxPotentiels {isQuery = false/true }  
pre:    -- énumération de clauses  
post:   -- énumération de clauses
```

Notations:

- `result` : la valeur retournée
- `oclIsNew()` : caractérise les instances créées pendant l'exécution de l'opération
- `@pre` : caractérise une valeur (d'attribut, d'association, ...) **en entrée** de l'opération

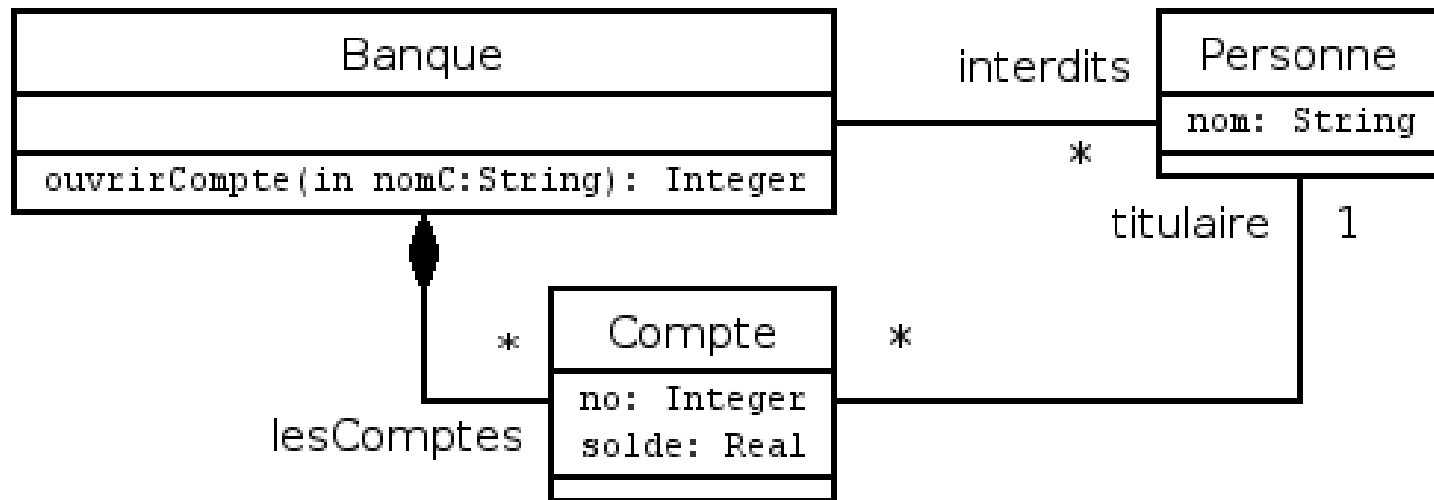


L'opération doit **aussi** conserver l'ensemble des invariants, cardinalités et contraintes des associations ou des classes !

Un exemple simple

Ouverture d'un compte dans une banque...

- ❑ Il existe une liste des interdits bancaires. On ne prend pas en compte qu'une personne devienne « interdite » après ouverture.
- ❑ Il y a une prime d'ouverture de 15 euros
- ❑ Les numéros de compte doivent être distincts
- ❑ Une personne peut avoir plusieurs comptes



L'ouverture de compte en OCL

```
context Banque::ouvrirCompte(nomC: String) : Integer
    signal Refus
pre: Personne.allInstances->one(p | p.nom = nomc)
post: let p: Personne =
        Personne.allInstances->any(nom = nomc)
    in
if interdits->includes(p | p.nom = nomc) then
    implies Refus.sent()
else c.oclIsNew() and c.oclIsTypeOf(Compte)
    and lesComptes->forall(c2 | c2 = c or c2.numero <> c.numero)
    and lesComptes = lesComptes@pre->including(c)
    and result = c.numero and c.solde = 15
    and nbComptes= nbComptes@pre + 1
    and c.titulaire = p
endif
```

Raisonnable ? ;-(

one () : il existe un et un seul tq ...
any () : l'un quelconque tq ...

Notations pour les changements d'états

- ❑ Dans un modèle d'opération, il faut distinguer la valeur en entrée de la valeur en sortie
 - Dans une précondition, on parle toujours de la valeur « avant » !
 - Dans une post-condition v représente la valeur de la variable v en sortie d'opération.
 - `@pre`: dans une post-condition désigne la valeur à l'entrée de l'opération

- ❑ `@pre` s'applique à une propriété (attribut, rôle) de l'objet modifié par l'opération

`lesComptes = lesComptes@pre->including(c)`

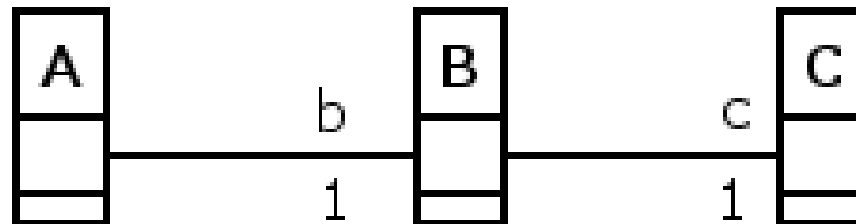
`a@pre` n'a pas de sens dans un invariant ou pour un objet créé par l'opération (`a.oclIsNew()=true`).

Idem pour `result` !



Exemples de @pre

- ❑ `a.b.c` : la nouvelle valeur de l'attribut (ou de la terminaison d'association) `c` pour la nouvelle valeur de l'attribut `b` de `a`.
- ❑ `a.b@pre.c` : la nouvelle valeur de l'attribut `c` pour l'ancienne valeur de l'attribut `b` de `a`.
- ❑ `a.b@pre.c@pre` : l'ancienne valeur de l'attribut `c` pour l'ancienne valeur de l'attribut `b` de `a`.



Invariants, Modèle d'opérations et sous-classes

On doit pouvoir substituer une instance d'une sous-classe à une instance de sa superclasse, tout en respectant les « contrats » de la superclasse:

- 👉 Les **invariants de sa superclasse sont hérités** par une sous-classe et s'ajoutent à ses invariants propres.

De même, si une méthode redéfinit une méthode de la superclasse:

- 👉 sa pré-condition peut être **identique ou moins contraignante** que la précondition de la méthode de base
- 👉 sa post-condition peut être **identique ou plus contraignante** que la postcondition de la méthode de base

Les Types en OCL

❑ Types prédéfinis classiques

- Boolean: `or`, `and`, `=`, `<>`, `implies`,
- `if then else endif`
- Integer: les opérateurs classiques...
- Real: `idem`
- String: `concat`, `size`, `=`, `<>`, `substring(i, j)`, `'Hello'`

Une assertion logique
qui est `true` ou `false`
pas une instruction !

❑ La hiérarchie de `Collection` et ses sous-classes

- Set
- Ordered Set
- Bag
- Sequence

❑ `OclAny`

❑ `OclVoid`

❑ Les classes, énumérations, types du modèle

Aspects syntaxiques

- ❑ Signification de l'opérateur =
 - Classes et interfaces : l'identité d'instances
 - Types, énumérations, collections : l'égalité de valeurs
- ❑ Syntaxe: notation « pointée », sauf pour les opérateurs arithmétiques et logiques classiques
 - `'Hello'.concat(' World')`
- ❑ Littéraux d'une énumération: `Carburant::SP95`
- ❑ Accès à un attribut
 - de classe : `Citerne::nivAlarme`
 - d'instance : `Citerne.niveau`
- ❑ Accès à une terminaison d'association (rôle): `p.achats`
 - ;-(Cas particuliers pour les "classes associations" et associations qualifiées !

A propos de `oclVoid`

`oclVoid`: type avec une unique valeur « `undefined` »

- ❑ Représente la notion de « valeur indéfinie »
 - Une expression est indéfinie si sa précondition n'est pas respectée !

L'indéfinition se propage et enlève toute signification à la spécification, sauf dans les cas suivants:

`true or undefined = true`

`false and undefined = false`

`false implies undefined = true`

`if ... then ... else ...` --selon la branche qui contient l'indéfinition !

- ❑ `e.oclIsUndefined()` : pour tout `e`, permet de savoir si `e` est indéfinie ou non.

A propos de `oclAny`

`oclAny`: le super-type commun à tous les types !

Implicitement toute expression OCL est de ce type, donc les opérations sont bien fondées:

- `e.oclIsUndefined()`
- `e.oclIsNew()`
- `e.oclIsKindOf(oclType)` -- typage au sens large
- `e.oclIsTypeOf(oclType)` -- typage au sens strict
- `e.oclAsType(oclType)` -- trans-typage (si possible)
- `e.oclInState(stateName)` -- pour les machines à états

Les opérations des collections OCL

- ❑ **Syntaxe:** `uneCollection->operation(...)`
- ❑ Dans tous les cas, les opérations n'altèrent pas leurs opérandes. Ce sont des fonctions au sens mathématique qui renvoient une nouvelle collection
 - `s.including(e)` : une « nouvelle » collection qui contient exactement tous les éléments de `s` plus `e`.
- ❑ **Abus de notation :**
 - Si `a.e` est de cardinalité `0..1` on peut écrire directement `a.e.b`.
Résultat indéfini si `a.e->isEmpty()` !

Collections et terminaisons d'associations

- ❑ On peut toujours considérer une extrémité d'association comme une collection à (0, 1, ... éléments)
- ❑ Le type de la collection obtenue dépend de la multiplicité de l'association et de la contrainte associée (*Set*, *Bag*, *Sequence*)
- ❑ Si on traverse plusieurs associations de type *, le résultat obtenu est un *Bag*

```
context Station::getAchats(): Bag(Achats)
```

```
pre: true
```

```
post: result= postes->collect(p| p.achats)
```

```
-- ou plus simplement postes.achats
```

Opérations de collections : `select`, `reject`, `collect`

- ❑ `select`: sélection dans une collection

context `Station::posteManuels(): Collection(PosteManuel)`

post: `result = postes->select(p|p.oclIsKindOf(PosteManuel))`

Raccourci (pour une condition simple) : `postes->select(no > 0)`

- ❑ Pour construire la collection complémentaire: **`reject`**

- ❑ **`collect`** : le bag des résultats d'une opération sur une collection

context `Station::calculeCA(): Real`

post: `result = achats->collect(a| a.prixTotal)->sum()`

Raccourci utilisable en l'absence d'ambiguïté (collect implicite):

`achats.prixTotal->sum()`

- ❑ `collectNested`: maintient la structure d'une collection, alors que `collect` aplatit toujours le résultat !

Opérations de collections: `exists` et `forall`

- ❑ `forall`: conjonction d'une expression booléenne sur tous les éléments d'une collection
- ❑ `exists`: disjonction d'une expression booléenne sur tous les éléments d'une collection

context `s: Station`

inv: `s.postes->forall(p1, p2 | p1 = p2 or p1.no <>p2.no)`

inv: `s.postes->exists(p | p.oclIsKindOf(PosteManuel))`
`and s.postes->exists(p | p.oclIsKindOf(PosteAuto))`

Opérations de collections: any, one

- ❑ `any(expr)` : accès « indéterminé » à un élément vérifiant une condition. Valeur indéfinie s'il n'y en a aucun.
- ❑ `one(expr)` : `true` ssi il existe exactement un élément vérifiant la condition, `false` sinon

```
context Banque::compteValide(num: Integer) : Compte
```

```
    signals Inconnu
```

```
pre: true
```

```
post: let ok: Boolean = lesComptes->one(numero=num)
```

```
    in if ok then result= lesComptes->any(numero=num)
```

```
    else signal Inconnu endif
```

```
context Banque::mainInnocente() : Compte
```

```
pre: lesComptes->notEmpty()
```

```
post: result = lesComptes->any(true) -- n'importe lequel
```

Les collections : autres opérations

- ❑ `size()` : la cardinalité de la collection
- ❑ `count(e)` : le nombre d'occurrences de `e` dans la collection
- ❑ `isUnique(exp)` : `exp` diffère pour tous les éléments de la collection
- ❑ `=, <>` -- entre collections (sens variable selon les types !)
- ❑ `isEmpty(), notEmpty()`
- ❑ `sum()` -- pour une collection de nombres
- ❑ `includes(e), includesAll(collection)` -- présence de `e`
- ❑ `excludes(e), excludesAll(collection)`
- ❑ *`including(e), excluding(e)`* -- ajout de `e`
- ❑ *`union(coll), intersection(coll), ...`*
- ❑ *`flatten()`* -- aplatit (récursivement) une collection
- ❑ *`asSet(), asBag(), asSequence(), asOrderedSet()`*

En italique: les opérations dont le sens dépend des collections auxquelles elles sont appliquées

Opérations spécifiques aux séquences

Ces fonctions profitent de la notion de collection ordonnée (selon l'ordre d'insertion des éléments)

- ❑ `first()`
- ❑ `last()`
- ❑ `at(anInt)` -- pour *anInt* entre 1 et `size()`
- ❑ `append(e)` -- ajout en queue
- ❑ `prepend(e)` -- ajout en tête

En guise de synthèse pour l'analyse

En préalable à l'analyse

- ❑ Partir d'une première version de **l'analyse des besoins** pour aboutir à une première description du système
 - Être raisonnablement sceptique sur les termes utilisés: leur signification pour le client n'est pas forcément la même !
 - ☞ Si besoin, les ré-exprimer différemment, les illustrer par des situations
 - ☞ Si un terme technique est utilisé, expliciter les implications, les hypothèses simplificatrices qu'il amène ou les contraintes supplémentaires
 - ☞ Si besoin, faire un glossaire des termes techniques

En préalable à l'analyse (2)

- Établir clairement ce qui est dans/en-dehors du système
- Établir clairement les contraintes techniques :
niveau de performance, contraintes de portabilité,
environnement matériel et logiciel, type d'interface, ...
- Établir les cas d'utilisations « non-fonctionnels »
(sérialisation, archivage, redondance, module d'échanges
avec d'autres applications)

☞ *Le document doit être validé par le client*

En préalable à l'analyse (3)

- ❑ Le client est peu disponible, et n'est pas un « pote »
 - Nommer un responsable des interactions avec le client
 - Préparer les réunions (documents, questions, déroulement)
 - Chaque réunion doit faire l'objet d'un compte-rendu
 - Les comptes-rendus sont diffusés et archivés

- ❑ Le client n'est pas forcément de votre domaine
 - Il n'anticipe pas forcément les limitations techniques, les implications en termes de délai, de coût, de technologie
 - Il connaît bien son domaine et oublie que vous ne le connaissez pas
 - Il y a des « non-dits », des aspects implicites (« évidences »)
 - Il ne connaît pas forcément exactement ses besoins, ou la façon de les exprimer clairement

Et réciproquement de votre part ...

- ❑ Vous avez une obligation de conseil et d'écoute critique !
- ❑ Il ne parle pas forcément UML: du texte ? Des maquettes ?

Le processus d'analyse

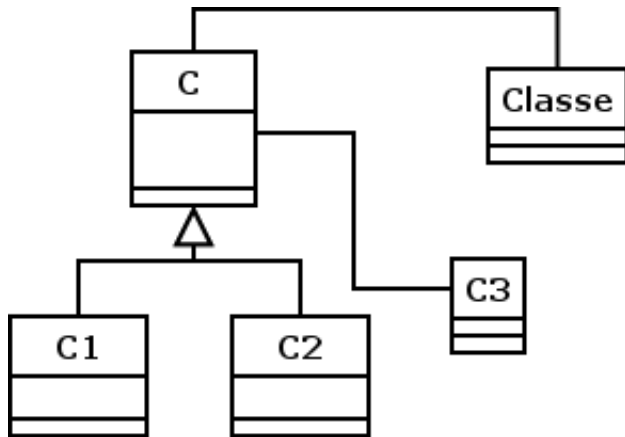
- ❑ Identifier les cas d'utilisation potentiels, les acteurs associés
- ❑ Produire un premier « *diagramme des classes du domaine* » en privilégiant classes élémentaires et associations
 - Identifier les termes du domaine (vocabulaire ou « abstractions » du monde modélisé) :
 - ☞ noms communs, personnes, qualificatifs, attributs, propriétés
 - Identifier les « responsabilités » associées à ces termes
 - Identifier qui est « moteur », qui est « sujet »
 - Identifier
 - qui a besoin de « communiquer » avec qui,
 - qui « partage de l'information » avec qui,
 - qui est « en première ligne », qui n'est qu'un support ...

Processus d'analyse (2)

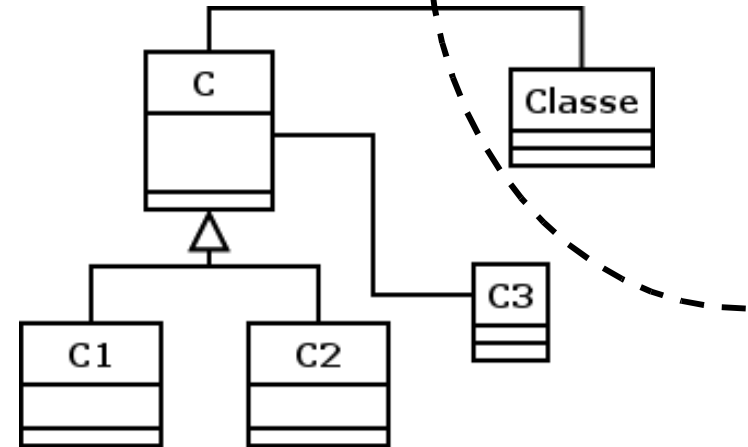
Raffiner ce premier diagramme

- Équilibrer les responsabilités entre abstractions (☹ une classe qui fait tout !)
- Éviter la dilution des responsabilités (regrouper les classes séparées mais inter-dépendantes)
- Distinguer entre abstractions et propriétés de ces abstractions (attributs)
- Distinguer les classes sans opérations naturelles (ce sont des types, pas des classes)
- Tracer les relations entre ces abstractions

Délimiter la frontière du système



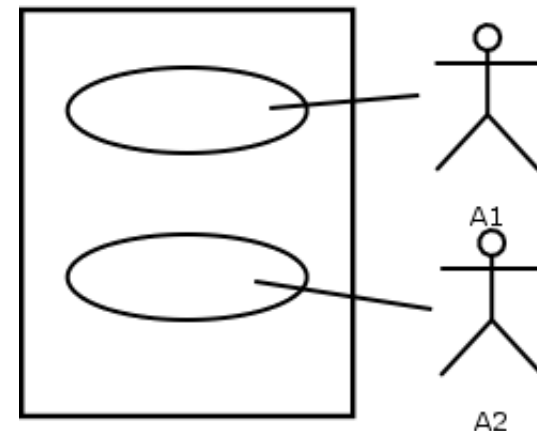
DCD: le domaine



DCS: le système

Déterminer ce qui dans le système

- 👉 En déduire les acteurs
- 👉 Faut-il une représentation des acteurs dans le système ?
- 👉 Mettre en évidence les interactions entre les acteurs et le système



Établir le premier modèle d'analyse

- ❑ Écrire le diagramme des cas d'utilisation
 - ❑ Établir le modèle des classes
 - ❑ Établir des scénarios d'utilisation
 - ❑ Exprimer les invariants, les modèles d'opérations
 - ❑ Déterminer si on a les bons « observateurs »
 - ❑ Dériver des tests d'acceptation des scénarios: en parallèle avec l'analyse, on prépare les tests fonctionnels !
 - Quels enchaînements d'opérations à tester ?
 - Quels paramètres
 - Quelles instances pour leur exécution
 - Comment les mettre dans le bon état ?
 - Comment tester le résultat de l'enchaînement des opérations et l'état final du système
- } *Environnement de test*

Interface du système avec l'extérieur

- ❑ Introduire des classes d'interface (« classes actives ») pour gérer la communication avec les acteurs
- ☞ Comment passer d'une représentation externe à l'instance qui lui correspond dans le système ?
 - ☞ Le système d'information de la Banque connaît des « Clients »
 - ☞ Pour entrer dans le système un client s'identifie grâce à des caractéristiques externes (nom, numéro de compte, cookie, ...)
- ☞ Avec les acteurs on n'échange généralement pas des objets puisqu'ils sont à l'extérieur du système !
- ☞ Comment passe-t-on du nom à l'instance ?
 - ☞ Table de hachage: nom -> Client ?
 - ☞ « mapping » relationnel + clef primaire ?
 - ☞ Collection statique de tous les clients + méthode de recherche ?
- ❑ Ces classes seront affinées lors de la conception !

Établir le « dictionnaire des données »

- ❑ Définit les identificateurs dans leur contexte
- ❑ En donne une description synthétique et claire
- ❑ Initié pendant l'analyse et enrichi ensuite

Nom	Catégorie	Description	Contexte
Poste	classe	texte court et explicatif !	n° § ou n°page
...	type		
...	acteur		
no	attribut		
achats	rôle		
...	...		

Documents à l'issue de l'analyse

- ❑ Cas d'utilisations, avec leurs spécifications (textuelles et sous forme de diagrammes de séquence)
- ❑ Scénarios qui illustrent la dynamique du système global
- ❑ Modèles des classes et invariants associés
- ❑ Modèles des principales opérations des cas d'utilisation
- ❑ Tests d'acceptation déduits des cas d'utilisation et des scénarios
- ❑ Dictionnaire des données
- ❑ Une version préliminaire du manuel utilisateur

Format des Documents

- ☞ Tout document doit être paginé et muni d'une table des matières
- ☞ Tout document doit être daté, muni d'un numéro de version, avoir une indication du rédacteur responsable
- ☞ Une page de garde doit résumer les principaux points modifiés par rapport à la version précédente (« historique » du document)
- ☞ Les versions doivent être archivées soigneusement: l'équipe doit conserver un exemplaire de chaque version.
- ☞ Il est conseillé d'effectuer des lectures croisées et « sceptiques » des documents
- ☞ Il y a un auteur mais l'équipe est collectivement responsable.