



UML pratique avec RSA

- **Introduction**

- Évolution des langages de programmation
- Motivations des technologies objet
- Motivations pour une notation unifiée
- UML et RSA

- **Les fondements objets**

- Encapsulation, attributs, méthodes
- Notion d'état et de comportement
- Fonction/procédure vs méthode
- Communication entre objets : Message
- La recherche de méthode : le polymorphisme
- Concepts de classe et d'instances

- Diagrammes de classe
- Diagramme de séquence

- **Les relations**

- Association
- Agrégation
- Composition
- Cardinalité
- Rôles dans une relation
- Héritage
- Représentation UML des relations

- **Les concepts avancés**

- Construction / destruction
- Classe abstraite
- Typage et interface
- Couplage fort / faible
- Diagramme d'état
- Stéréotype

- **Identifier les objets**

- Objet métier, objet secondaire,
- Objet passif, objet actif (Acteur)
- Identifier les besoins
- Modéliser les exigences et le métier
- Diagramme de cas d'utilisation
- Relations entre cas d'utilisation
- Diagramme d'activité

- **Les autres diagrammes**
 - Diagramme de packages
 - Diagramme de composants
 - Diagramme de structure composite
 - Diagramme de déploiement
 - Diagramme de temps
 - Récapitulatif des diagrammes
 - Extensions d'UML
 - BPMN, SysML, ...
- **Conclusion**

Introduction

- **Langage machine**

- Codage en hexadécimal des instructions et des données

- **Assembleur**

- Des instructions un peu plus lisibles (move, jump, ...)
- Très proche de la machine (notions de registre, d'adresse mémoire, ...)

- **Programmation structurée**

- Élimination des **GOTO** au profit des procédures (sous-routines)
- Le programme est structuré par une « décomposition fonctionnelle »

- **Programmation fonctionnelle**

- Théorie de l'information : tout est fonction, constituée d'autres fonctions
- Très puissant mais très théorique avec des concepts très abstraits

- **Programmation objet / logique / L4G**

- Décrire ce que l'on veut faire plutôt que comment le faire

V
E
R
S

P
L
U
S

D
'
A
B
S
T
R
A
C
T
I
O
N

- **Approche classique de programmation**

- Des variables pour représenter des zones mémoires
- Des opérations de calcul et d'affectation pour changer la mémoire
- Des structures de contrôle (branchement conditionnel, boucle, ...)
- Notion de fonction et de procédure permet la décomposition fonctionnelle

- **Les années 80 voient l'industrialisation de l'informatique**

- PC, langages, base de données, réseaux, méthodologies, ...
- Mais complexité croissante des applications, de plus en plus difficiles à maintenir

- **Il faut trouver des solutions pour améliorer les « -ilities »**

- Adaptability, maintainability, lisibility, stability, usability, testability, reusability, ...
- La première crise du logiciel fait s'intéresser aux technologies objets naissantes qui promettent réutilisation, robustesse et diminution drastique des coûts
- La **POO** (Programmation Orientée Objet) s'impose alors progressivement

- **Entrée progressive par le biais des interfaces graphiques**

- Le concept d'objet est d'abord tactile avec l'apparition de la souris et des widgets
- Puis il s'impose dans des projets de plus en plus importants
- Pour venir au cœur des projets de refontes des systèmes d'information

- **Problème**

- Inadéquation des méthodes classiques (MERISE, SADT, ...)
 - Elles ne peuvent pas prendre en compte les concepts spécifiques de la POO
 - Chacun y va de sa propre méthodologie et notation
- Au début des années 90, il existera plus de 50 méthodes « orientée objet »
 - Chacune accompagnée de sa propre notation pour représenter les concepts
- Trois s'imposeront par leur rigueur et complétude : OMT, OOD et OOSE
 - Mais le passage d'un projet à l'autre par les équipes projets nécessite d'unifier les notations
 - Il faut un standard et aucune ne peut s'imposer comme un standard
- Leur synthèse donnera **UML** : Unified Modeling Language

- **UML : une notation, pas une méthodologie**
 - Il est illusoire de vouloir imposer une démarche méthodologique commune
 - Chaque domaine a ses propres habitudes et ses propres contraintes
 - L'adoption d'une notation ne remet pas en cause l'organisation et les manières de faire
 - On se met « seulement » d'accord sur la notation pour représenter les concepts
- **UML : un langage (graphique) de modélisation**
 - Ces concepts objets sont très liés aux langages
 - Smalltalk, C++, Eiffel, CLOS, Objective-C, Pascal objet, ... (Java apparaîtra plus tard)
 - Il faut d'abord se mettre d'accord sur les concepts essentiels et communs
 - Et garantir ainsi qu'on restera à un niveau d'abstraction suffisant pour modéliser le système

- **Une notation pour concevoir des logiciels orientés objet**
 - Née au milieu des années 90
- **Inspirée par les méthodes précédentes dont :**
 - Booch, la méthode de Grady Booch
 - OMT (Object Modeling Technique) de James Rumbaugh
 - OOSE (Object Oriented Software Engineering) de Ivar Jacobson
- **Devenue un standard OMG en 1997**
 - OMG = Object Management Group
 - Une association visant la promotion et la standardisation du modèle objet sous toutes ses formes
 - L'OMG est aussi à la base du standard CORBA
 - Et des spécifications associées comme l'IDL
 - CORBA : norme pour les architectures à base d'objets distribués
 - Ainsi que d'autres normes et recommandations
 - BPMN, MOF, MDA, QVT, ...



- **Unified**

- La **POO** (Programmation Orientée Objet) s'est imposée à la fin des années 80
- Très vite, il a fallu créer des méthodes de travail spécifiques
 - MERISE ne permet pas de représenter les concepts particuliers de la POO
 - Vers 1994, on comptait plus de 50 méthodes et notations
- UML unifie les différentes notations pour s'imposer comme un **standard commun**

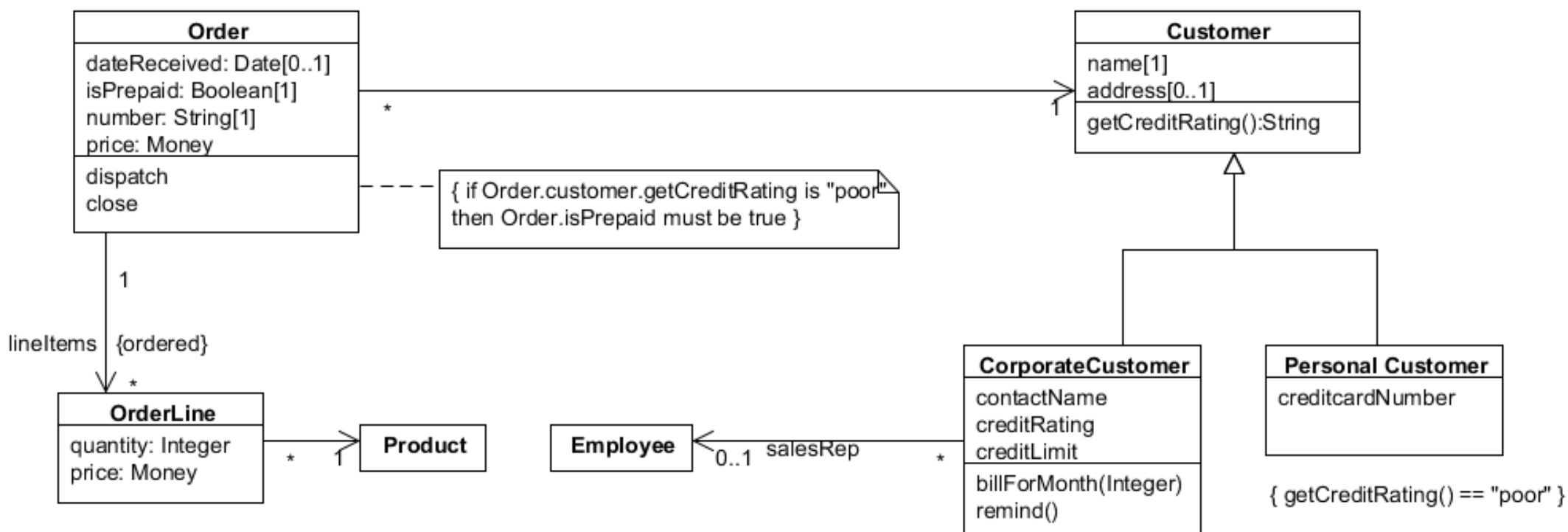
- **Modeling**

- UML permet de concevoir un **modèle du logiciel** à construire
- Un modèle permet de réfléchir, extrapoler, exposer, discuter, prédire, estimer, ...

- **Language**

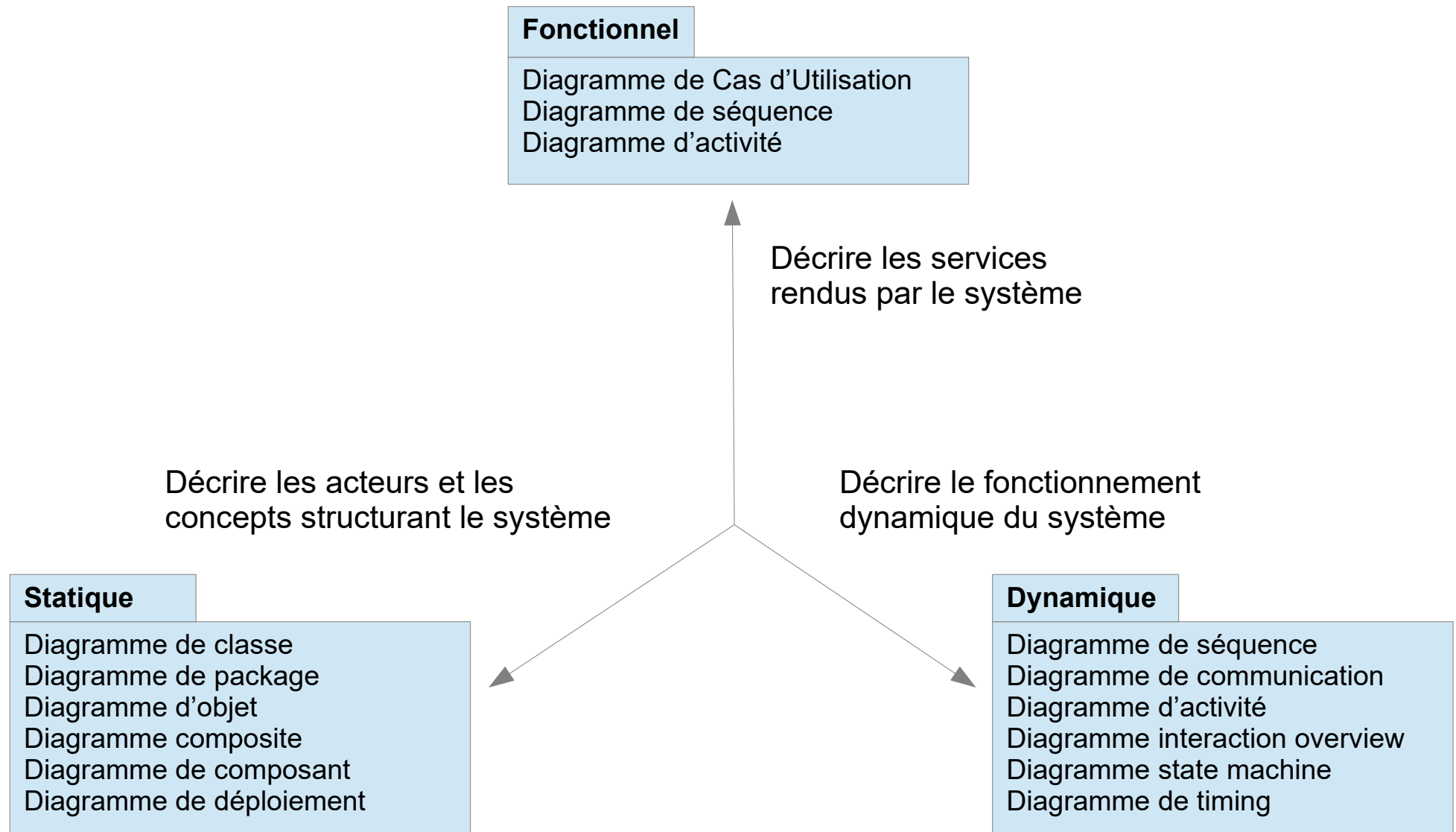
- UML est une notation, c'est-à-dire un **langage graphique**
 - Il permet d'exprimer les concepts et les modèles sous forme de schémas
 - C'est le langage unifié des informaticiens pour mettre à plat et communiquer les idées
- Dans sa version 2, c'est aussi un **langage textuel** (un dialecte **XML** appelé **XMI**)
 - Cela permet d'envisager la création d'outils standards (générateur de documentation, de code, ...)

- Question 1 : décrire ce schéma (que voyez-vous sur ce schéma ?)
- Question 2 : que comprenez-vous ? (on peut deviner des choses)



- **Toute construction nécessite des plans au préalable**
 - Nécessaires pour réfléchir, expérimenter, consolider, communiquer, valider, ...
- **Différents points de vue sont nécessaires**
 - Pour les différents corps de métier, pour représenter sous plusieurs angles, ...
- **Ces plans doivent être compréhensibles de tous**
 - En tout cas, de ceux à qui ils sont destinés
 - Il doivent respecter les mêmes conventions
- **UML est construit en respectant ces mêmes principes**
 - Une notation unifiée et complète pour représenter toutes les situations
 - Dédiée à toutes les activités et corps de métier d'un projet informatique
 - En offrant de nombreux types de diagrammes (points de vue différents)
 - Pas moins de 15 types de diagrammes différents dans les dernières spécifications

- Les différents diagrammes sont organisés selon 3 axes



- **Il n'est pas nécessaire de tous les utiliser**

- Certains n'existent que pour répondre à des besoins bien précis
- D'autres sont redondants

- **Il n'est pas nécessaire d'utiliser toute la notation**

- La notation est très (trop) riche
- Certains estiment que 20% sont suffisants pour couvrir les besoins courants
- Il vaut mieux utiliser un sous ensemble réduit de la norme compris par tous

- **Les schémas n'éliminent pas l'utilisation du texte**

- Accompagnez d'explications les schémas complexes
- Décrivez les éléments particuliers utilisés (pour être sûr d'être compris)
- Décrivez les utilisations spécifiques d'un élément de notation

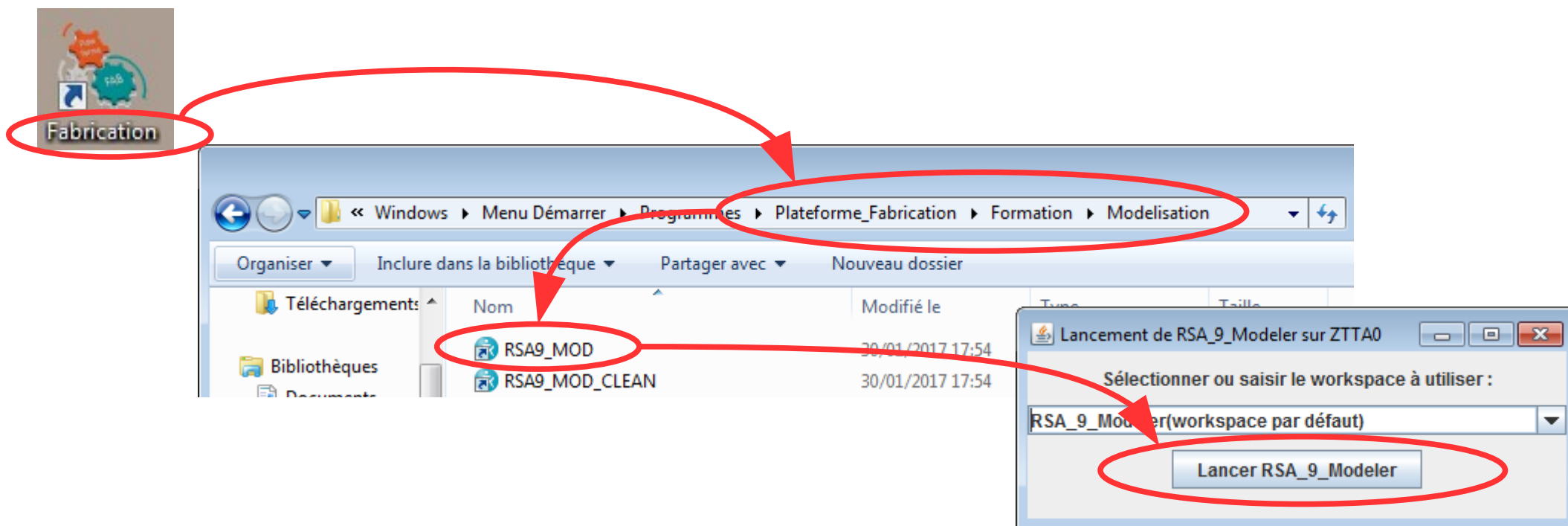
- **A garder à l'esprit**

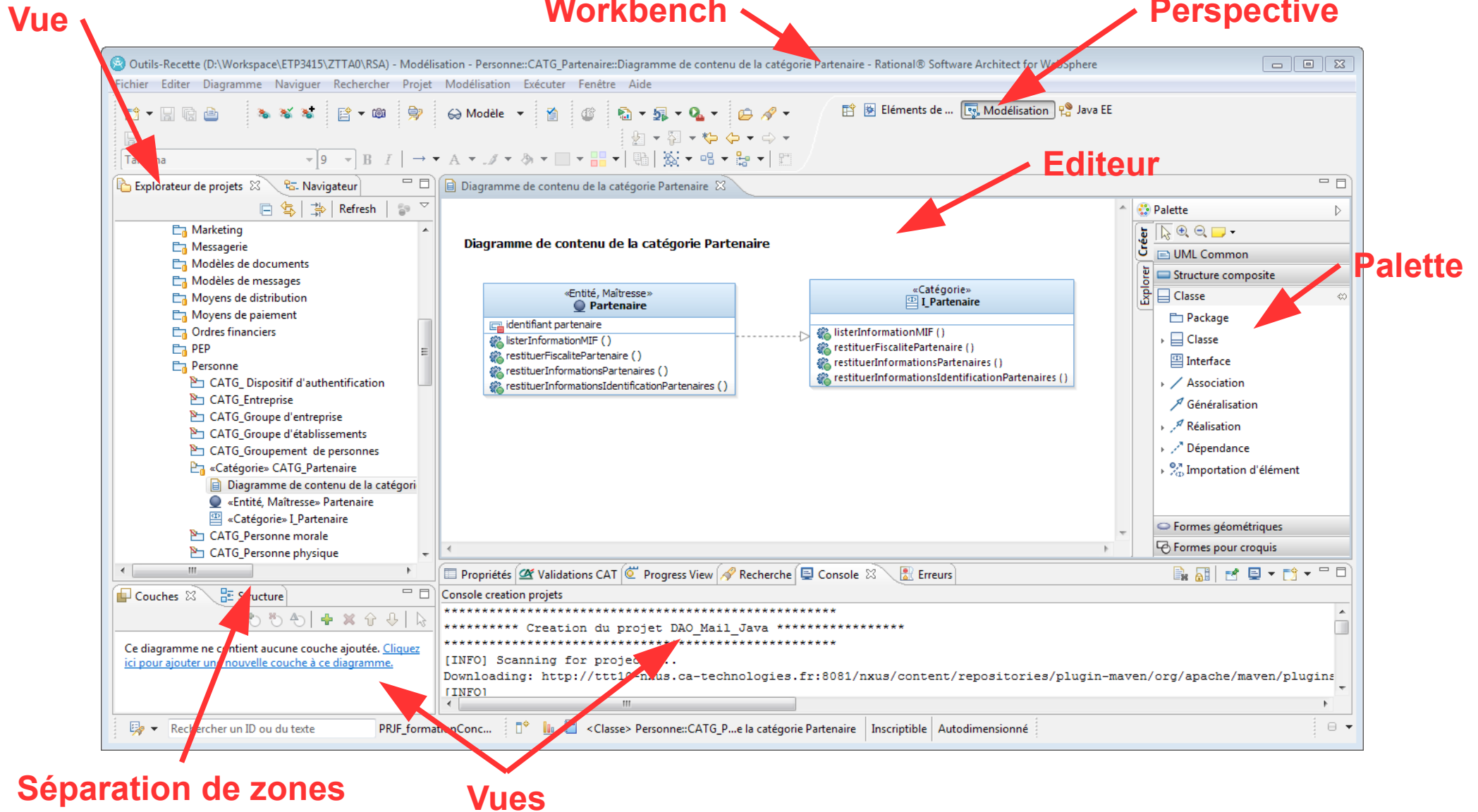
- Un modèle sert à communiquer => s'assurer qu'il reste lisible
- Un modèle doit être simple à comprendre => respecter la notation
- Un modèle structure le système => faire plusieurs diagrammes
- Un modèle doit être précis => il sert de référence
- UML2 a beaucoup évolué => n'utiliser que les diagrammes utiles
- UML n'est pas une méthode => c'est uniquement une notation
- Une démarche/méthode doit être définie => aide à la modélisation

- **UML peut s'utiliser pour faire des croquis**
 - Permet d'exposer ses idées de conception sur un tableau blanc ou un papier
 - Offre une vision synthétique des idées autour desquelles discuter
 - Constitue le support d'une discussion pour entrer dans les détails
- **UML peut s'utiliser pour faire les plans à l'encre bleue**
 - Des architectes conçoivent le logiciel puis donnent les plans aux exécutants
 - Dans ce cadre, la notation doit permettre d'exprimer tous les détails nécessaires
 - Les développeurs doivent alors traduire les plans en code et en composants
 - Ils doivent imaginer les détails et adapter la conception aux contraintes
- **UML peut être utilisé pour aller plus loin**
 - Très tôt, les concepteurs ont souhaité aller vers de la **génération** de code
 - Cela a conduit à la version 2 d'UML et une notation textuelle interprétable informatiquement
 - En poussant la conception encore plus dans les détails, on peut générer
 - Des documents, du code, des métriques, des composants, d'autres diagrammes, ...

- **Certains outils se contentent de produire des dessins**
 - C'est le cas d'**UMLet** et sa version web **UMLetino**
 - Ils sont simples mais n'assistent pas le concepteur qui doit maîtriser la notation
 - Pratique pour faire un croquis et le coller dans un document
- **D'autres constituent un modèle complet**
 - Des outils comme **StarUML**, **EA**, **RSA Modeler**, **MagicDraw**, ...
 - Ils guident le concepteur en lui interdisant les constructions incohérentes
 - Ils peuvent être très complexes et réclamer une longue phase d'apprentissage
 - Ils sont plutôt dédiés à faire les plans et à générer
- **Certains concepteurs rejettent ces outils trop complexes**
 - Pour eux, le meilleur outil de conception est « papier/crayon/scanner »

- **RSA = Environnement intégré (modélisation, développement, ...)**
 - Basé sur Eclipse, il intègre les outils de la suite Rational (UML, RTC, ...)
 - RTC = Rational Team Concert (gestion de version et de configuration)
- **Lancement à partir de l'icône Fabrication**
 - Aller dans une branche (**Formation** ou **Branche_A**)
 - Puis aller dans **Modelisation** > **RSA9_MOD**





Détails page suivante

- **Workspace**

- Répertoire de stockage des informations (les fichiers des projets)
 - Un workspace par défaut est créé automatiquement par le script de lancement
 - Il correspond au répertoire **D:\Workspace\ZTTA0\RSA_9_Modeler** (branche **Formation**)

- **Workbench**

- La fenêtre globale de travail (en général unique)

- **Vue : fenêtre interne d'affichage**

- Permet d'afficher les informations des projets ou d'un élément sélectionné
 - Souvent sous forme de liste hiérarchique
- De nombreuses vues sont disponibles (Fenêtre > Afficher la vue > ...)

- **Editeur : variante de vue permettant de modifier les informations**

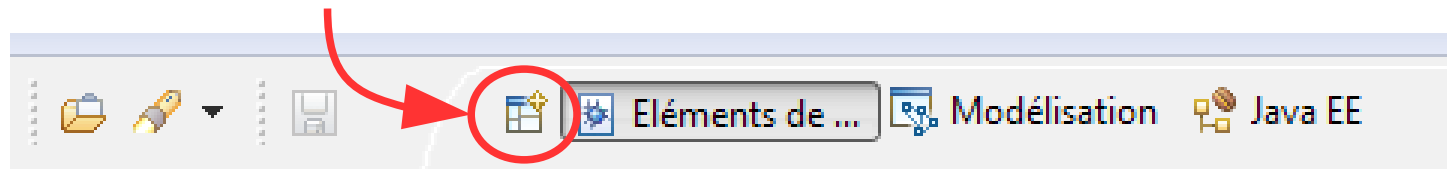
- Les plus simples sont textuels et permettent de modifier le contenu des fichiers
- Les éditeurs graphiques tels que éditeurs UML comportent des palettes d'outils

- **Disposition des vues et éditeurs**

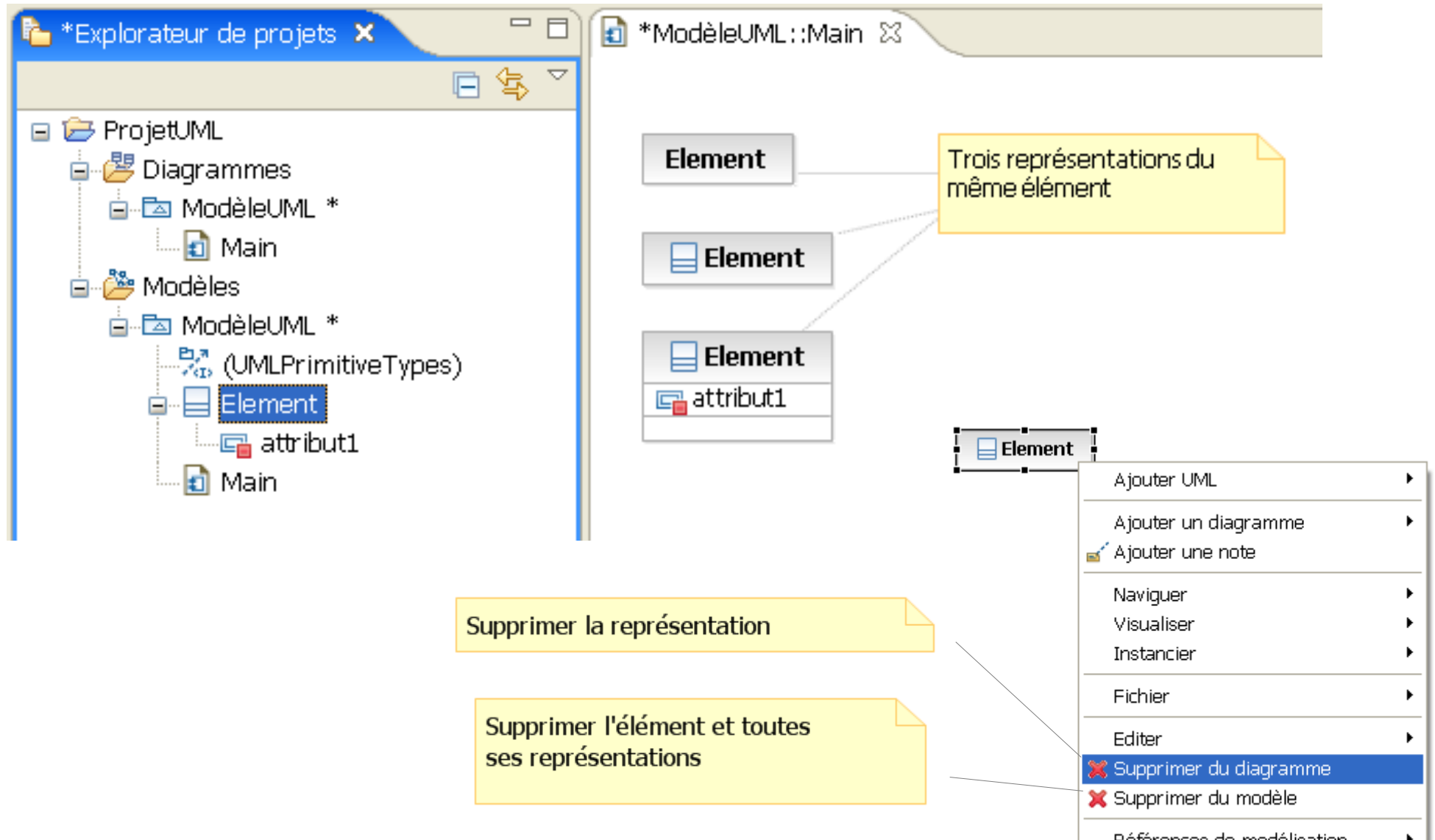
- Les vues sont rangées dans différentes zones pouvant être retaillées à la souris
- Ces vues peuvent être déplacées et empilées dans une même zone (onglets)
 - Astuce : un double clic dans l'onglet d'une vue permet de la positionner en plein écran
- Une vue peut aussi être fermée ou repliée si elle n'est pas utilisée

- **Perspective : disposition prédéfinie du Workbench**

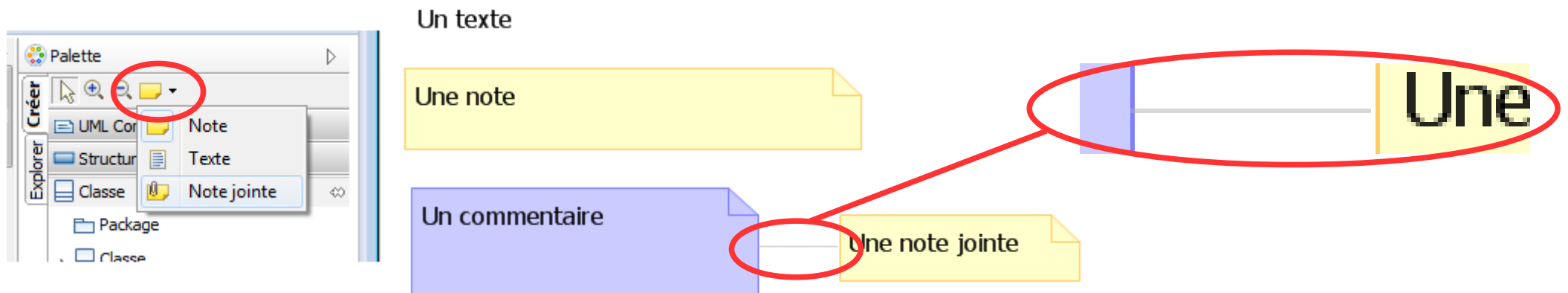
- Cible une activité spécifique (modéliser, coder, gérer les versions, exécuter, ...)
- Positionne les zones, les vues affichées par défaut, les menus spécifiques, etc.
- On peut toujours restaurer la disposition d'origine
 - Fenêtre > réinitialiser la perspective (utile si on a fermé ou déplacé des vues par inadvertance)
- Pour ouvrir une autre perspective
 - Menu Fenêtre ou bien bouton en haut à droite du Workbench



- Les outils UML évolués dissocient le modèle de sa représentation



- **Un diagramme doit parfois être documenté**
 - Pour faciliter sa compréhension, préciser un point particulier, ...
 - C'est l'équivalent des commentaires dans un langage de programmation
- **Trois formes possibles**
 - Le **texte libre**, pour mettre un titre sur le diagramme par exemple
 - Le **commentaire**, pour préciser des informations de documentation
 - La **note**, une sorte de post-it pour attirer l'attention sur un point
- **Les notes et commentaires peuvent être liés à un élément**



- **Les concepts objets visent à simplifier la communication**

- On cherche à décrire d'une manière simple, uniforme et abstraite
- En utilisant le moins possible de concepts techniques

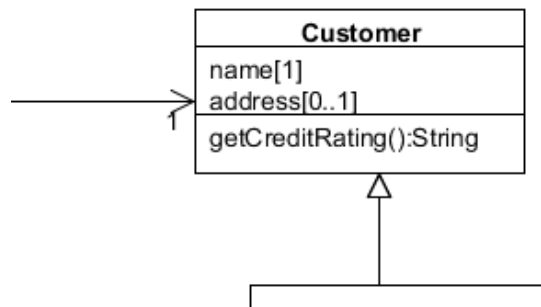
- **Objet = Chose nommée**

- Tout élément constituant le système est décrit comme une **chose**
 - Client, Compte, ...
- C'est valable aussi pour des concepts plus abstraits
 - Solde, événement, parenté, ...
- C'est valable aussi pour les éléments externes au système
 - Systèmes externes, organisations, individus, contraintes...

- **Objet = Sujet (au sens « objet de la conversation »)**

- Toute activité de réflexion (analyse, description, conception...) nécessite de définir et de décrire les **choses** dont on parle

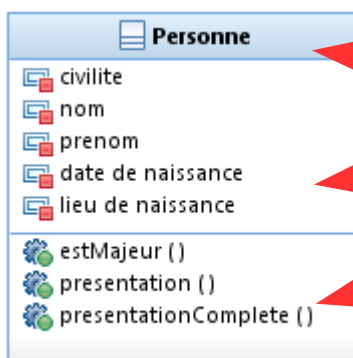
- **Un objet (chose) est décrit en terme d'attributs : ses propriétés**
 - On décrit ainsi ses caractéristiques propres
- **Un objet (chose) est décrit aussi en terme d'opérations**
 - Ce sont les traitements mis à disposition par l'objet
 - Ces traitements seront activés en lui envoyant des **messages**



Le système traite de **Client**
Chaque client comporte un **nom**
Ainsi qu'une éventuelle **adresse**
Chaque client permet d'obtenir sa **solvabilité**

- **La philosophie objet est, en quelque sorte, une vision animiste**
 - Le système est constitué d'objets intelligents qui communiquent entre eux

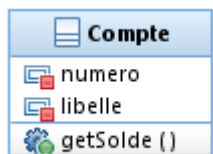
- **La modélisation s'intéresse à la description de cas généraux**
 - Ex : on veut décrire la notion de **Personne** en général, pas **Martin DUPONT**
- **On décrit donc des types d'objets qu'on appelle des classes**
 - **Remarque** : il existe aussi une notation UML pour représenter une **instance**
 - Instance = occurrence, exemplaire
 - Elle est surtout utilisée dans des croquis au tableau ou sur papier, rarement dans RSA
- **Description graphique d'une classe en UML : trois compartiments**



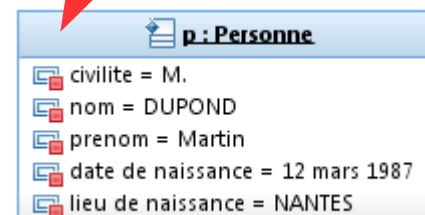
Nom de la classe

Attributs

Opérations



Instance (souligné)
(jamais utilisé dans les outils)

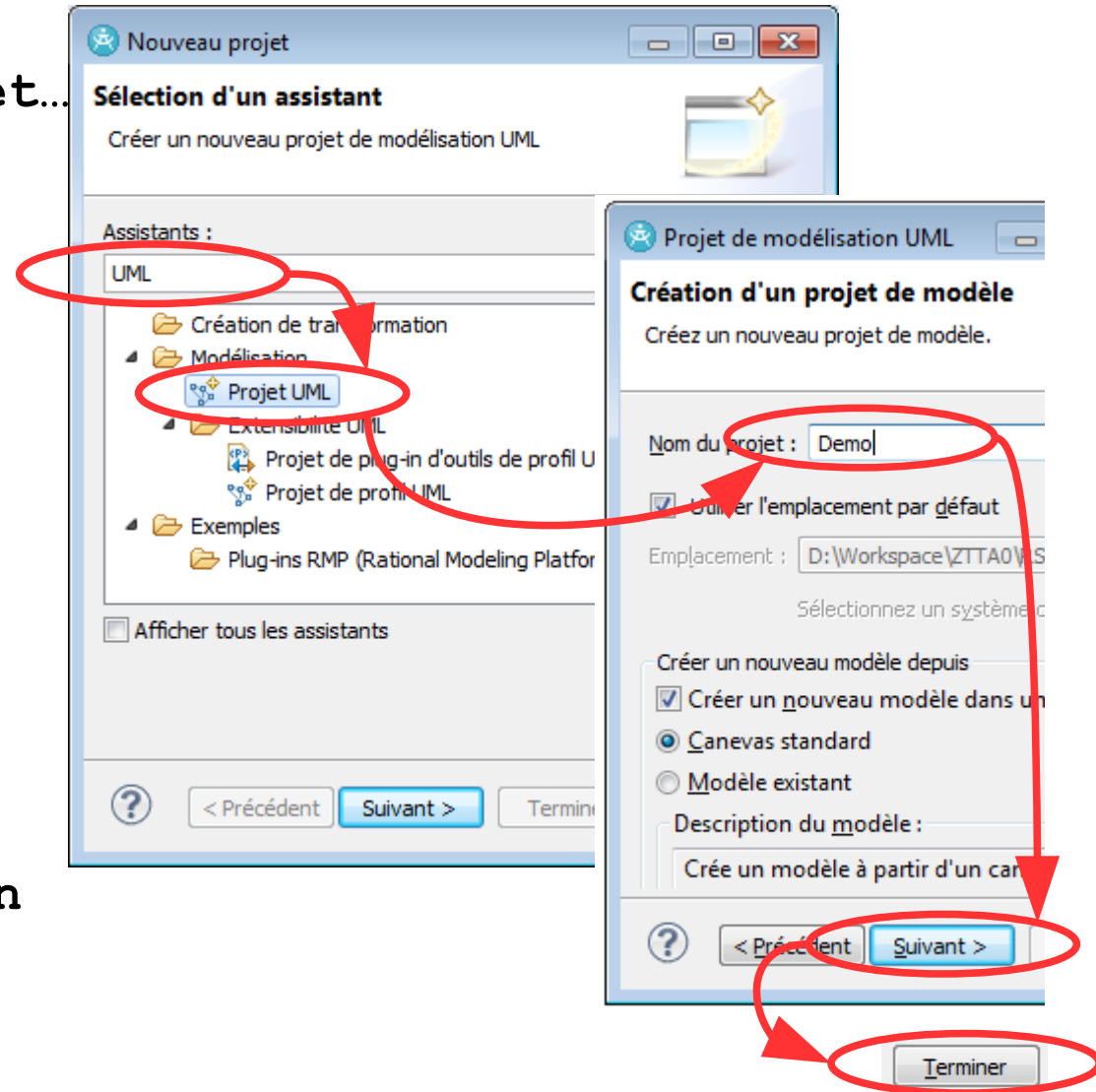


• Création d'un projet de modélisation (perspective Modélisation)

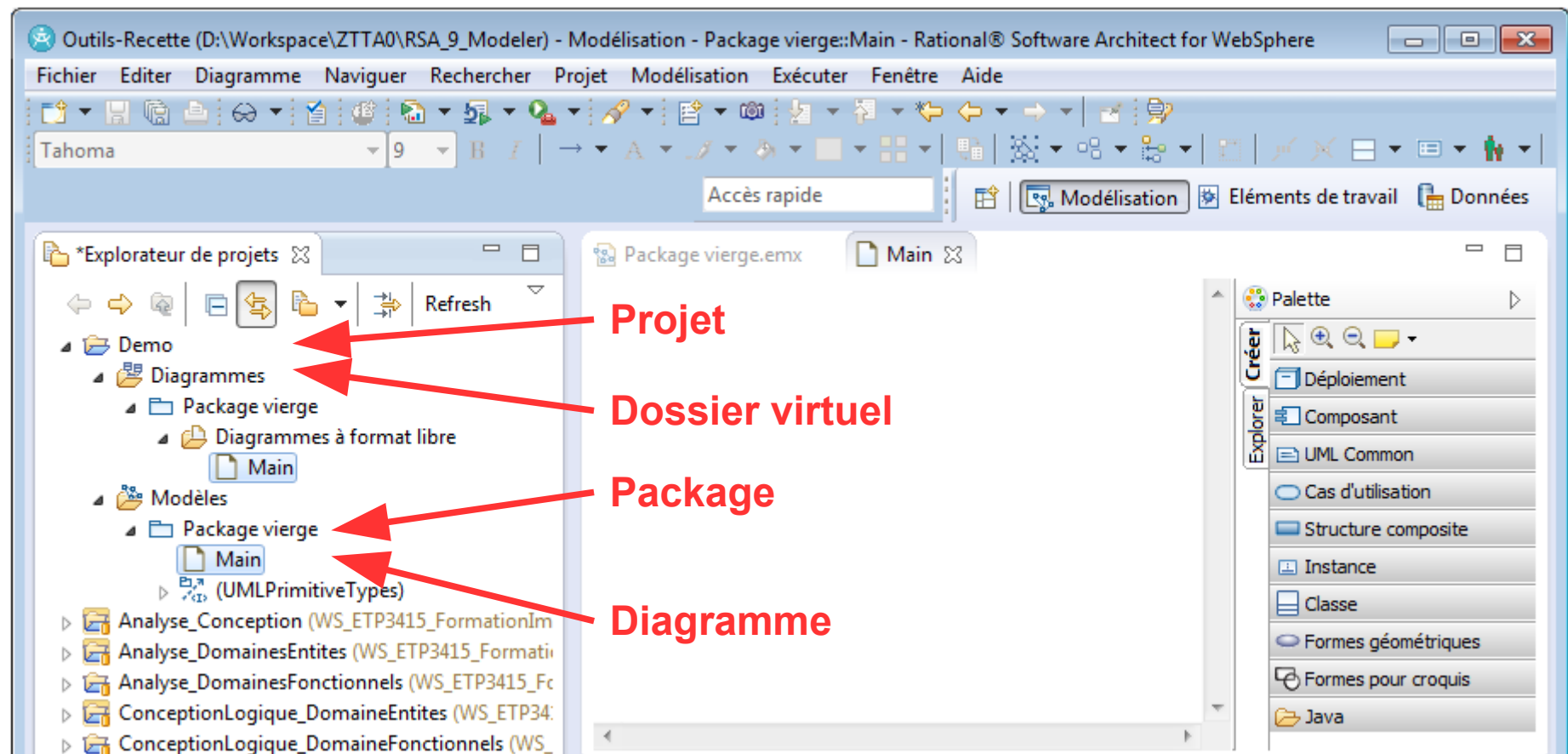
- Dans l'explorateur de projet
Clic-droit > Nouveau > Projet...
- Filtrer en saisissant **UML**
- Choisir **Projet UML**
- Nommer le projet **Demo**
- Laisser les autres choix par défaut

• Remarque

- Vous n'aurez jamais à faire ces actions car les projets de conception existent et sont partagés dans le projet transverse **Analyse_Conception**

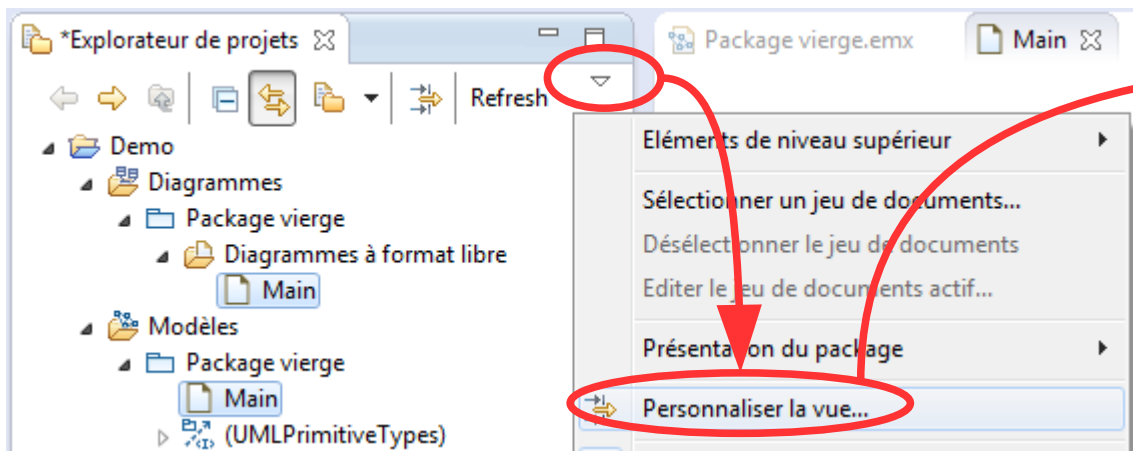


- **Le projet Demo est créé**
 - Il contient un **Package** (Package vierge) et un **Diagramme** (Main)
 - Un package est un **conteneur UML** (très similaire à des dossiers de systèmes de fichiers)
 - Les **dossiers virtuels** créés par RSA sont censés nous aider (mais créés de la confusion)

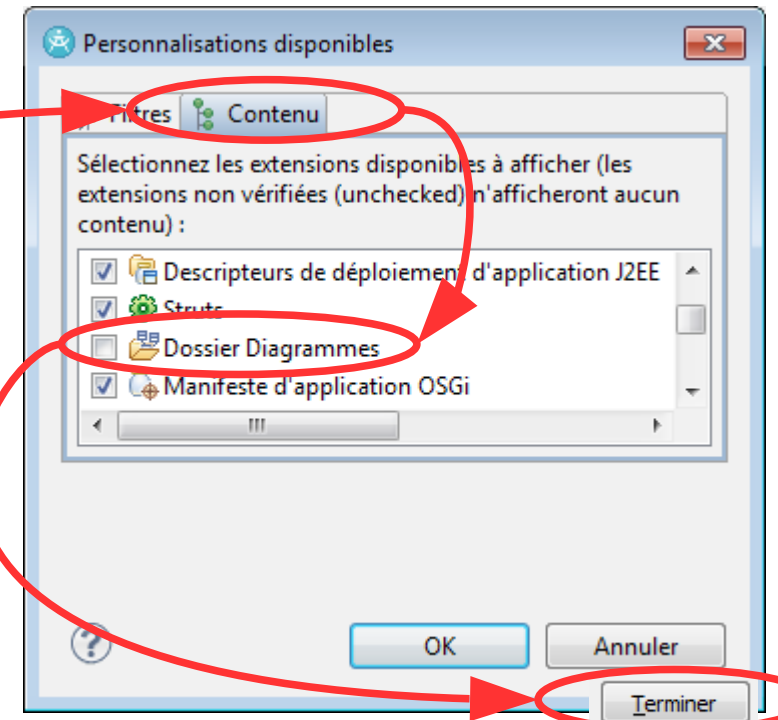
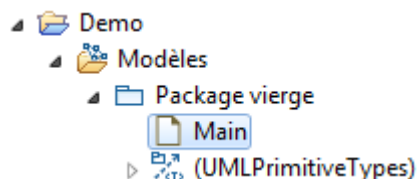


- **La vue Explorateur de projets utilise des dossiers virtuels**
 - **Diagrammes** : affiche les packages et ne montre que les diagrammes
 - **Modèles** : affiche les packages et montre tous les éléments du modèle
- **Le dossier virtuel Diagrammes est source de confusion**
 - Astuce : il est possible de configurer RSA pour ne plus l'afficher

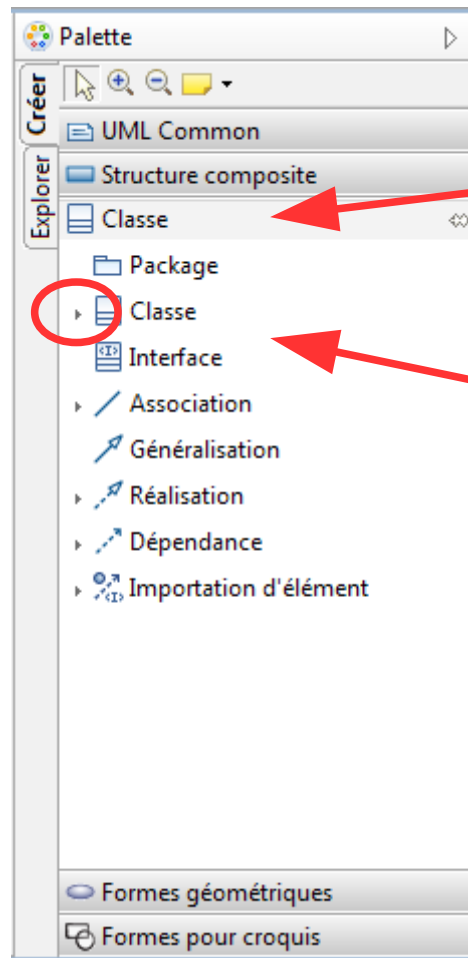
Avant



Après

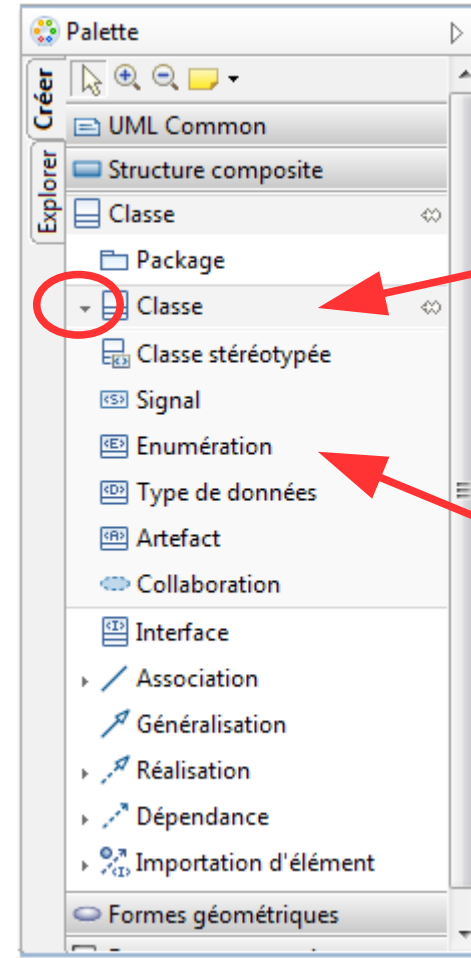


- La palette est contextuelle (elle dépend du diagramme)
 - Les deux niveaux de tiroirs sont peu intuitifs



Tiroir déplié

**Outil « Classe »
(par défaut)**

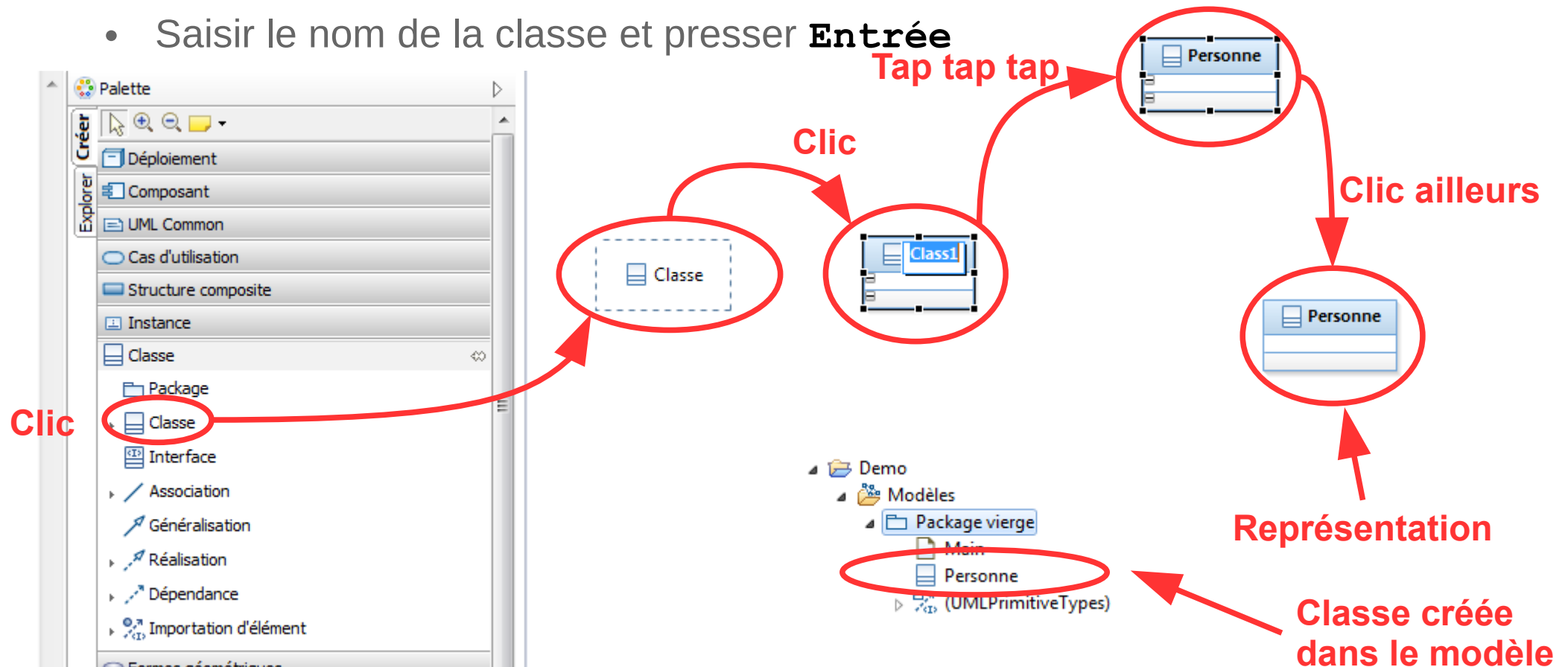


**Sous tiroir
déplié**

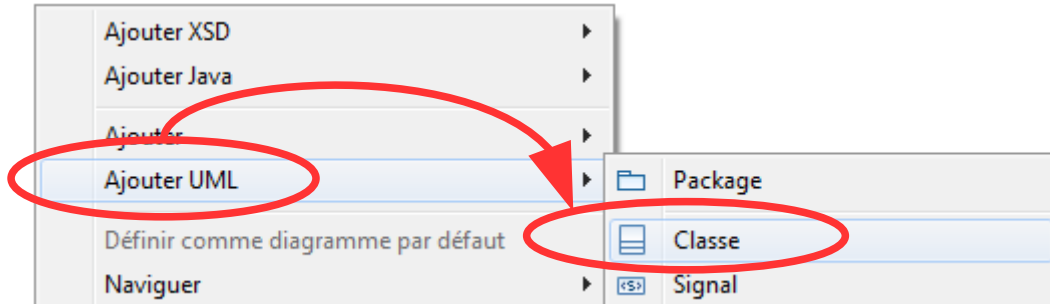
**L'outil sélectionné
devient le nouvel
outil par défaut**

- **Sélectionner l'outil Classe dans la palette (cliquer-relacher)**

- Le curseur change, la classe en construction suit les déplacements du curseur
- **Cliquer** dans le diagramme pour déposer la classe
 - Ou touche **Echap** pour annuler
- Saisir le nom de la classe et presser **Entrée**



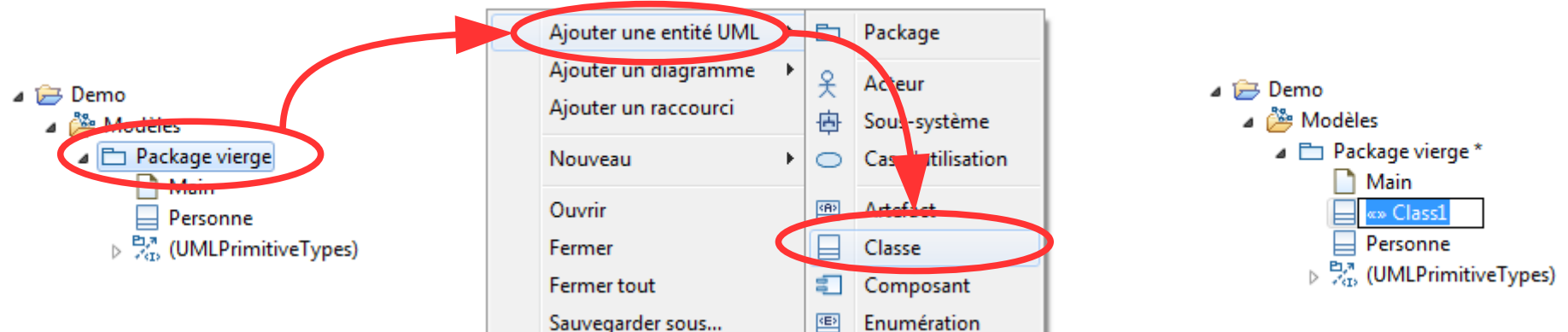
- Autre moyen de créer une classe : Clic-droit dans le diagramme



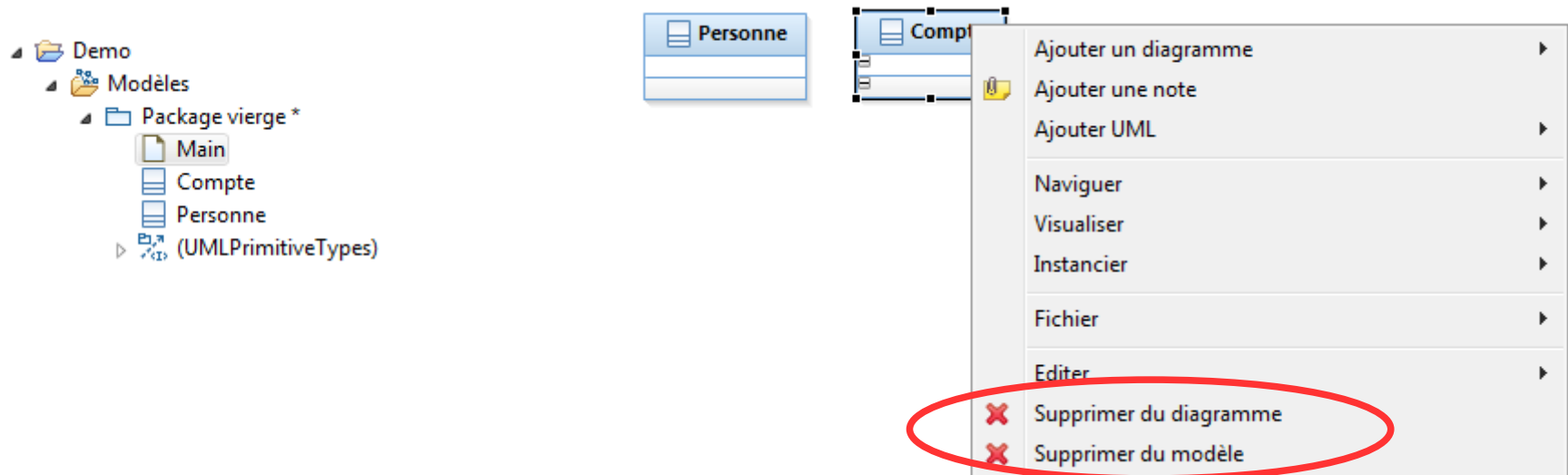
- Ou encore, immobiliser la souris pendant 2s dans le diagramme



- Ou enfin, créer l'élément dans l'arborescence
 - Puis le glisser/déposer dans le diagramme pour créer une représentation



- Expérimenter les suppressions (**Ctrl-z** pour annuler)
 - Dans le diagramme

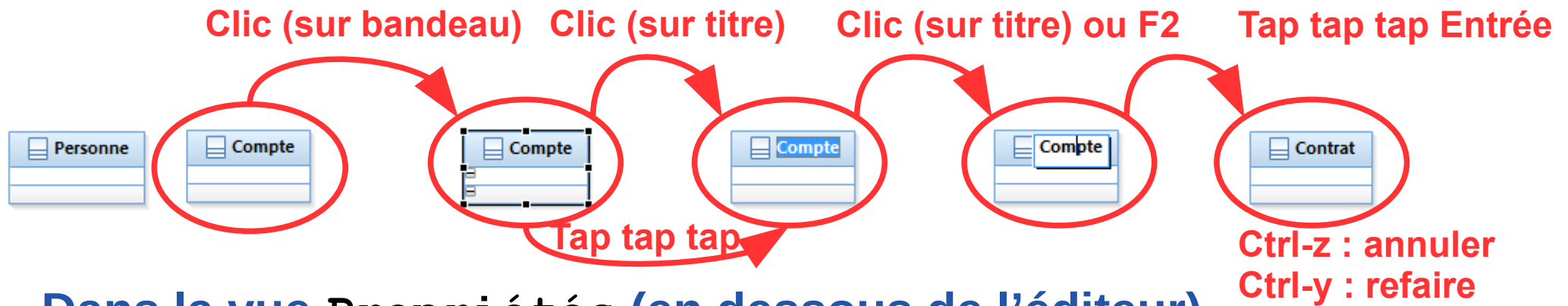


- Dans le modèle (l'arborescence)

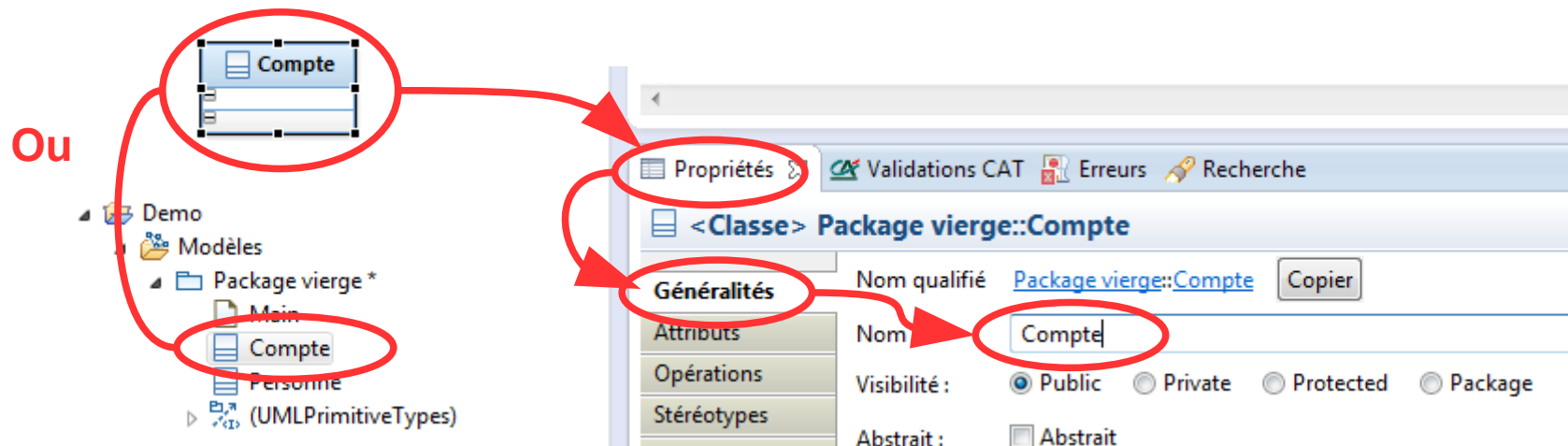


• Dans un diagramme

- Sélectionner la classe (clic dans son bandeau de titre bleu)
 - Les poignées (petits carrés noirs) apparaissent
 - Attention : si on clique sur le titre, on sélectionne le titre, ce qui n'est pas pareil



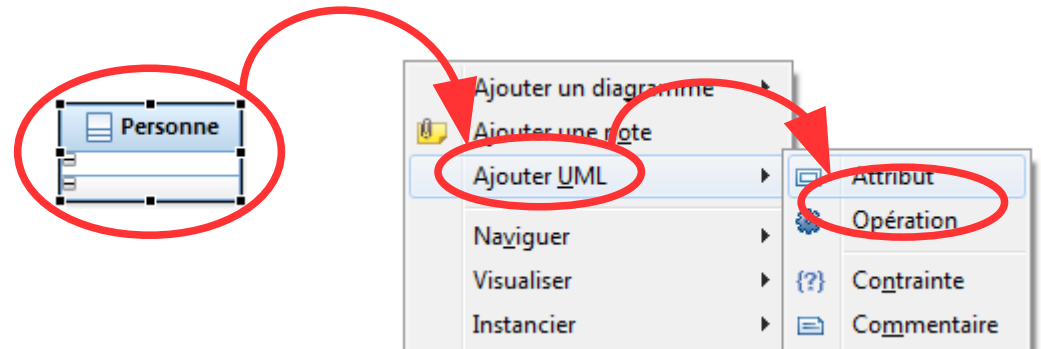
• Dans la vue Propriétés (en dessous de l'éditeur)



- **Pas de règles particulières sur le nommage pour UML**
 - Chacun est libre de nommer les choses comme il le souhaite
- **Le nom est généralement un identifiant externe (fonctionnel)**
 - Attention à la casse : **client** <> **Client**
 - Attention au conflit si deux éléments de même type ont le même nom
 - RSA émet un avertissement mais laisse faire car il utilise des identifiants techniques internes
- **En général, on retrouve une certaine logique dans le nommage**
 - Tout ce qui est général ou partagé est **capitalisé**
 - En particulier, les classes : **Client**, **Commande**...
 - Tout ce qui est local est en minuscules
 - En particulier, les propriétés et les opérations : **nom**, **adresse**, **calculerLaSolvabilité()** ...
 - L'analyse laisse généralement des espaces dans les noms composés
 - La conception utilise souvent le camelCase (concaténation des mots)
 - Élément de structure => ElementDeStructure

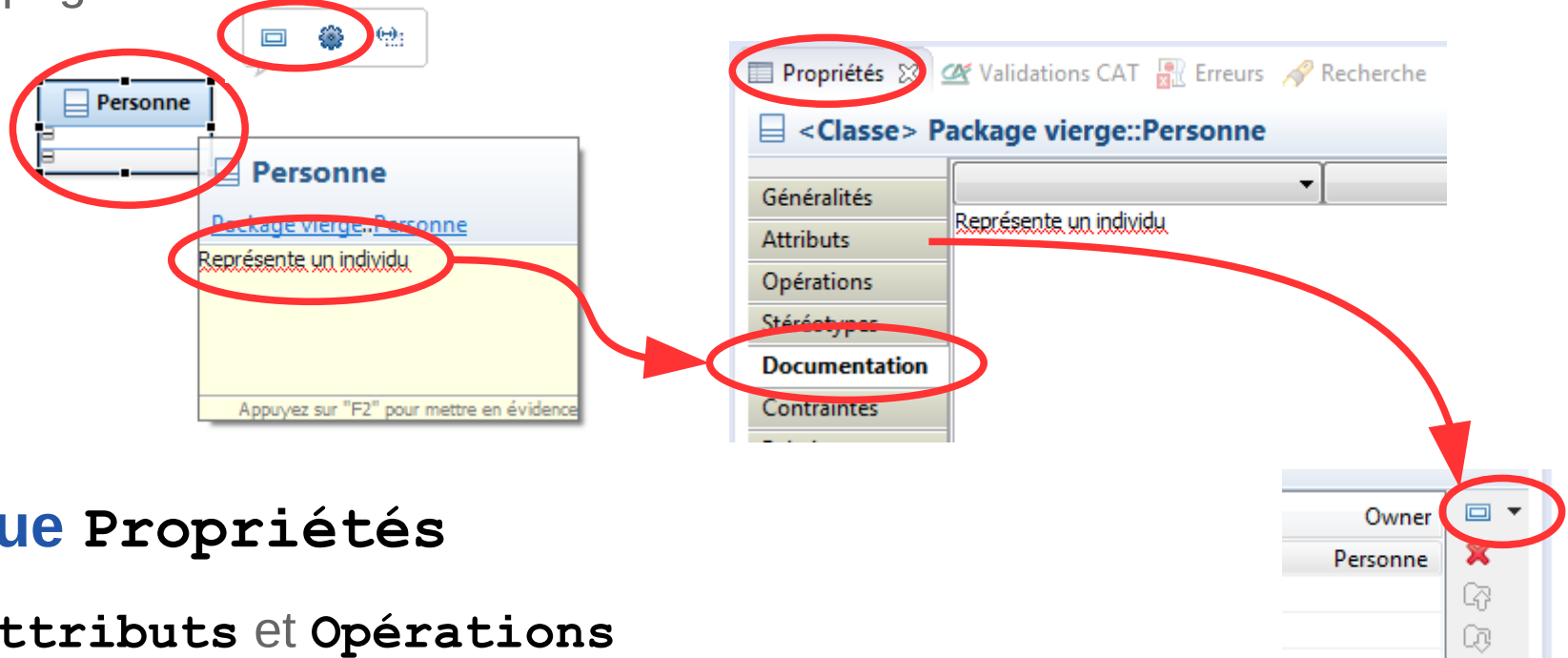
• Clic-droit sur la classe

- Dans le diagramme
- Ou dans l'arborescence



• Ou avec le menu fugitif, au survol de la classe dans le diagramme

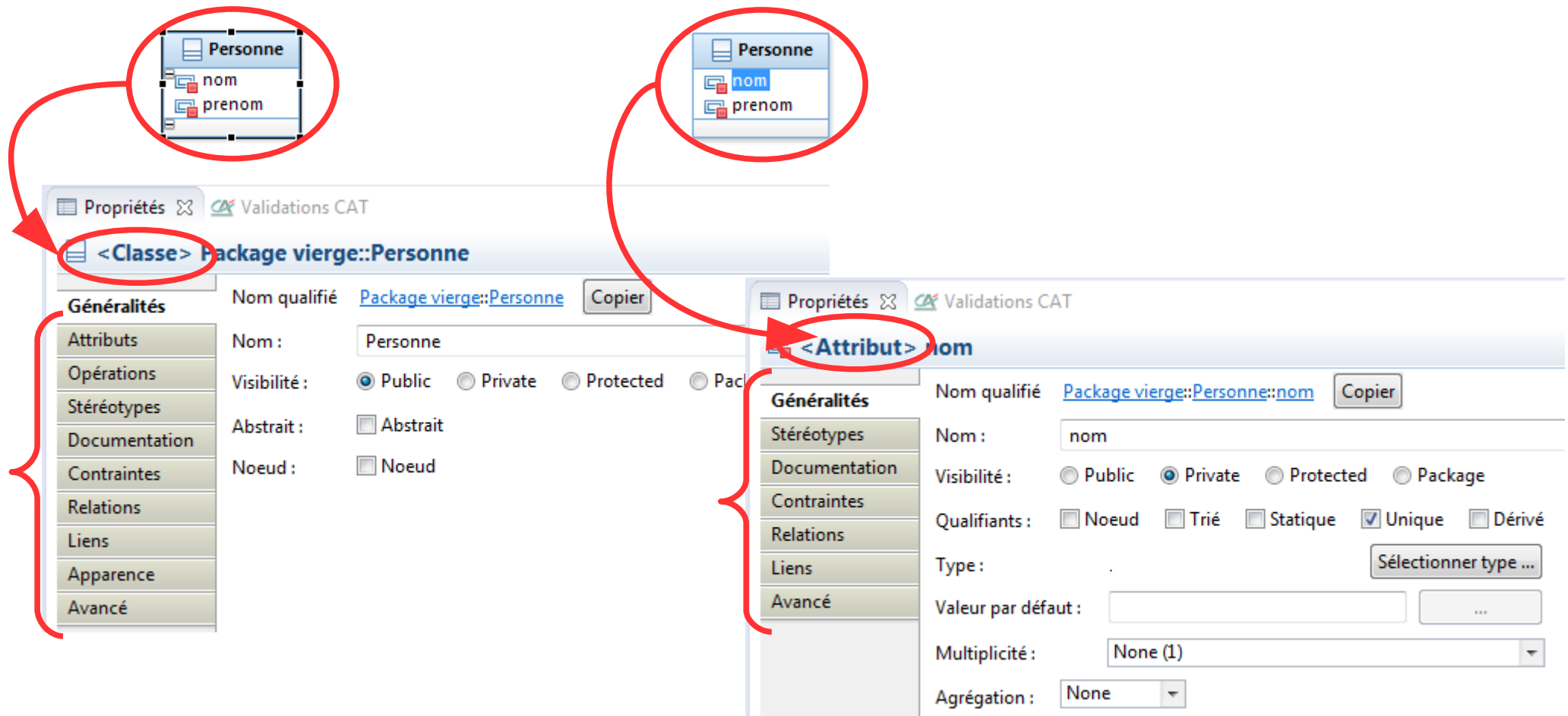
- Il s'accompagne d'une boîte d'information affichant la documentation



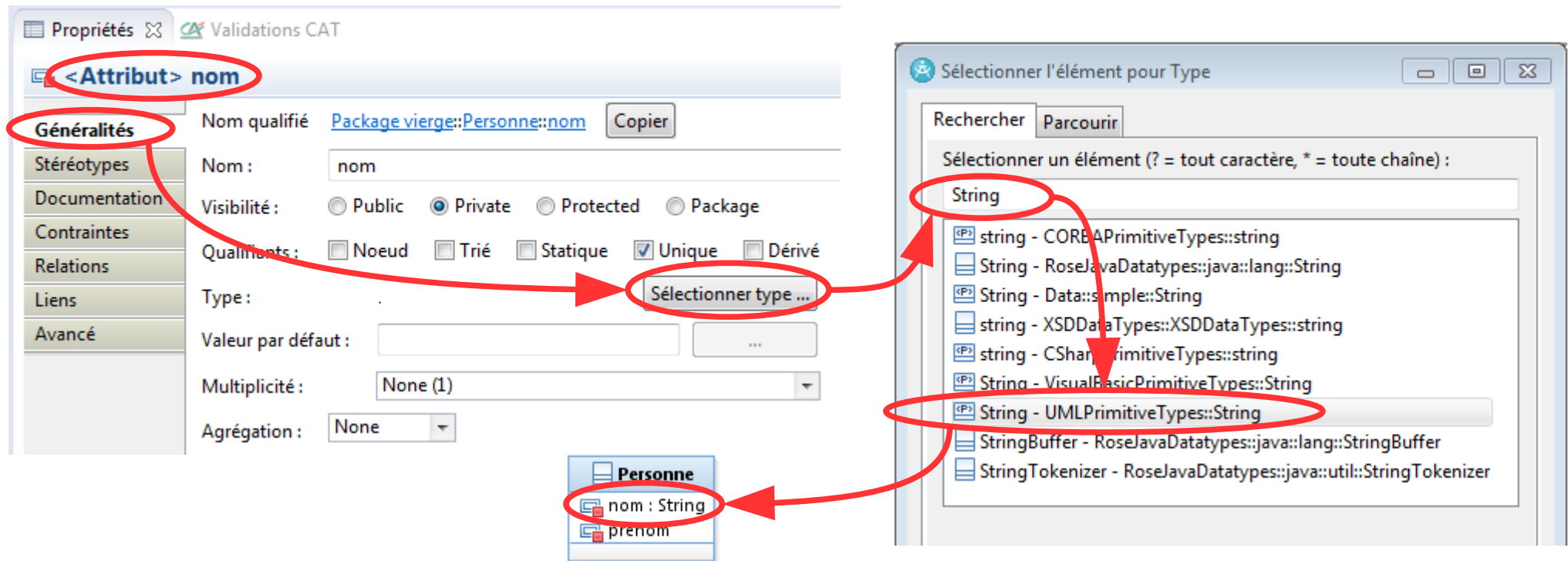
• Ou via la vue Propriétés

- Onglets **Attributs** et **Opérations**

- Cette vue permet de voir les détails de l'élément sélectionné
 - Plusieurs onglets, selon la nature de l'élément sélectionné
 - Permet de renommer, d'ajouter la documentation, d'indiquer des détails



- Avec le bouton **Sélectionner type...**
 - Un type est un objet (ou plutôt une classe)
 - Certains sont prédéfinis mais on peut aussi utiliser ses propres classes
 - Le choix dépend de ce que l'on modélise (analyse, Java, MOMP, XSD, ...)
 - Par exemple `UMLPrimitiveTypes::String` en analyse
 - Et `XSDDataTypes::XSDDataTypes::string` en conception XML



- **Phases traditionnelles après le recueil des exigences**

- L'analyse permet de réfléchir et de s'assurer de la compréhension du problème
- La conception explore dans les détails et expose des solutions au problème

- **Une question de détails**

- Il est tentant d'ajouter les détails dans les diagrammes d'analyse
- Ce n'est pas une bonne idée car on perd la vue « stratosphérique »
- Présenter des diagrammes très détaillés amène de la complexité
- Ces considérations relèvent de la méthodologie

- **UML se concentre sur la notation, pas sur la méthodologie**

- On doit pouvoir descendre dans les détails si on le souhaite
- UML offre une notation complète, on s'organise comme on veut
- Ou plutôt comme le dicte la méthodologie qu'on applique
- Problème : il faut connaître les concepts communs de la POO



Les fondements objets

• Idées de la POO (Programmation Orientée Objet)

- Se rapprocher des problématiques de simulation du monde réel
- Voir les choses autrement (remise en cause des habitudes)
- Favoriser la réutilisation, la maintenabilité, la robustesse, ...
- S'éloigner des couches basses proches du matériel
- Représenter les concepts sous forme d'abstractions, de modèles

CorpsMobile

- position
- vitesse
- accélération
- accélérer()
- calculerPosition()

• Un premier concept fondamental : l'encapsulation

- Un objet = une capsule autonome enfermant **données** et **traitements**
 - La structure interne et ses données sont cachées de l'extérieur
 - L'objet est responsable de la cohérence interne de ses données
 - On ne pourra plus modifier une variable depuis n'importe où
 - On ne pourra plus oublier de vérifier une contrainte ou une règle métier
- On applique simplement ce qu'il se passe dans la vie courante
 - Des objets bien hermétiques, avec des scellés qui garantissent leur intégrité

Compte

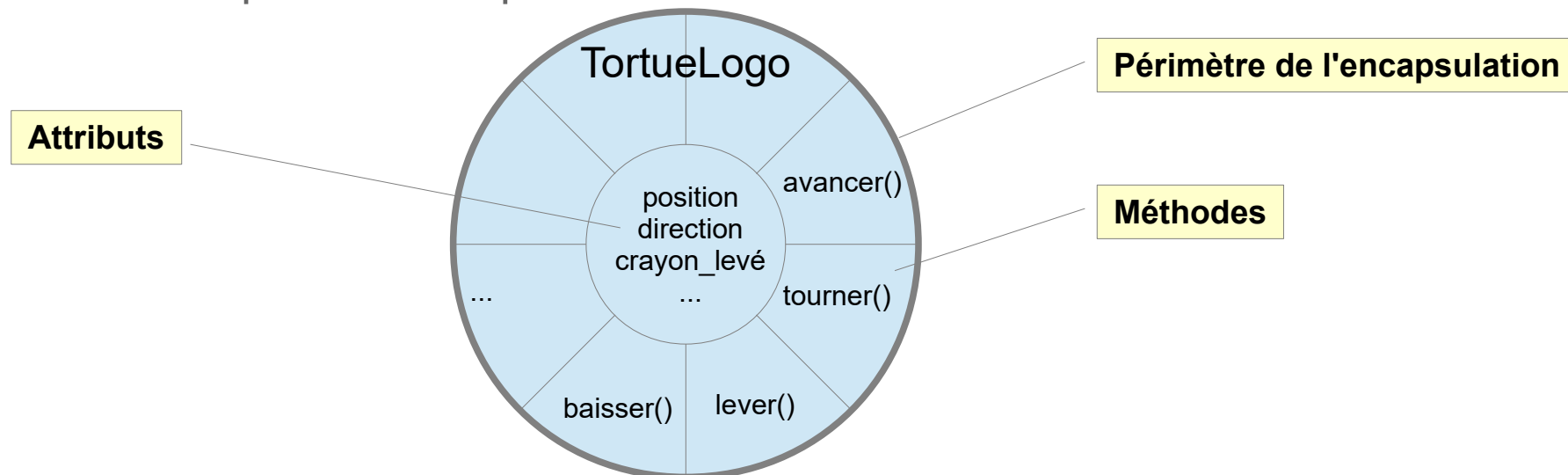
- numéro
- solde
- débiter()
- créditer()
- consulterSolde()

- **L'objet encapsule ses données**

- Dans des **attributs** ou **propriétés**
- Décrit les caractéristiques comme on décrirait un animal d'une espèce inconnue
 - « Il a une carapace dure, possède six pattes, mesure 3,2 cm de long, ... »
- Fondamentalement des variables mais propres à l'objet

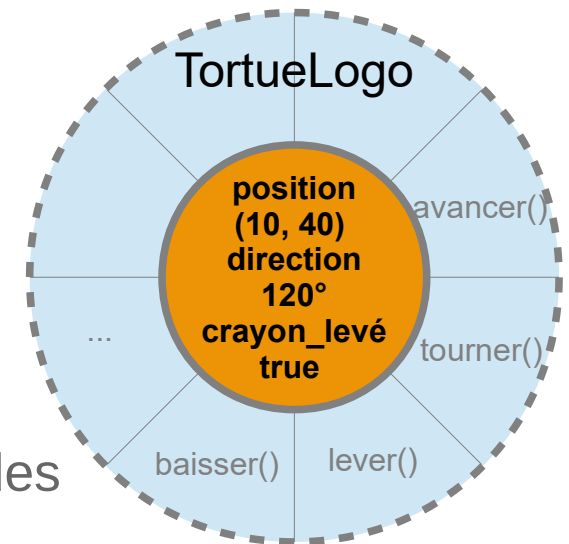
- **L'objet encapsule ses traitements dans des méthodes**

- Le seul code autorisé à accéder à ses attributs et éventuellement à les modifier
- Correspond techniquement à des sortes de fonctions mais internes à l'objet



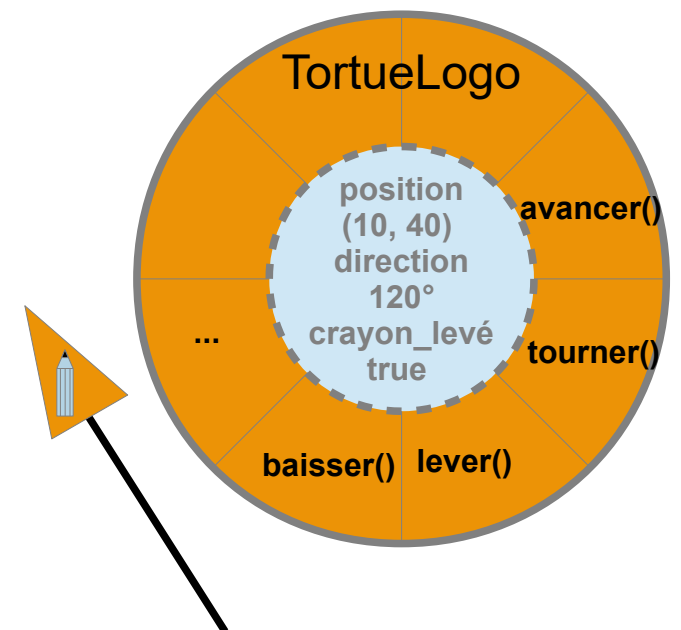
• Les attributs représentent l'état de l'objet

- Contiennent les valeurs qui modélisent un état global
- Dans le cas d'un déplacement de la tortue logo
 - Laissera une trace selon la valeur de **crayon_levé**
 - Avancera dans sa direction actuelle
- L'état d'un objet ne peut être modifié que par ses méthodes

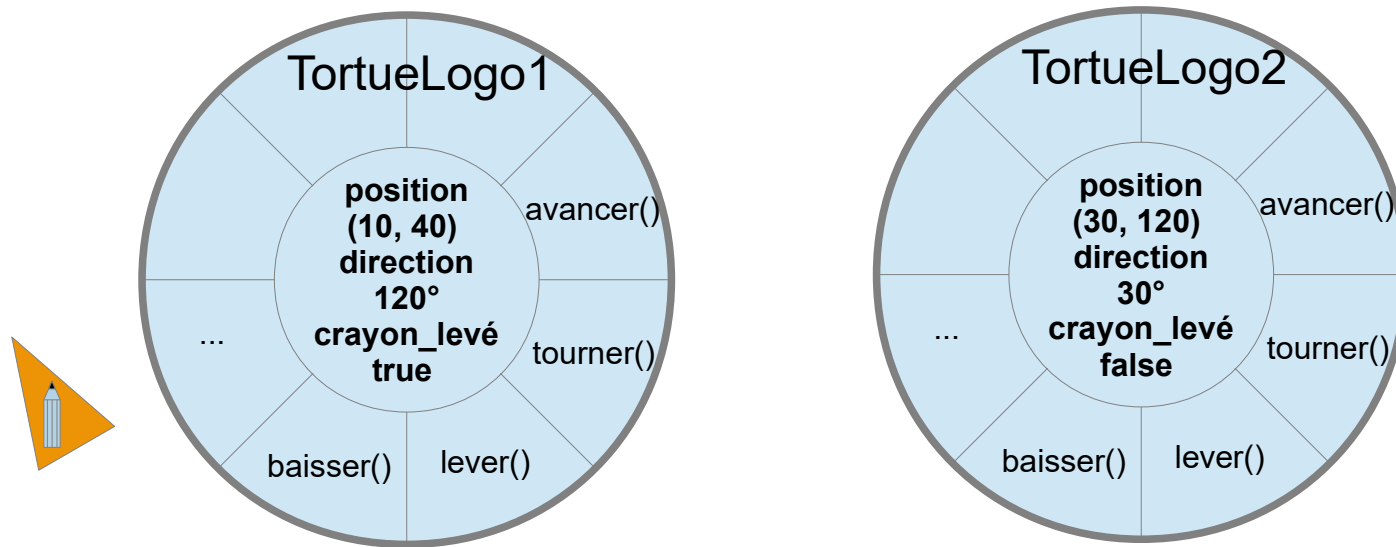


• Les méthodes représentent son comportement

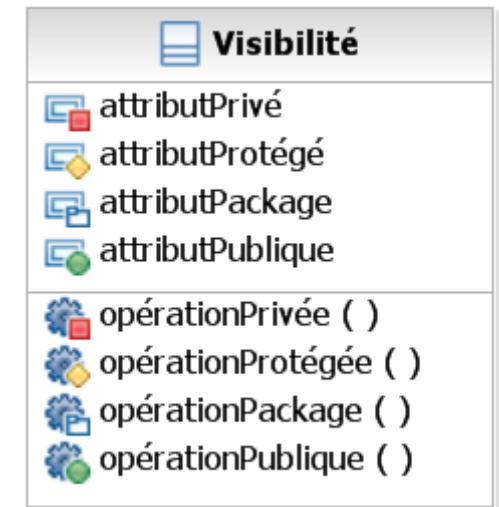
- Ce qu'il sait faire, les traitements qu'il peut réaliser
- C'est son **mode d'emploi**
- Une méthode peut :
 - Changer l'état de l'objet
 - Retourner une information
 - Produire un effet de bord (laisser une trace sur le papier, ...)



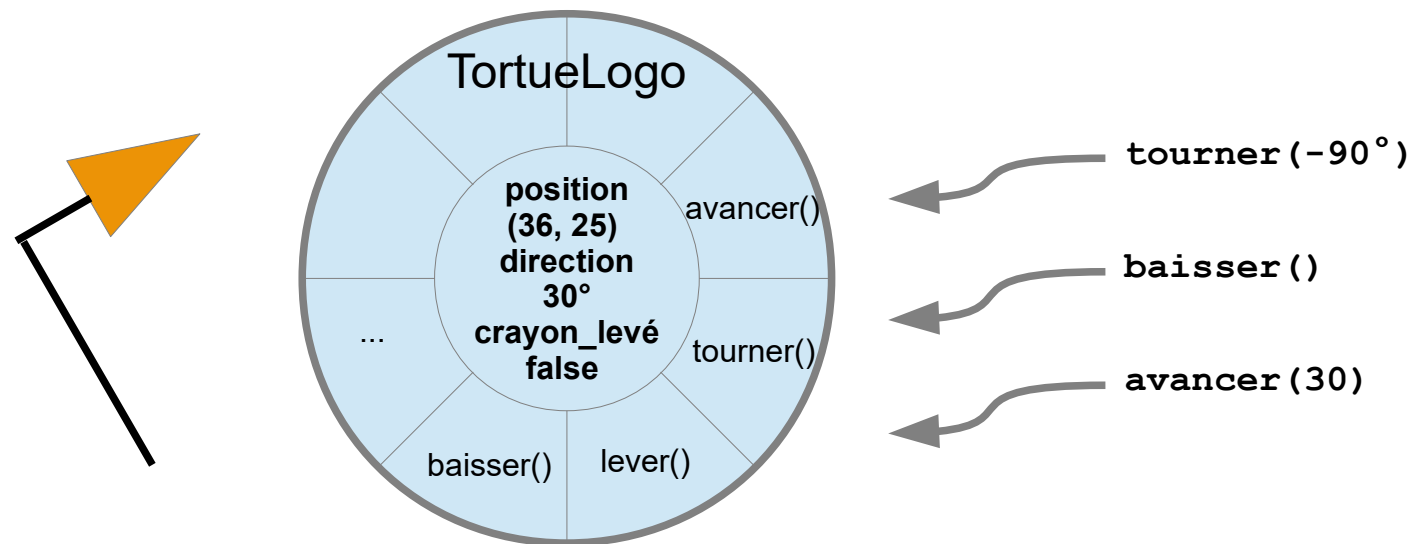
- **Chaque objet dispose de sa propre identité intrinsèque**
 - Correspond techniquement à son adresse mémoire
 - Il dispose de ses propres valeurs d'attributs (son état)
 - Il correspond à un individu unique dans le système



- **L'encapsulation est plus ou moins renforcée selon les langages**
 - Dans les langages dérivés du C, on distingue 4 niveaux de visibilité
- **Ces niveaux sont repris dans la notation UML**
 - Applicables aux attributs, aux opérations et à tout ce qui peut être contenu
 - RSA les représente par des icônes
- **Visibilités**
 - **Privé** « - » : accessible uniquement l'objet
 - **Public** « + » : exposé publiquement
 - **Package** « ~ » ou rien : exposé aux objets du package
 - **Protected** « # » : exposé au package et sous classes
 - Lié à la notion d'héritage que nous verrons plus loin



- **Les méthodes étant encapsulées, comment les activer ?**
 - L'appel de fonction ou procédure habituel ne permet pas d'y accéder
 - L'invocation est plus subtile : elle correspond à une invocation « à distance »
- **Les objets communiquent par envoi de « message »**
 - On envoie un message à un objet (notion de **destinataire** ou de **receveur**)
 - Celui-ci répond en activant la **méthode** qui correspond au message
 - Le message peut comporter des **arguments** (des valeurs de **paramètres**)



- Cette approche se retrouve dans les langages orientés objets
 - Smalltalk, C++, Java, JavaScript, ...

```
Transcript show: 'Hello world!'.
```

Smalltalk
(envoi du message show:
à l'objet Transcript)

```
Console::WriteLine(L"Hello World");
```

C++/CLI
(envoi du message
WriteLine() à l'objet
Console)

```
System.out.println("Hello world!");
```

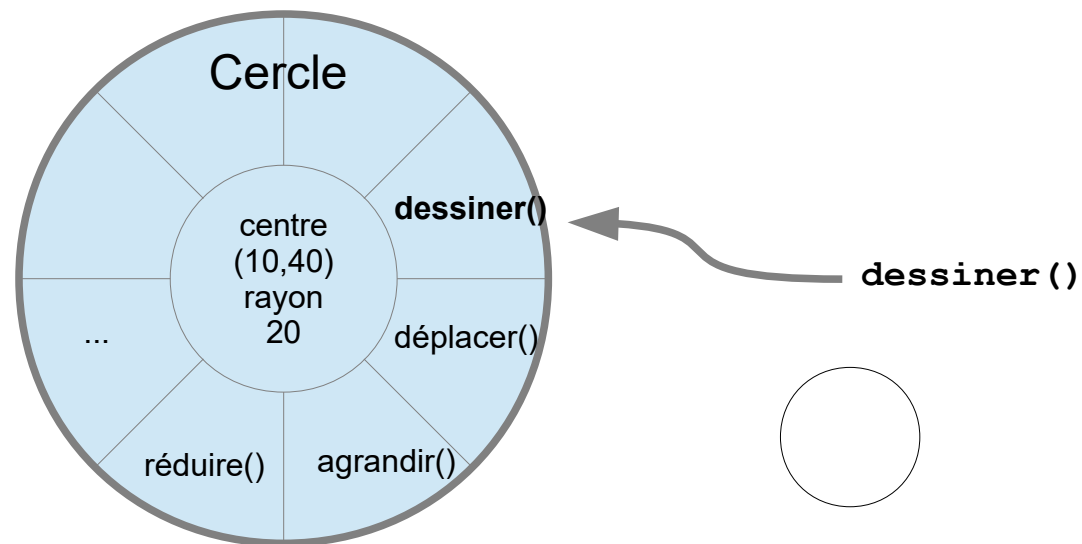
Java
(envoi du message println()
à l'objet out attribut
de l'objet System)

```
window.alert("Hello world!");
```

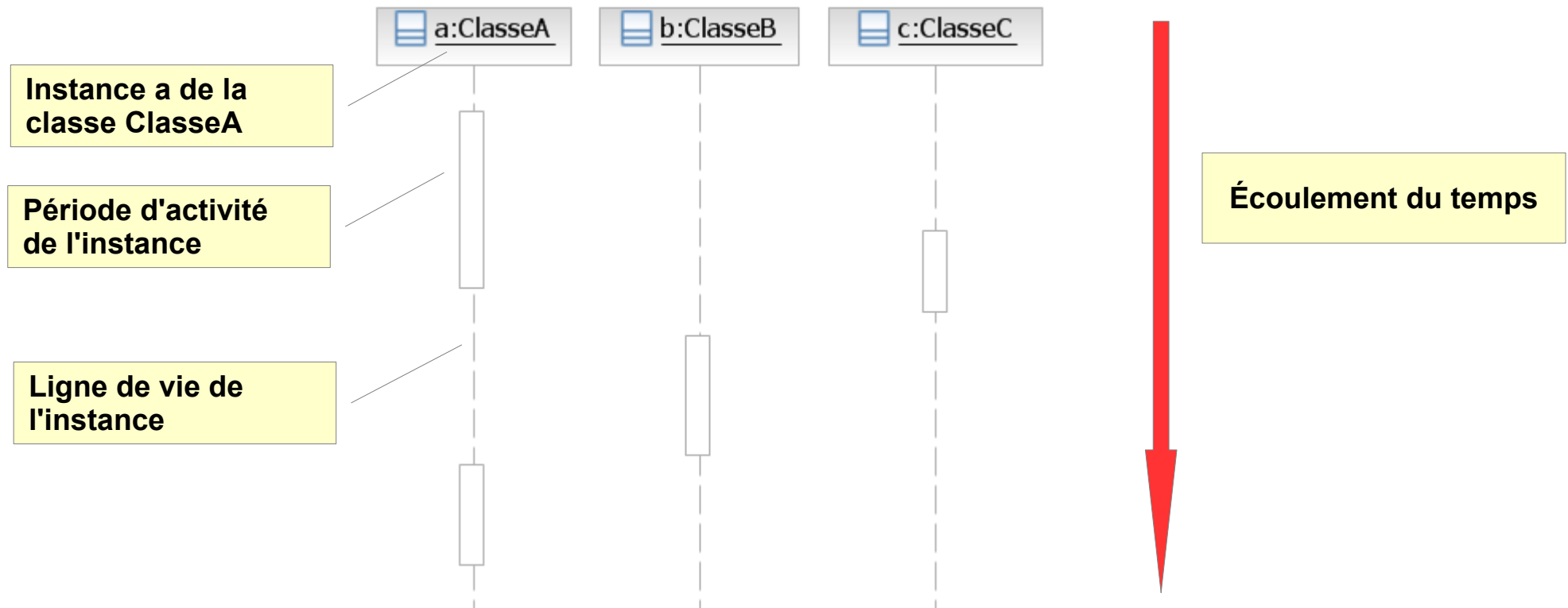
JavaScript
(envoi du message alert()
à l'objet window)

- **La recherche de méthode**

- Lorsqu'un objet reçoit un **message**, il recherche la **méthode** à activer
- Il recherche, parmi ses méthodes, celles qui correspondent au message
 - Cette correspondance « message – méthode » dépend du langage
 - Correspondance au moins sur le nom
 - Les langages typés prennent généralement en compte le nombre et le type des paramètres
 - Et autorisent ainsi la notion de « **surcharge** »

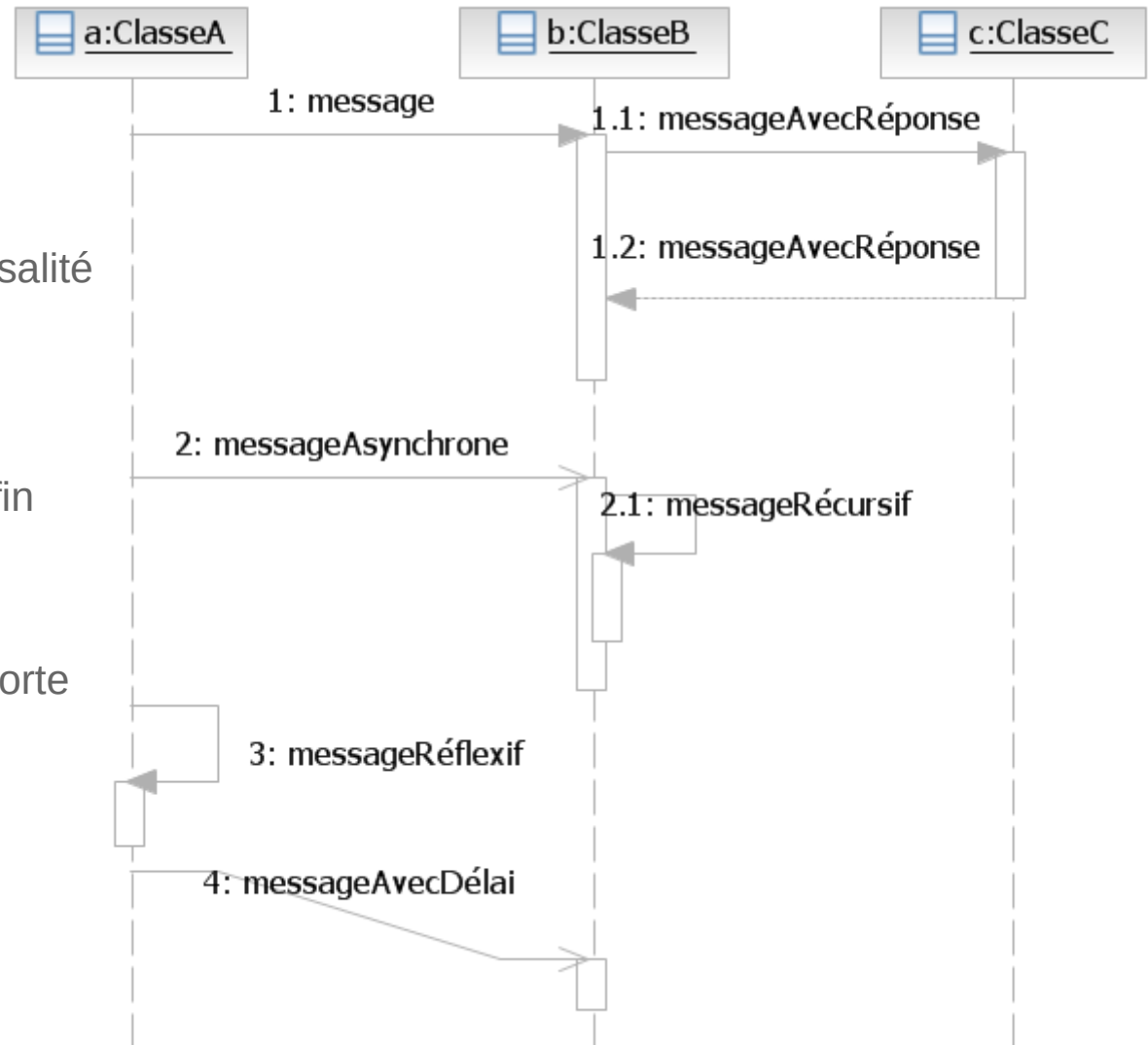


- Le diagramme de séquence **montre une séquence de messages**
 - Il fait partie des **diagrammes dynamiques**
 - On montre une séquence de messages échangés entre des objets (notion de **collaboration**)
 - Il fait aussi partie des **diagrammes fonctionnels**
 - Par extension, on considère les utilisateurs et le système comme des objets
 - Le diagramme de séquence montre alors les échanges entre l'utilisateur et le système



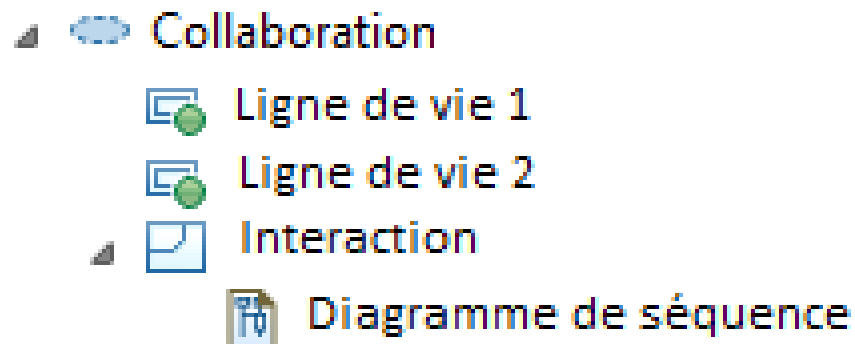
- **Les messages sont représentés par des flèches**

- De l'émetteur vers le récepteur
- Chronologiquement
 - RSA les numérote
 - L'activité (rectangle) montre la causalité
- Message **synchrone**
 - Flèche pleine
 - L'émetteur est bloqué et attend la fin
 - RSA explicite le retour par un message réponse (en pointillés)
 - On peut effacer ce retour s'il n'apporte pas d'information
- Message **asynchrone**
 - Flèche ouverte
 - L'émetteur n'est pas bloqué



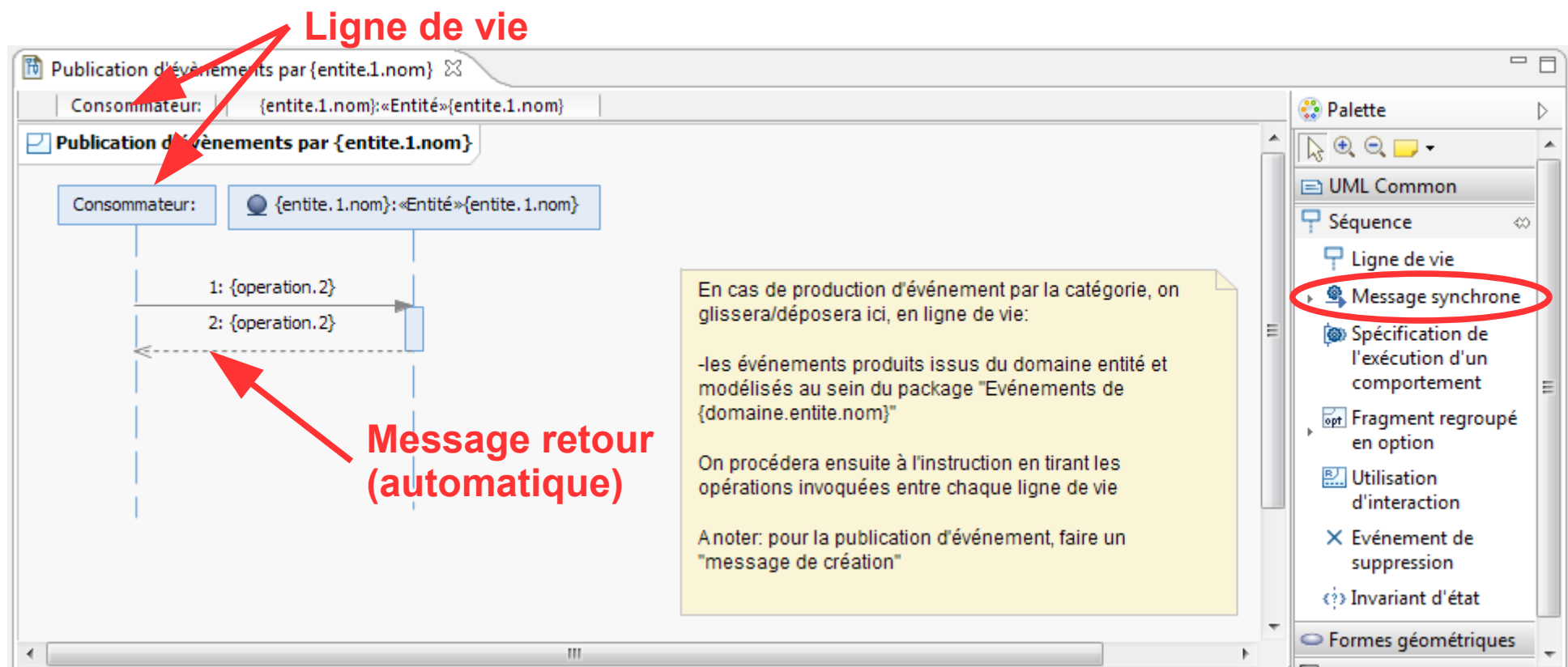
- **Ergonomie complexe dans RSA**

- Le diagramme est caché dans le modèle sous 3 niveaux d'imbrications
 - D'abord une **collaboration** (sorte d'objet virtuel contenant les objets qui collaborent)
 - La collaboration contient les attributs représentant les lignes de vie
 - Ensuite, une ou plusieurs **interaction** (objet contenant une liste de messages)
 - Enfin le **diagramme de séquence** (présente graphiquement l'interaction)
- Construction des messages malaisées (nécessite de l'entraînement)



• Pour créer un message

- Sélectionner le type de message dans la palette
- Puis cliquer sur la ligne de vie de départ et relâcher sur la ligne de vie d'arrivée
- Choisir une opération cible ou choisir d'en créer une nouvelle
 - On peut changer plus tard en double cliquant sur le libellé du message

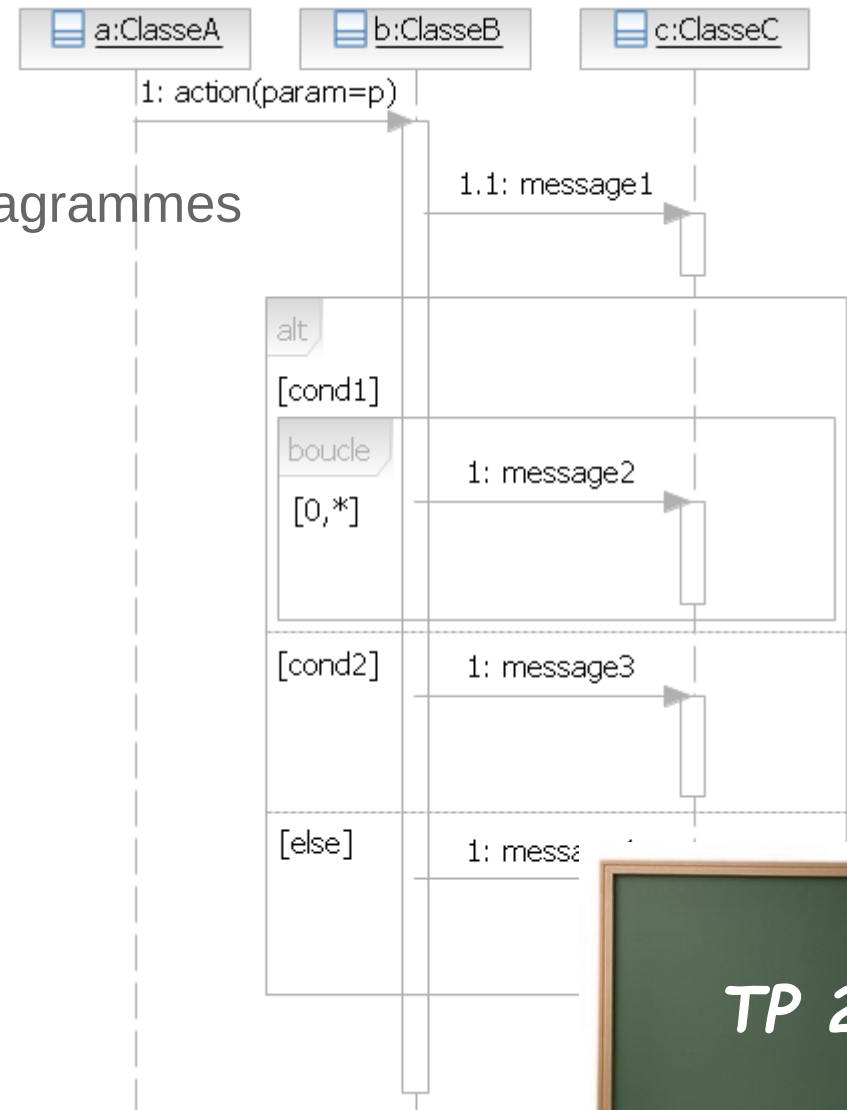


- **Les fragments de séquence : exprimer des structures de contrôle**

- Pour regrouper des sous-séquences
- Peuvent être imbriqués
- Alourdit le schéma, préférer plusieurs diagrammes

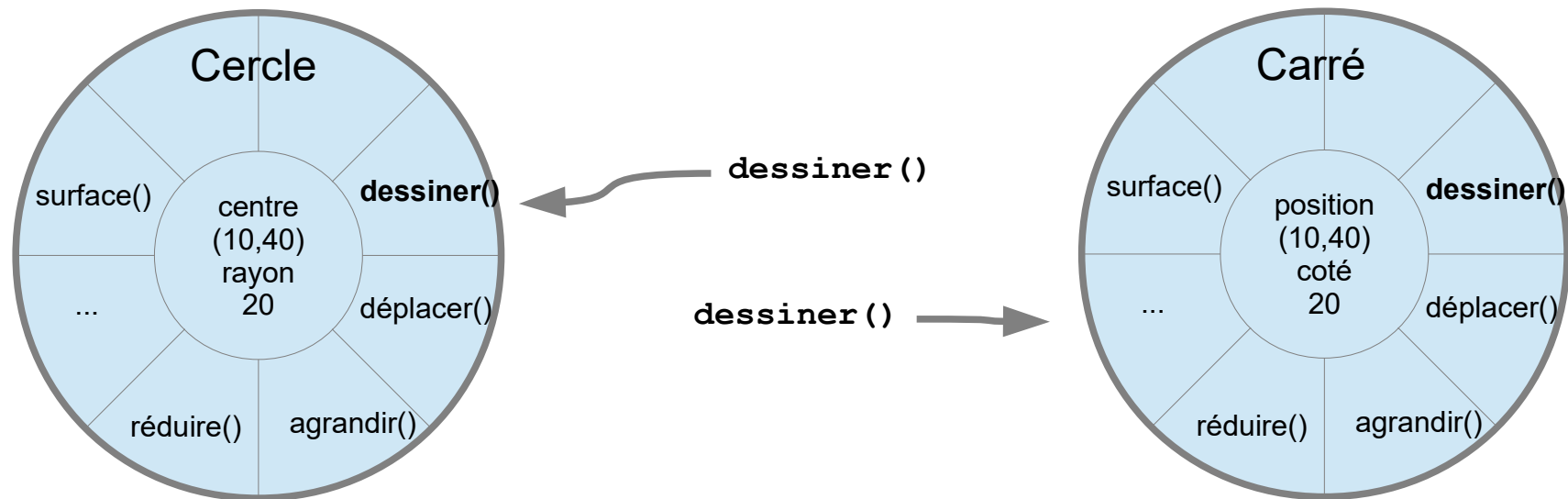
- **Nombreux choix possibles**

- « opt » : sous-séquence optionnelle
- « alt » : choix selon conditions
- « loop » : boucle
- « strict » : ordre strict des messages
- « seq » : ordre non strict
- « par » : messages en parallèle
- ...



TP 2

- **Une conséquence des messages et de la recherche de méthode**
 - Deux objets différents peuvent répondre au même message
 - Éventuellement de manières différentes
- **C'est le message qui est « polymorphe » (« plusieurs formes »)**
 - Le traitement qui sera exécuté dépend de l'objet qui reçoit le message
- **Une deuxième conséquence : principe de substitution**
 - On peut remplacer un objet par un autre qui répond aux mêmes messages



- **Cette notion de polymorphisme est extrêmement puissante**
 - Et souvent mal comprise
 - Elle a de profonds impacts sur la manière de penser et de nommer les choses
 - Le premier impact est sur le nom même de « méthode » plutôt que « fonction »
- **Fonction et procédure : un nom unique dans tout le système**
 - Le nom de fonction identifie une séquence d'instructions de manière unique
 - Le compilateur garde une correspondance nom - adresse mémoire
 - Procédure dessiner_carré(unCarré) → #45AF
 - Procédure dessiner_cercle(unCercle) → #FF67
 - Une invocation de fonction est transformée en un appel direct à l'adresse
 - La fameuse « édition de liens » des compilateurs (« static binding »)
 - dessiner_carré(monCarré) → call #45AF

- **Message : un nom réutilisable par plusieurs objets**
 - Pour un même nom, plusieurs méthodes dans le système
 - Ce qui contribue à la diminution du vocabulaire du système
 - Une invocation (envoi d'un message) provoque une recherche dynamique
 - Effectuée par l'objet qui reçoit le message (« dynamic binding »)
- **Permet l'écriture de code plus générique, maintenable et évolutif**

Modification si ajout de Triangle

```
Procédure calculer_surface(tab_figures)
  total = 0
  pour chaque figure de tab_figures faire
    si figure.type = "Carré" alors
      total = total + figure.coté2
    fin
    si figure.type = "Cercle" alors
      total = total + (Pi*figure.rayon2)
    fin
  fin
  retourner total
fin
```

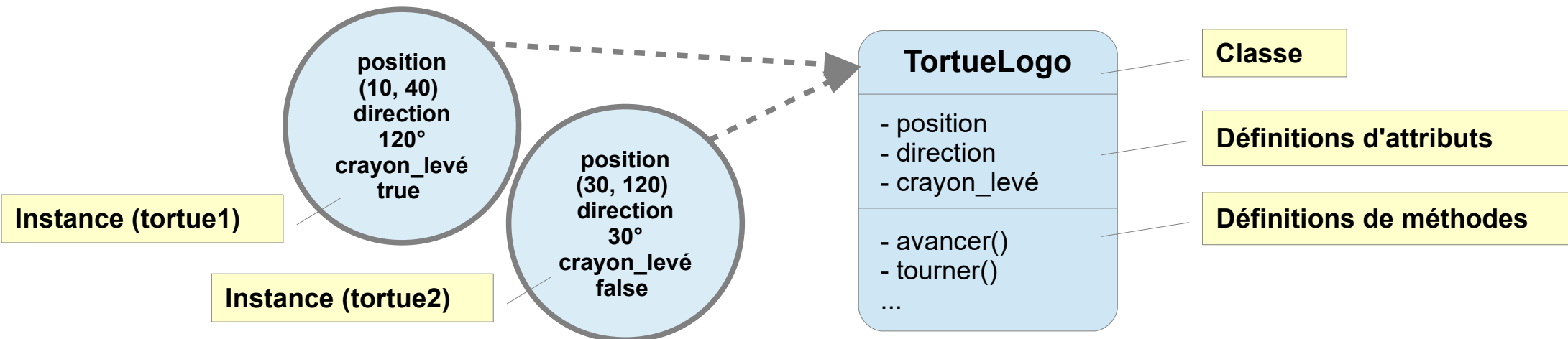
Code générique Pas de modification si ajout de Triangle

```
Procédure calculer_surface(tab_figures)
  total = 0
  pour chaque figure de tab_figures faire
    total = total + figure.surface()
  fin
  retourner total
fin
```

```
Carré
  Méthode surface()
    retourner coté2
  fin
fin
```

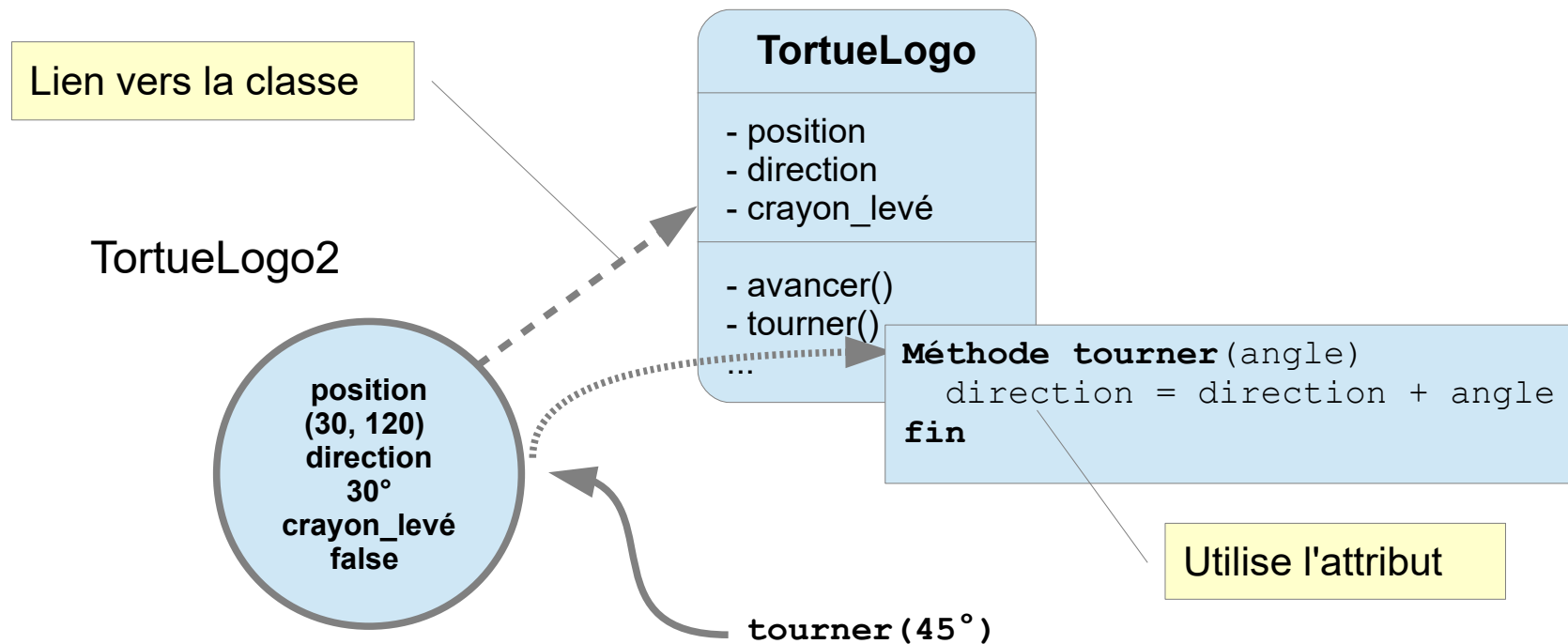
```
Cercle
  Méthode surface()
    retourner PI*rayon2
  fin
fin
```

- **De nombreux objets vont disposer des mêmes caractéristiques**
 - Nos deux tortues logo disposent exactement des mêmes attributs et méthodes
 - Il serait intéressant qu'elles partagent une définition commune
- **La classe comme définition de la structure générale d'un objet**
 - Une classe pour définir une liste d'attributs et de méthodes
 - Les objets (instances) créés à partir de cette classe auront les mêmes attributs
 - Instance = exemplaire, occurrence
 - Ils gardent un lien vers la classe, qui contient les méthodes
 - Dans les langages typés, la classe constitue un type (pour typer les variables)



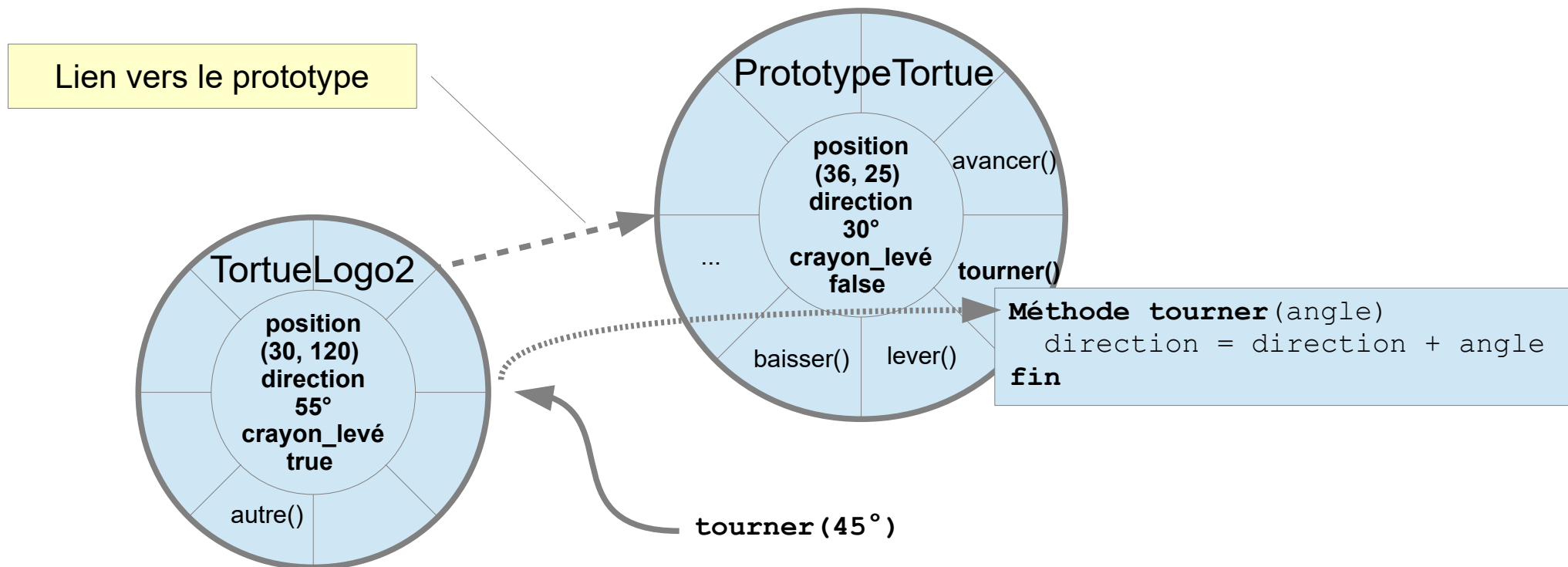
• La recherche de méthode est légèrement modifiée

- Lorsqu'un objet reçoit un message, il recherche dans les méthodes de sa classe
- Puis applique la méthode dans son contexte (avec ses attributs)

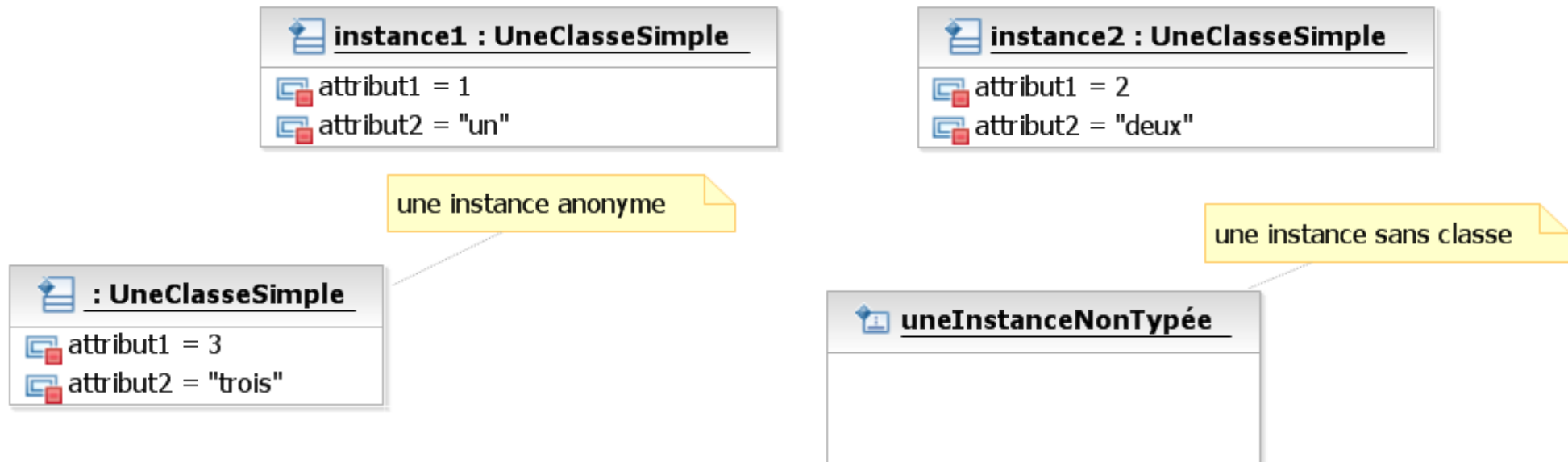


• Cette notion de classe n'est pas réellement fondamentale

- Les langages de « prototype » gèrent cela différemment (ex : JavaScript)
- Dans ce type de langage, les instances sont construites à partir d'un prototype
 - Et gardent un lien vers ce prototype
- Lorsque l'objet reçoit un message, il recherche dans ses méthodes
- S'il ne trouve pas de méthode correspondante, il recherche dans son prototype



- **La plupart des langages objet sont des langages de classe**
 - UML vise le consensus entre les langages objets communs
 - Il vise une notation pour le génie logiciel où les classes sont plus adaptées
- **UML offre aussi la notation pour faire les diagrammes d'objets**
 - Permet de représenter des **snapshot** d'instance (photo de l'état d'une instance)
 - On explicite ainsi un exemple (ou un contre-exemple)
 - Cette notation est peu utilisée dans les outils



- **Un attribut (ou une méthode) est parfois général**
 - Non spécifique à une instance particulière
 - Soit une valeur commune à toutes les instances
 - Constante ou variable partagée (ex : AgeMajorité, EpaisseurParDéfaut, Pi, ...)
 - Soit un traitement qui ne nécessite pas de contexte d'instance spécifique
 - Souvent des traitements utilitaires (ex : estBissextile(uneAnnée), convertirEnRadians(unAngle))
- **On les qualifie d'attribut et de méthode de classe**
 - Souvent appelées « **statique** » par les langages héritiers du C
 - Attention : des variations subtiles selon les langages
- **On les invoque en envoyant un message à la classe**
 - Nom de la classe comme destinataire du message
- **Recommandation : ne pas en abuser**
 - Ou bien on risque de basculer naturellement vers une logique procédurale

- **Attribut dérivé**

- Lorsqu'un attribut peut être calculé simplement à partir d'autres attributs
 - Ex : « âge » calculé directement par « date du jour - date de naissance »

- **Attribut multiple**

- Lorsqu'un attribut contient plusieurs valeurs (typiquement un tableau)

- **Attribut optionnel**

- Lorsque la valeur d'un attribut est optionnelle
- Concept similaire au « nullable » des bases de données relationnelles

- **Type d'un attribut ou de retour de méthode**

- Dans les langages typés, toute variable est déclarée avec un type
- Dans les langages objets, toute classe constitue un type

- **Valeur initiale d'un attribut**

- Constitue une valeur par défaut

- **Attributs et opérations de classe (statiques)**

- Notés en les soulignant

- **Attributs dérivés**

- Préfixés par « / »

- **Paramètres d'opération**

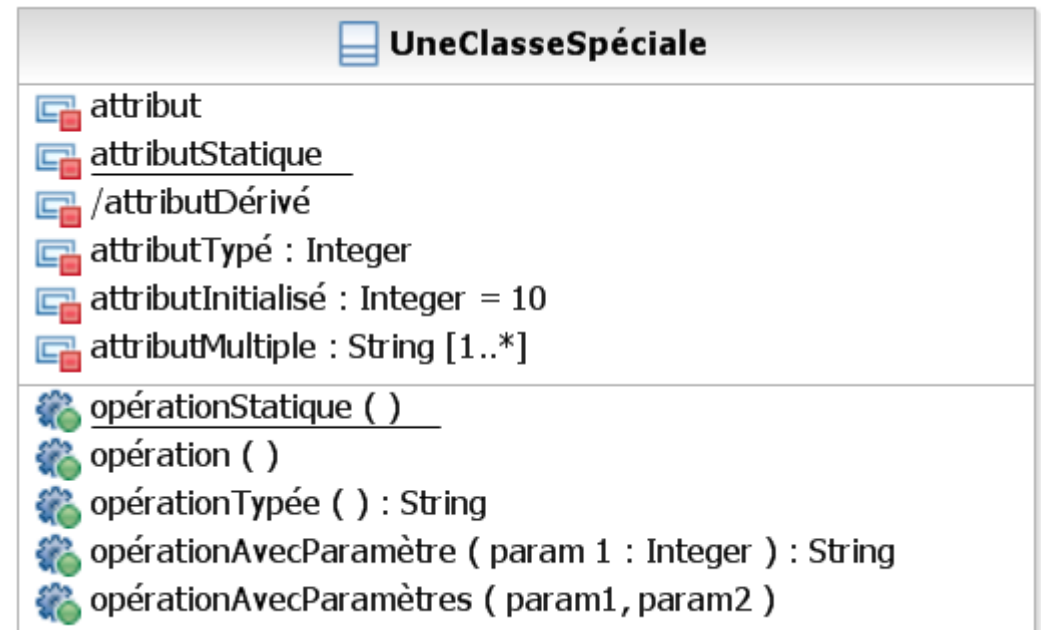
- Entre les parenthèses
 - Séparés par une virgule

- **Type**

- Pour les attributs, paramètres et retour d'opération
 - En faisant suivre le nom par « : » suivi d'un type

- **Multiplicité**

- Entre crochets, parmi [0..1], [*], [1..*]



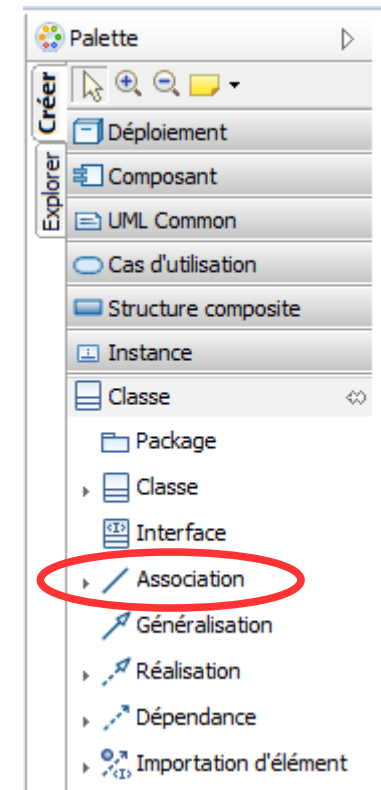
TP 3

Les relations

- **Un objet tout seul ne fait pas grand chose**
 - Il doit souvent collaborer avec d'autres objets
- **Un objet complexe est souvent composé d'autres objets**
 - C'est une vision proche de nos constructions matérielles habituelles
 - Cette organisation est la base de la réutilisation en objet
- **Nécessité d'une référence vers l'objet cible**
 - Pour pouvoir lui envoyer des messages
 - Une référence = une variable qui référence cet objet cible
 - En général un attribut
 - C'est aussi le principe de base du « GarbageCollector »
 - Lorsqu'il n'y a plus aucune référence vers un objet, il ne peut plus être utilisé et peut être détruit
- **La conception objet prévoit plusieurs types de liens entre classes**

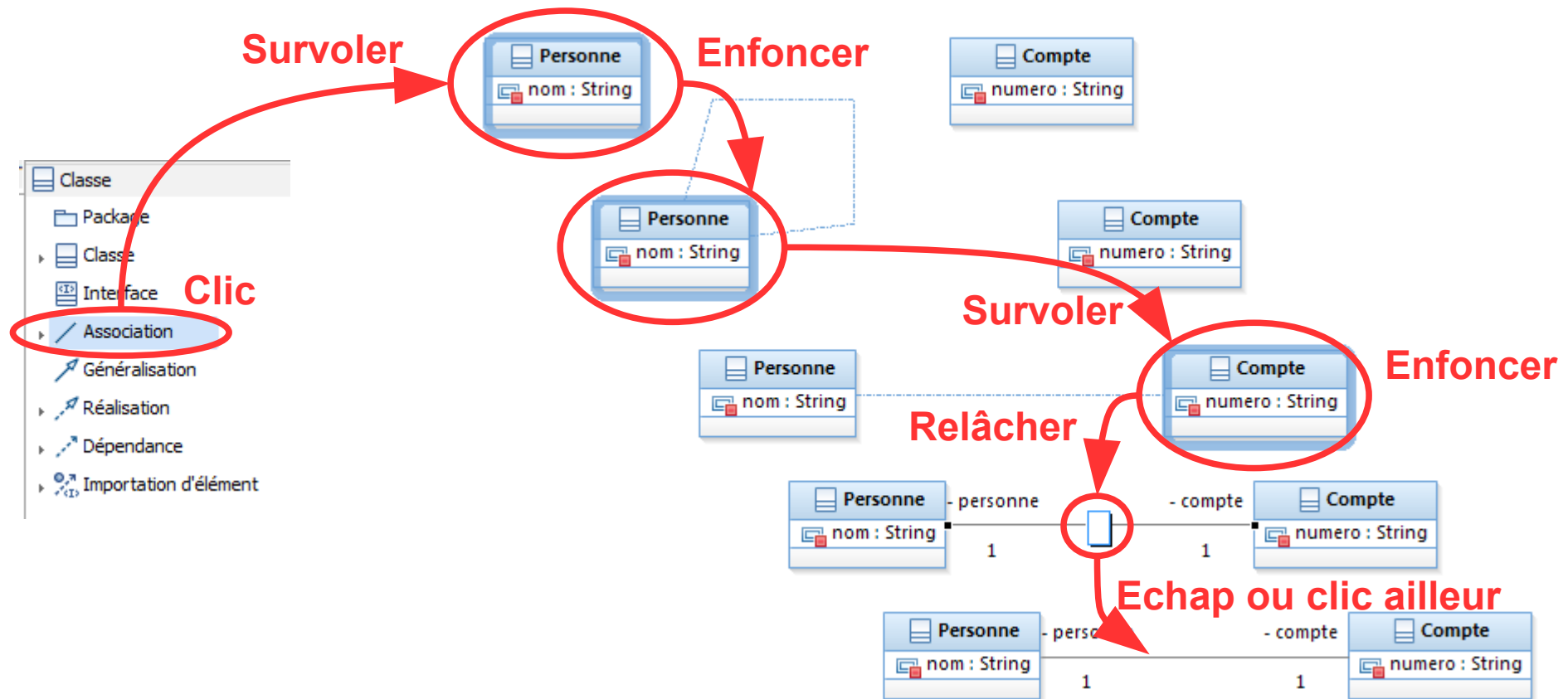
- **Les éléments UML sont liés entre eux par des relations**
 - Représentées par des traits, généralement entre deux éléments
- **UML propose de nombreux types de relations**
 - **Association** : lien de collaboration durable entre les éléments
 - **Généralisation** : lien de parenté entre les éléments (héritage)
 - **Réalisation** : lien entre une spécification et une implémentation
 - **Dépendance** : lien faible indiquant un impact entre les deux éléments
 - Généralement stéréotypé (marqueur entre chevrons) pour indiquer la nature de la dépendance
 - **Confinement** : indique l'inclusion d'un élément dans un autre
 - ...

- **Indiquent que les éléments collaborent de manière durable**
 - C'est-à-dire que l'on peut passer d'un élément à l'autre, qu'il y a une référence
- **Équivalent à la relation des schémas entités-relations classiques**
 - Ex : association entre **Personne** et **Compte**
 - Se lit : « les Personne sont associées à des Compte »
 - On peut indiquer le nom de la relation au dessus du trait
 - Ce n'est pas dans les habitudes (voir plus loin)

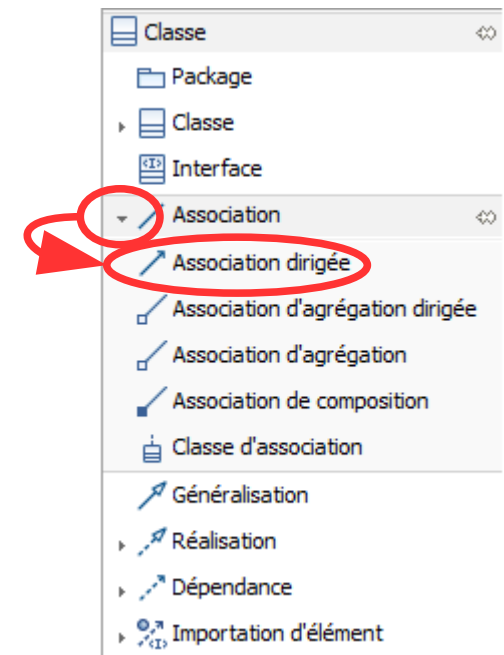


- **S'habituer à la logique ergonomique de RSA**

- On sélectionne l'outil dans la palette (clic)
- On clique sur la source, on tire et on relâche sur la cible
- **Remarque** : on désigne la source et cible (**inutile de s'arrêter au bord**)

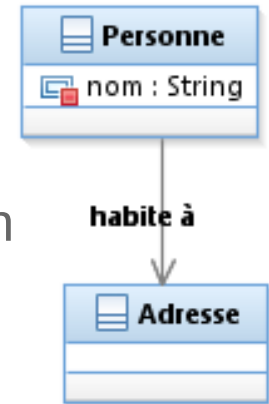
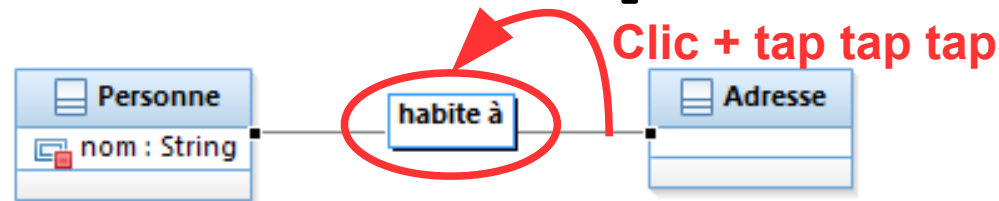


- **Les associations peuvent être orientées**
 - Sinon elles sont bidirectionnelles (le développeur devra coder en conséquence)
 - L'orientation traduit un sens privilégié de navigation
 - C'est une indication pour le développeur, il doit optimiser ce sens de navigation
- **Remarque : le modèle objet n'est pas un modèle relationnel**
- **Exemple : on ajoute la notion d'adresse d'une personne**
 - Le développeur doit optimiser l'accès à l'adresse
 - Il n'est pas tenu d'optimiser le chemin inverse



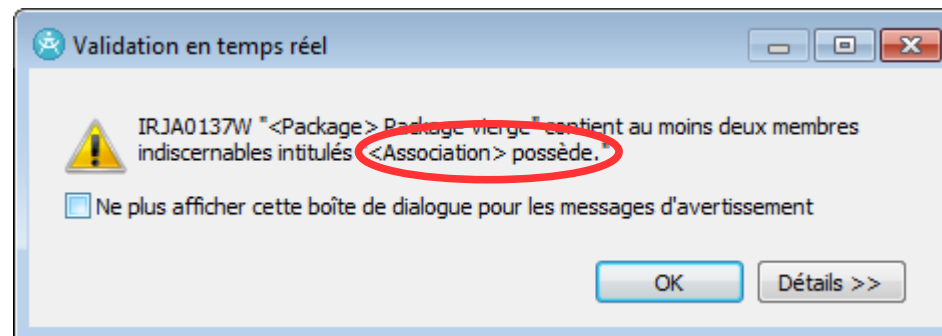
- **Les associations peuvent être nommées**

- En indiquant un libellé vers le milieu du lien
- Il peut être indiqué directement à la création de l'association
- On peut saisir directement au clavier en sélectionnant l'association
- On peut aussi le voir dans la vue **Propriétés**

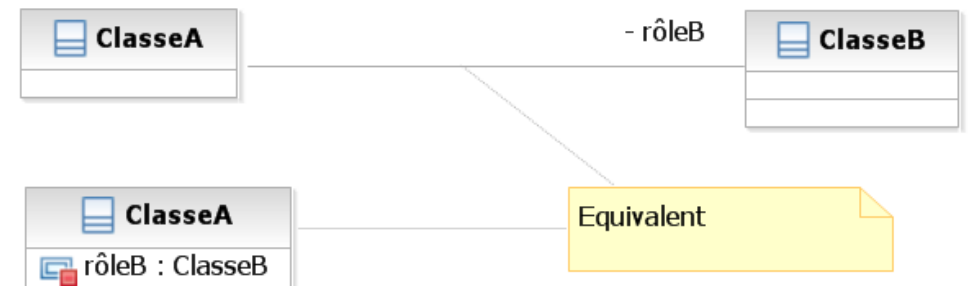
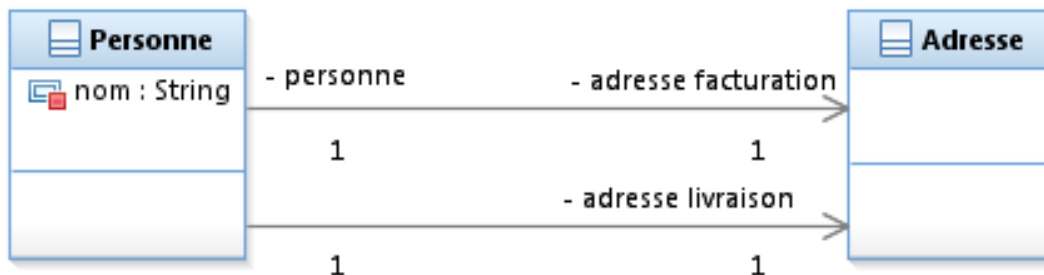


- **Remarque : on nomme très rarement les associations**

- Le nom sert à identifier, pour faire référence dans un texte par exemple
- RSA vérifie l'unicité et déclenche un avertissement en cas de doublon



- On préfère plutôt nommer le **rôle** joué par chaque élément
 - En l'indiquant aux extrémités de l'association
 - Permet de lever les ambiguïtés, en particulier en cas d'associations multiples
 - Le rôle peut être modifié en le cliquant et en saisissant directement au clavier
 - On peut le modifier dans la vue **Propriétés** en sélectionnant l'association
- Remarques
 - Le rôle peut être absent s'il est évident (mais RSA le génère automatiquement)
 - Le rôle est du côté de l'élément référencé (inverse du schéma entités-relations)
 - Un rôle est équivalent à un attribut (seulement depuis UML 2)



- **Les associations indiquent aussi des multiplicités (cardinalités)**
 - Correspond au nombre d'occurrences possibles entre les instances
- **Les multiplicités sont exprimées avec un intervalle**
 - 2..5 signifie entre 2 et 5 occurrences
 - 0..1 signifie que l'occurrence est optionnelle
 - 2 est équivalent à 2..2 (exactement 2)
 - 1..* signifie « 1 ou plus » (* est un symbole spécial indiquant « plusieurs »)
 - * est un raccourci pour 0..*
 - **Remarque** : l'absence de multiplicité est ambiguë
 - En général, équivalent à 1 mais peut être qu'on ne sait pas encore combien précisément

Exemple



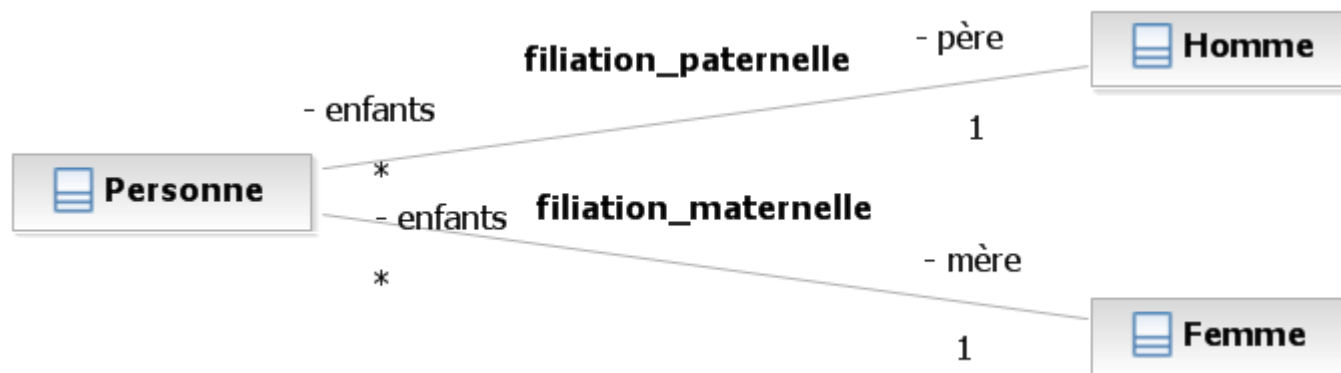
- Ici, chaque personne est associée à 0, 1 ou plusieurs comptes (*)
- Tandis que chaque compte est associé à exactement une personne (1)



Un homme peut être marié ou célibataire.
Il ne peut avoir qu'une seule épouse.
Il est l'époux de son épouse.
De même, une femme peut être célibataire.

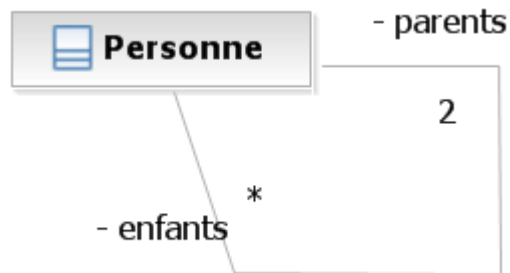


Un homme peut avoir été marié plusieurs fois.
Une femme aussi



Une personne possède un seul père et une seule mère.
Un homme peut être le père de plusieurs personnes, ses enfants.
Une femme peut être la mère de plusieurs personnes, ses enfants.

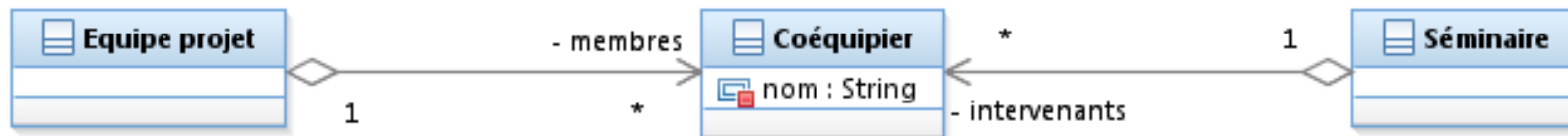
- **Une classe peut être en association avec elle même**
 - Les constructions récursives utilisent régulièrement ce type d'association



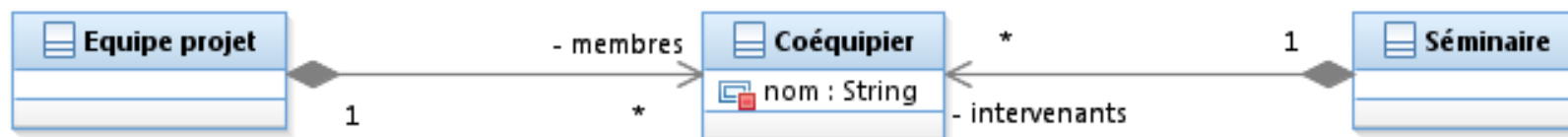
- **Les associations peuvent correspondre à un agrégat**
 - Indique une asymétrie dans l'association (un des éléments « contient » l'autre)

- **UML propose 3 niveaux**

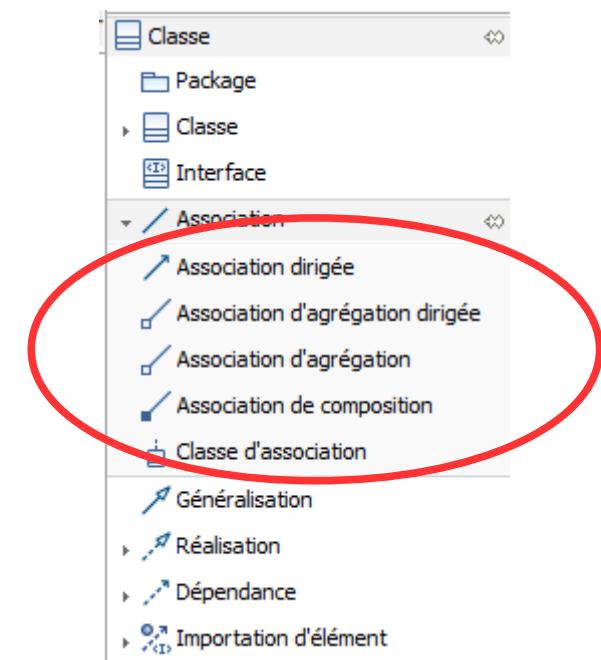
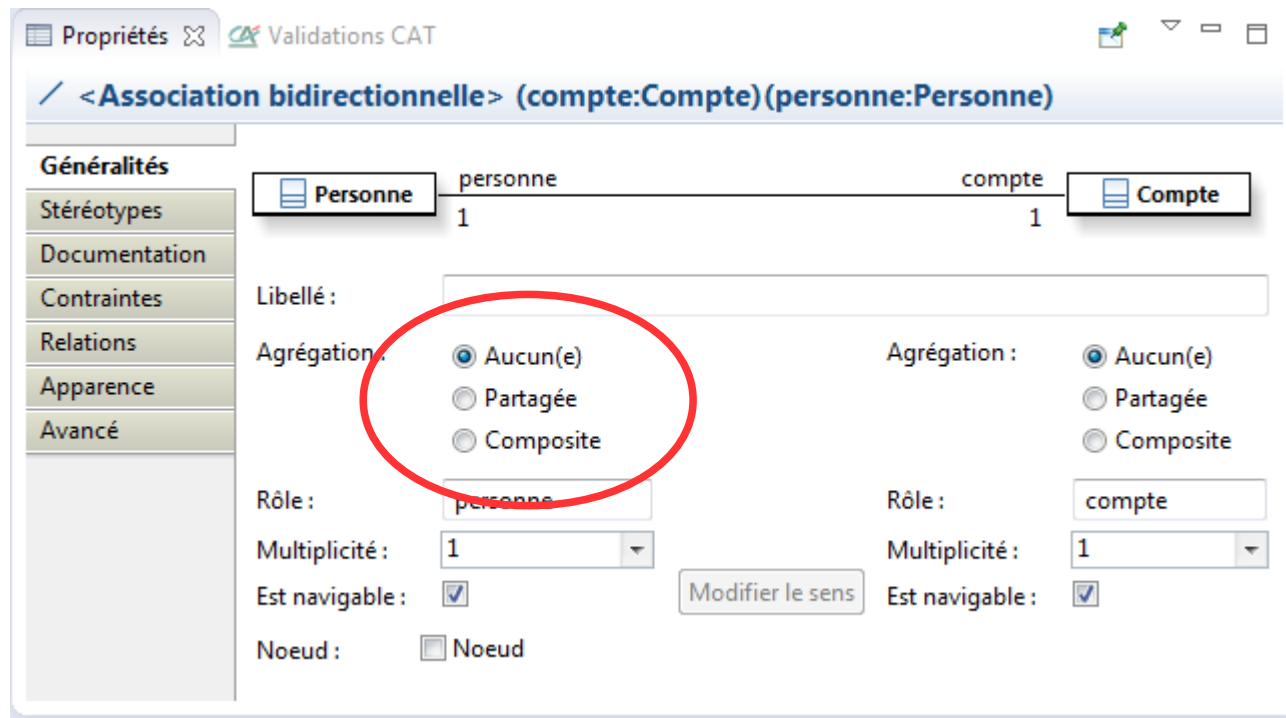
- **Association simple** : les éléments sont seulement associés
- **Agrégation (losange blanc)** : les éléments peuvent être partagés
 - Chaque équipe projet et chaque séminaire est « constitué » de coéquipiers
 - Un même coéquipier peut être dans une équipe et intervenir dans un séminaire



- **Composition (losange noir)** : les éléments ne peuvent pas être partagés
 - Correspond généralement à une contrainte sur le cycle de vie des éléments (et à une exclusion)
 - L'équipe projet est « composée » de coéquipiers (supprimer l'équipe supprimera les coéquipiers)
 - Un même coéquipier ne peut donc pas être en même temps intervenant dans un séminaire

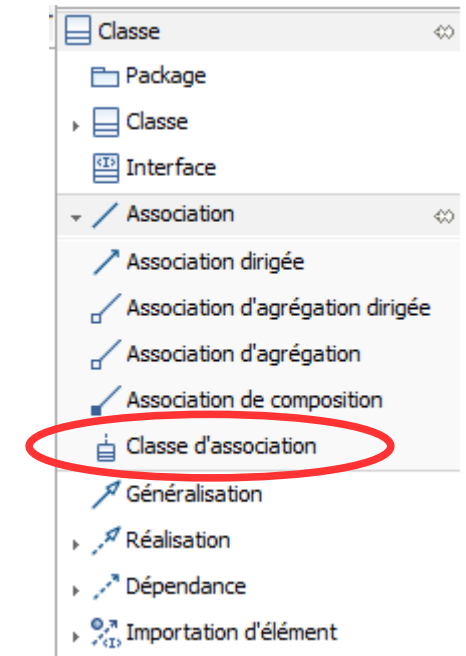
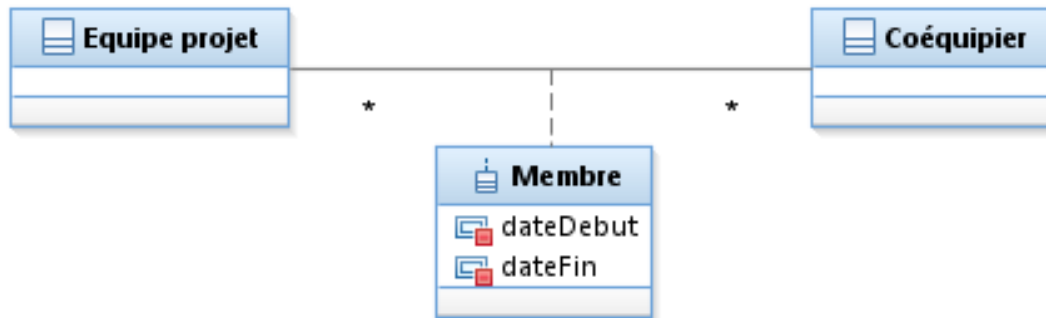


- Le niveau d'agrégation peut se choisir à la création (palette)
- Ou bien se changer après coup (vue Propriétés)

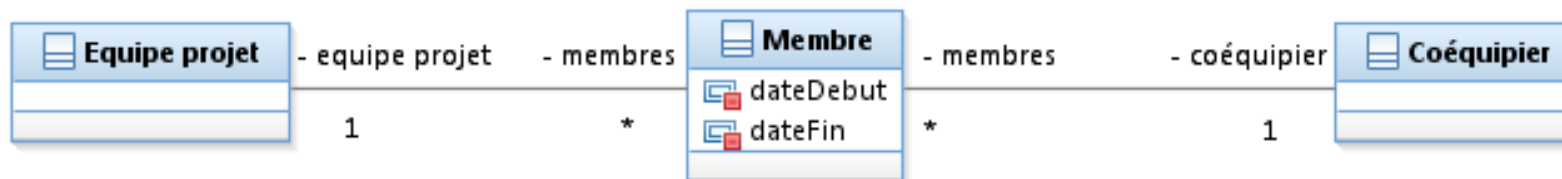


- **La palette propose aussi la Classe d'association**

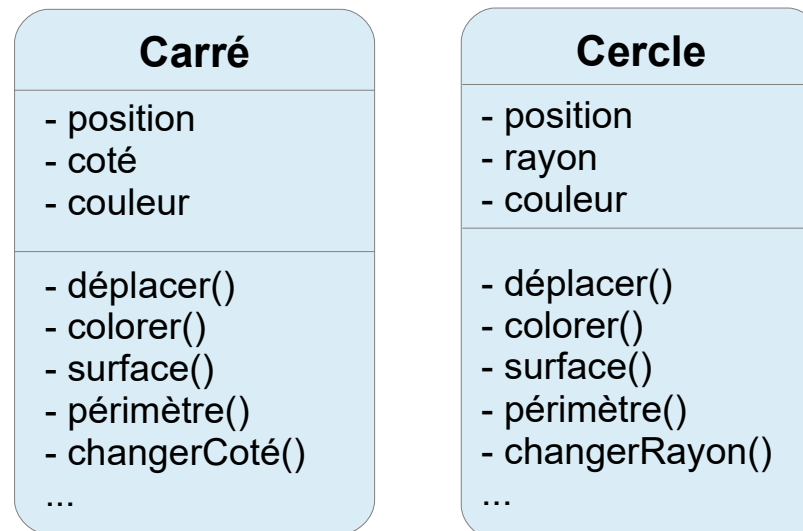
- Une mauvaise traduction de « Association as class »
- Lorsque l'association porte de l'information (attributs)
- Ici, on conserve les dates de début et fin pour chaque coéquipier dans chaque équipe projet où il est membre



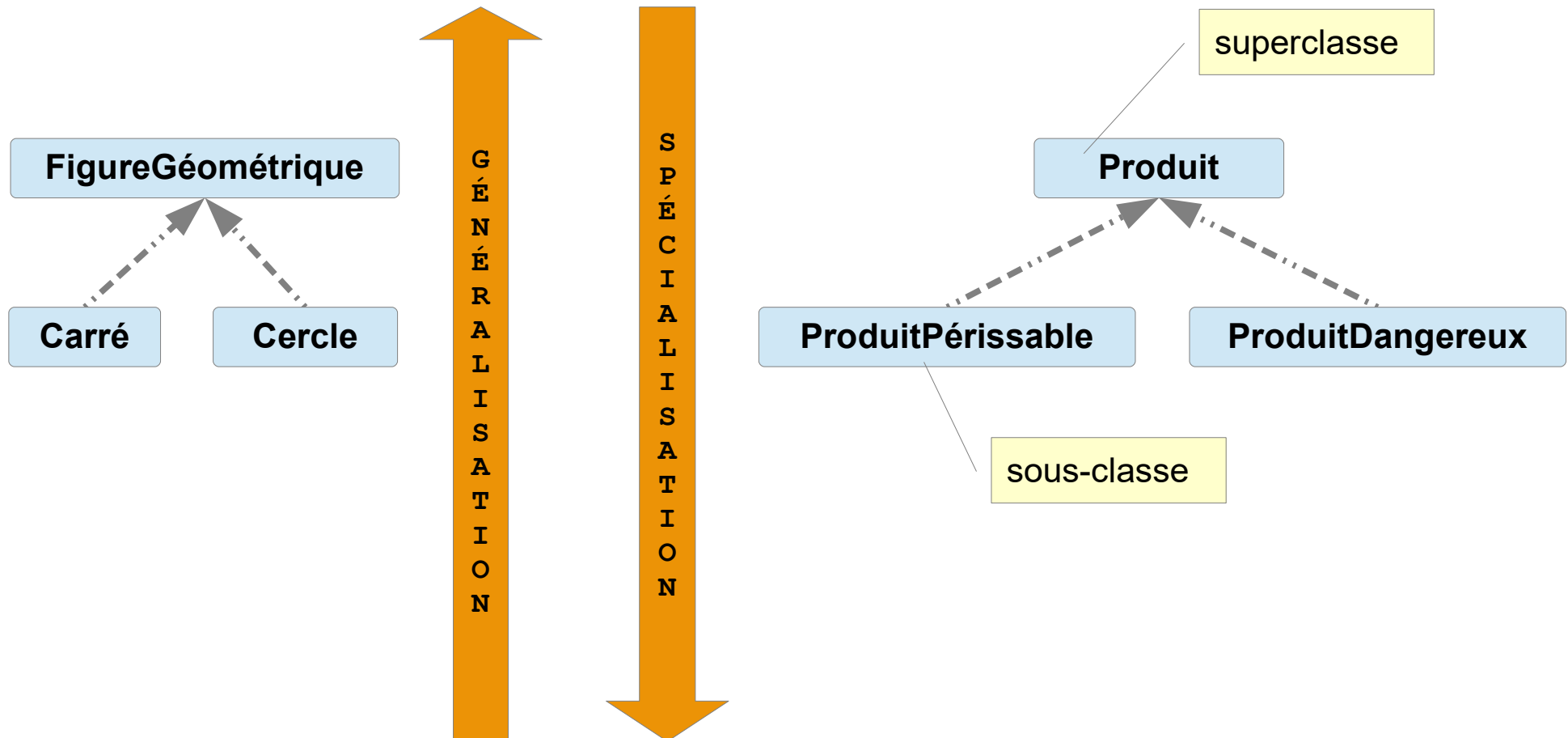
- Remarque : on préfère généralement créer une classe à part



- **La conception fait parfois apparaître des similitudes entre classes**
 - Des attributs identiques, des comportements similaires, ...
- **La relation d'héritage permet de les exprimer formellement**
 - On réunit ce qui est commun et on ne décrit que le delta
- **Exemple**
 - Les classes **Carré** et **Cercle** présentent des similitudes
 - Un concept plus général est envisageable, celui de **FigureGéométrique**

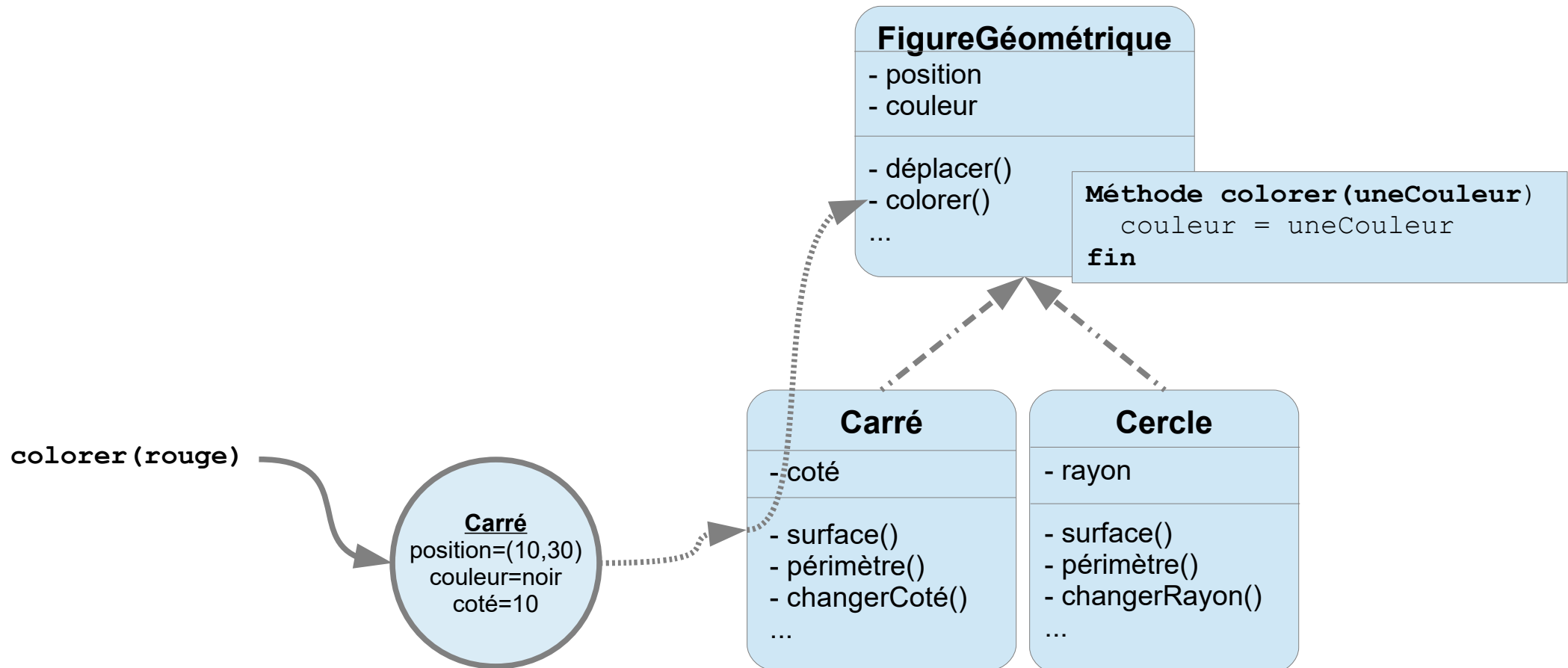


- Une question de point de vue et de chronologie de construction
 - Généralisation : on part de cas particuliers pour construire un concept général
 - Spécialisation : on part d'un cas général et on construit un cas plus spécifique

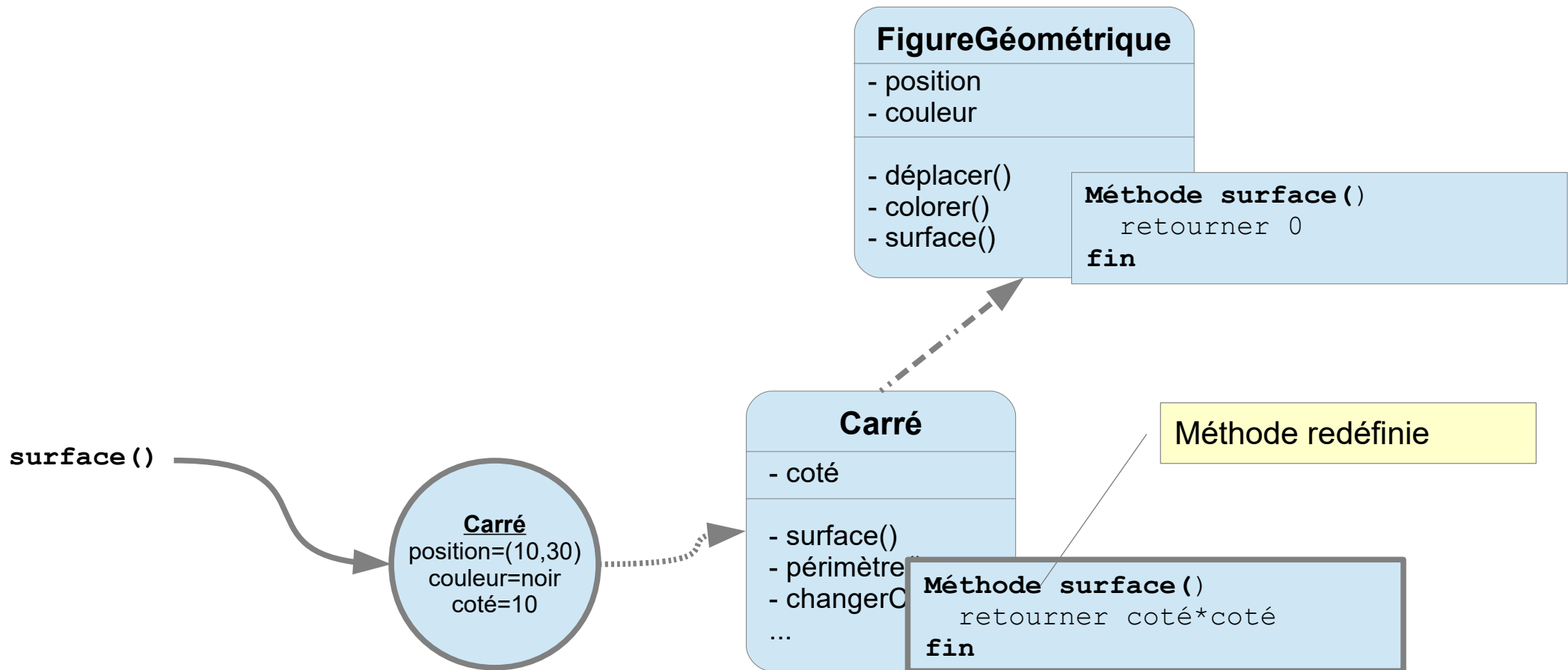


- **L'héritage prolonge la recherche de méthode**

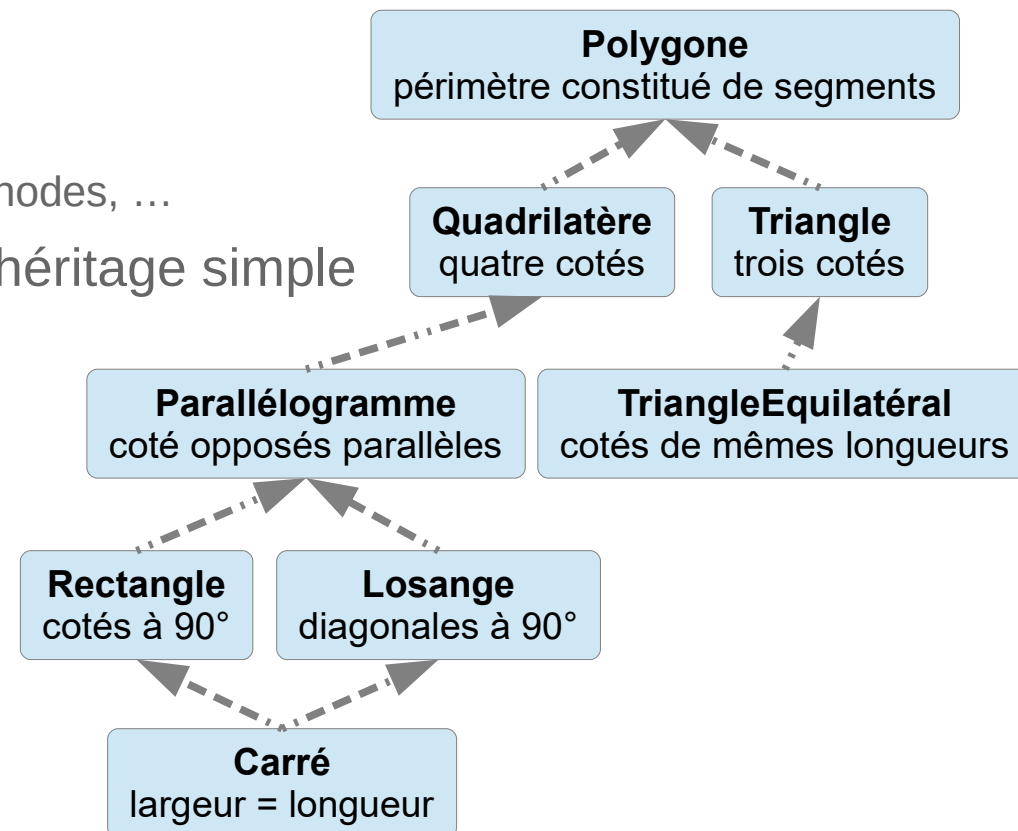
- Une instance qui reçoit un message, recherche dans les méthodes de sa classe
- Si elle ne trouve pas, elle recherche dans la superclasse
- La méthode trouvée est toujours appliquée dans le contexte de l'instance



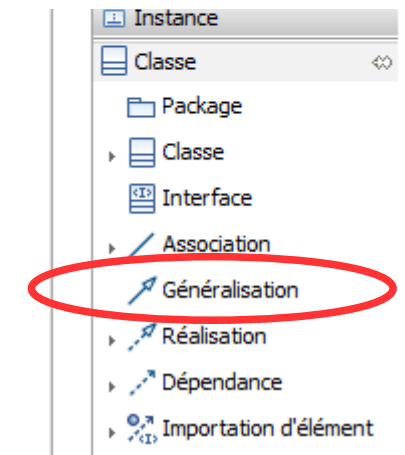
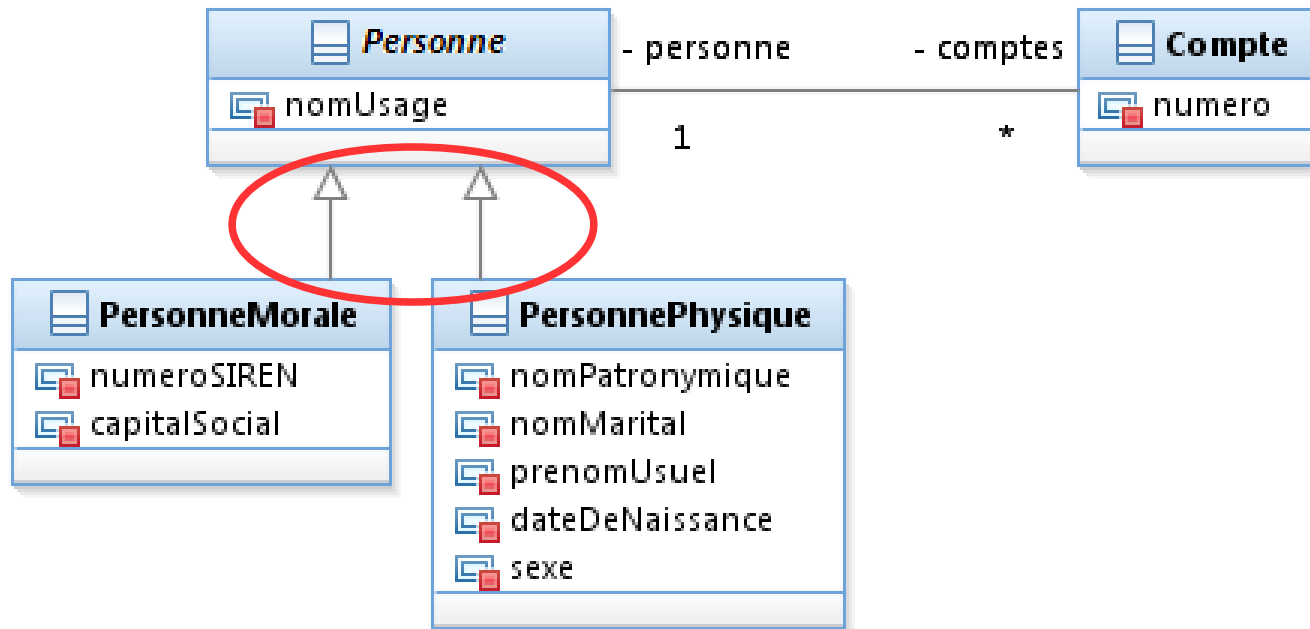
- Que se passe t-il si une méthode porte le même nom ?
 - Elle est tout simplement masquée pendant la recherche de méthode



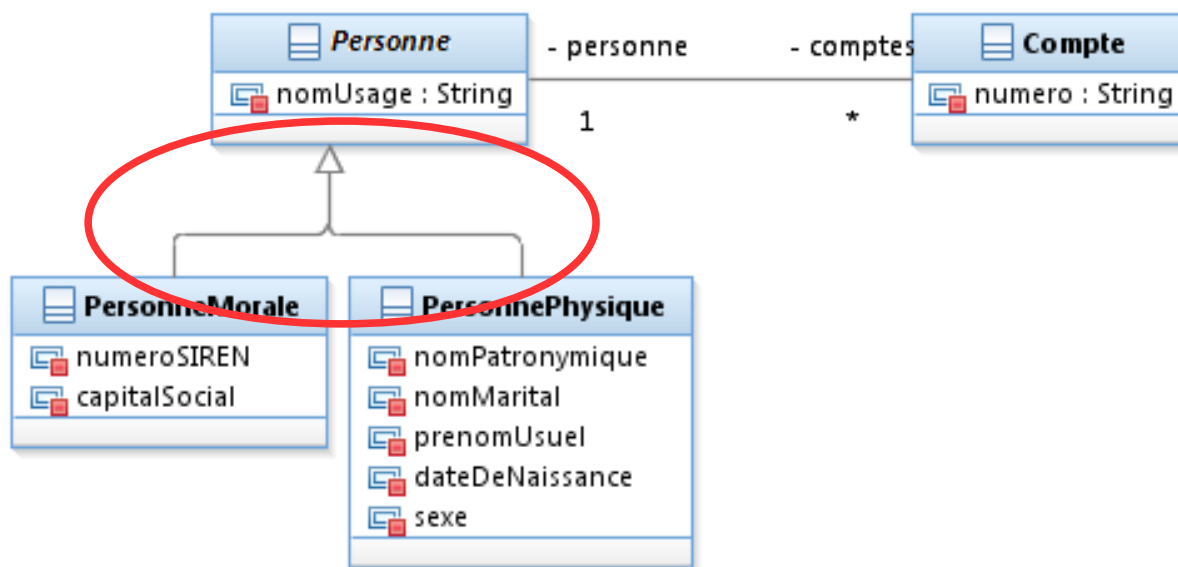
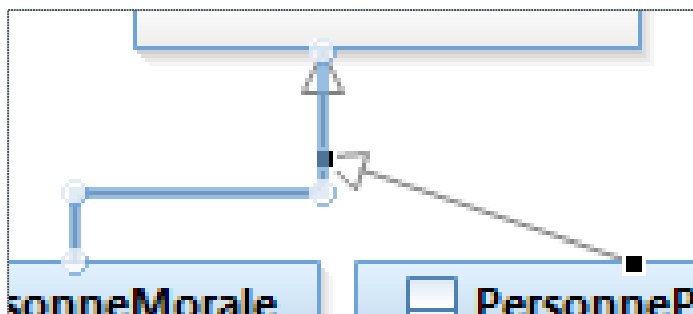
- **Conceptuellement, rien n'interdit l'héritage multiple**
 - Consiste à autoriser plusieurs super-classes pour une classe
 - Devrait permettre de factoriser encore mieux
- **Dans la pratique, de nombreux problèmes surviennent**
 - Fonctionne bien pour la description
 - Mais complique l'implémentation
 - Héritage multiple d'attributs, conflits de méthodes, ...
 - La plupart des langages objets sont à héritage simple



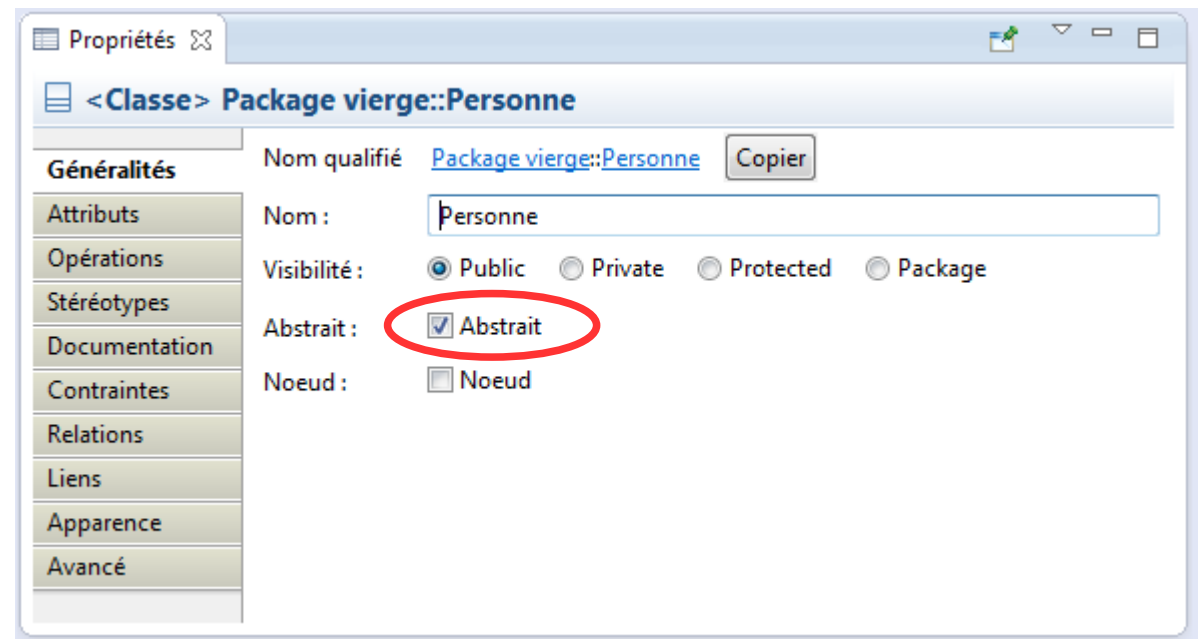
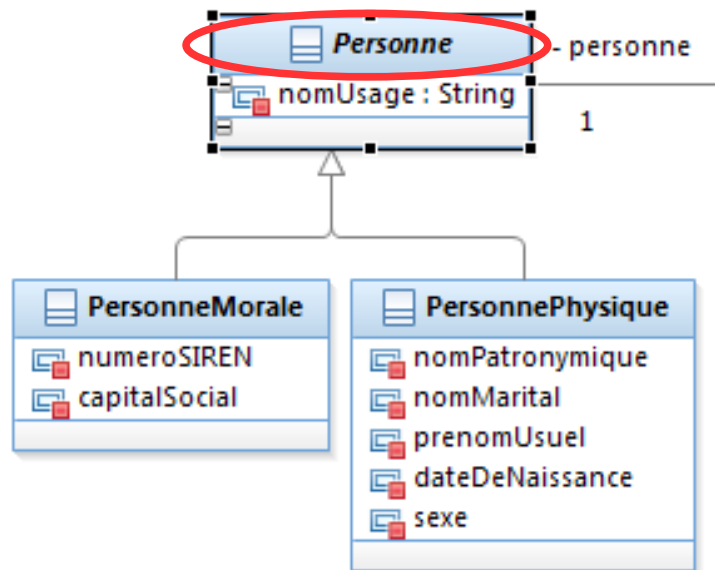
- La relation de généralisation correspond à ce lien d'héritage
 - L'élément cible est une « généralisation » du concept source
 - Les attributs et relations sont alors « héritées » (inutile de les repréciser)
 - Ici, la notion de **PM** et **PP** se généralise par le concept de **Personne**
 - Toute personne a un nom d'usage et peut posséder des comptes
 - On parle de **superclasse**, ou **classe mère** ou encore **classe parente**
 - On parle de **sous-classe**, ou **classe fille** ou encore **classe enfant**



- Les flèches de généralisation peuvent être réunies
 - Rend plus explicite l'arbre d'héritage



- Dans l'exemple, il ne peut pas exister d'instance de **Personne**
 - Une personne est nécessairement soit une **PP**, soit une **PM**
 - Personne est un concept **abstrait** ajouté pour factoriser des points communs
 - On l'indique dans la vue **Propriétés** et le nom apparaît alors en **italique**
 - **Remarque** : une superclasse n'est pas toujours abstraite



- **L'exemple sur la redéfinition de méthode pose un problème**
 - On y définit qu'une **FigureGéométrique** a, par défaut, une surface de 0
 - En réalité, on ne peut pas calculer la surface, ça dépend vraiment de la figure
 - La seule chose sûre c'est qu'une vraie **FigureGéométrique** aura une surface
 - Et on aimerait qu'un développeur n'oublie pas de définir ce calcul
- **FigureGéométrique est une classe abstraite**
 - Elle ne sert qu'à représenter un concept abstrait
 - Elle ne peut pas permettre de créer des instances
 - Une figure est un **Carré** ou un **Cercle**
 - **surface()** est une **opération abstraite**
 - Une opération qui doit être redéfinie dans les sous-classes

FigureGéométrique

- position
- couleur

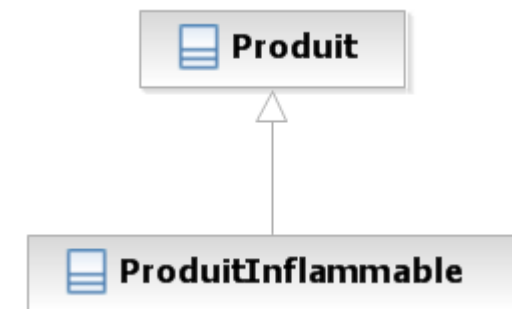
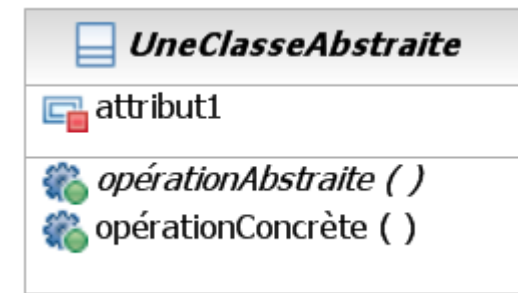
- déplacer()
- colorer()
- *surface()*

Méthode *surface()*

~~Retourner 0~~ abstraite
fin

Méthode abstraite

- **Une classe abstraite peut avoir :**
 - Une superclasse (abstraite ou concrète)
 - Des sous classes (abstraites ou concrètes)
 - Des opérations (abstraites ou concrètes)
 - Des attributs (un attribut ne peut pas être abstrait)
- **Remarque**
 - Une superclasse n'est pas toujours abstraite
 - Une classe abstraite sans sous-classe concrète est intrigante
 - À moins de modéliser un framework, c'est une incohérence



- **Ce n'est pas un terme informatique**

- Ensemble d'êtres ou d'objets réunis en raisons des traits qui leur sont communs
- Catégorie de choses, collection d'objets ayant un ou plusieurs points communs

- **Deux raisons**

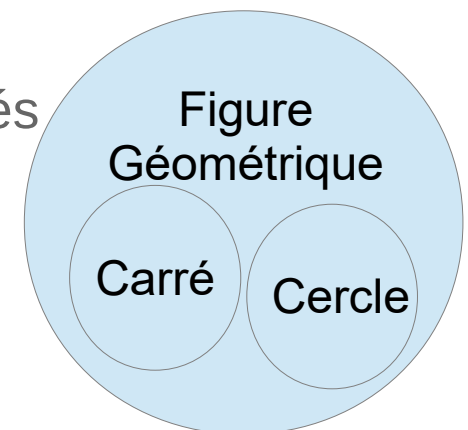
- Une classe regroupe des individus partageant des caractéristiques communes
- Une classe permet d'opérer une classification, c-à-d classer des éléments

- **La classe correspond donc à une vision ensembliste**

- Elle regroupe un ensemble d'individus (d'instances)

- **Et l'héritage correspond à un sur-ensemble**

- L'ensemble des figures géométriques contient celui des carrés
- Tout carré est aussi une figure géométrique

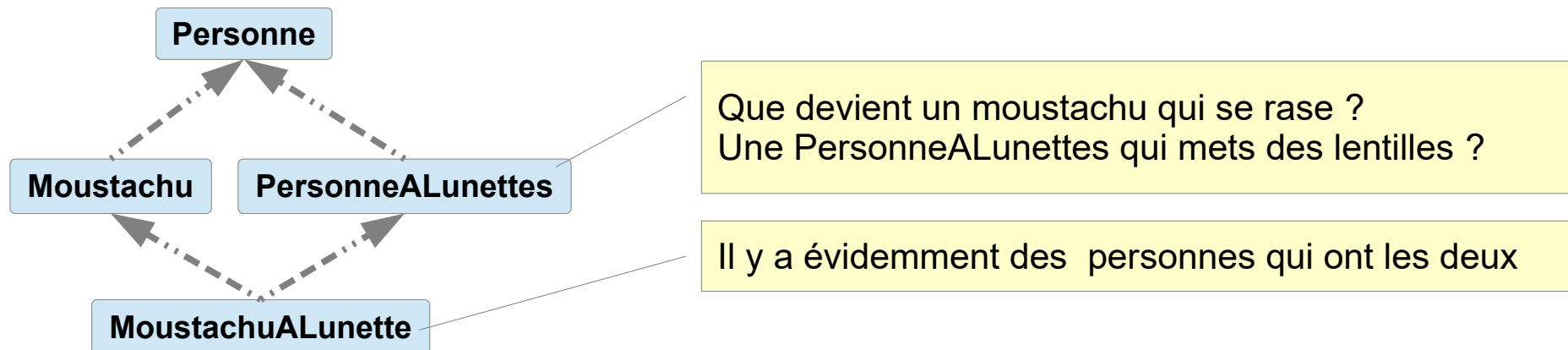


- **Classer**

- « Ranger, distribuer dans des catégories distinctes selon des critères définis »

- **Le choix des critères de classification est important**

- Il faut prendre des critères stables (les objets ne changent pas de classe)
- Prendre de mauvais critères nécessite rapidement de l'héritage multiple
- Et conduit rapidement à des situations complexes

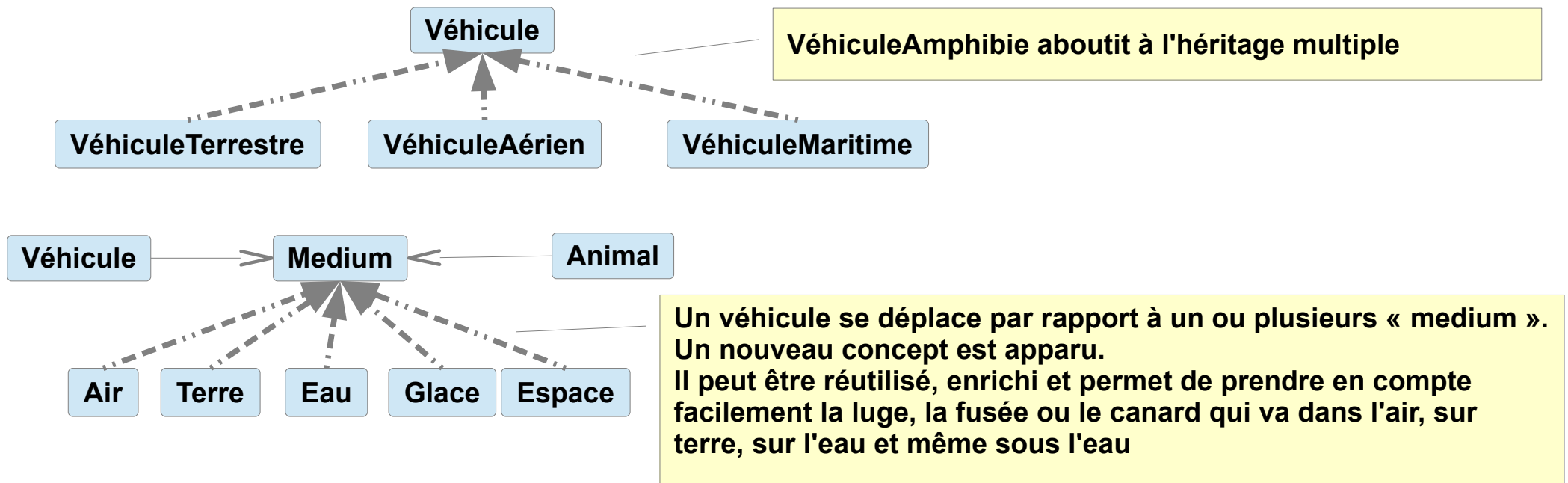


- **Limiter l'utilisation de l'héritage**

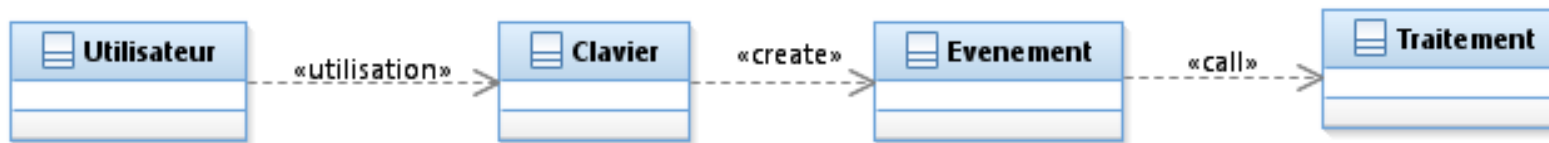
- C'est une relation complexe et contraignantes

- **Privilégier l'agrégation et la délégation quand c'est possible**

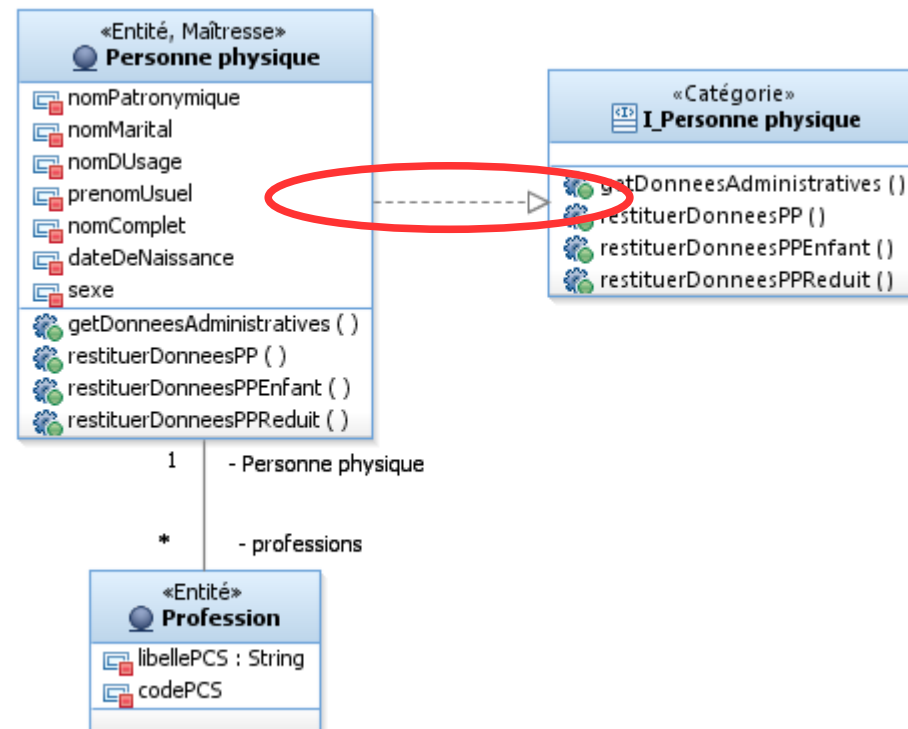
- En particulier lorsqu'un nom composé apparaît
- Un effort supplémentaire d'analyse permet souvent de passer à une association
- Et apporte souvent une meilleure évolutivité



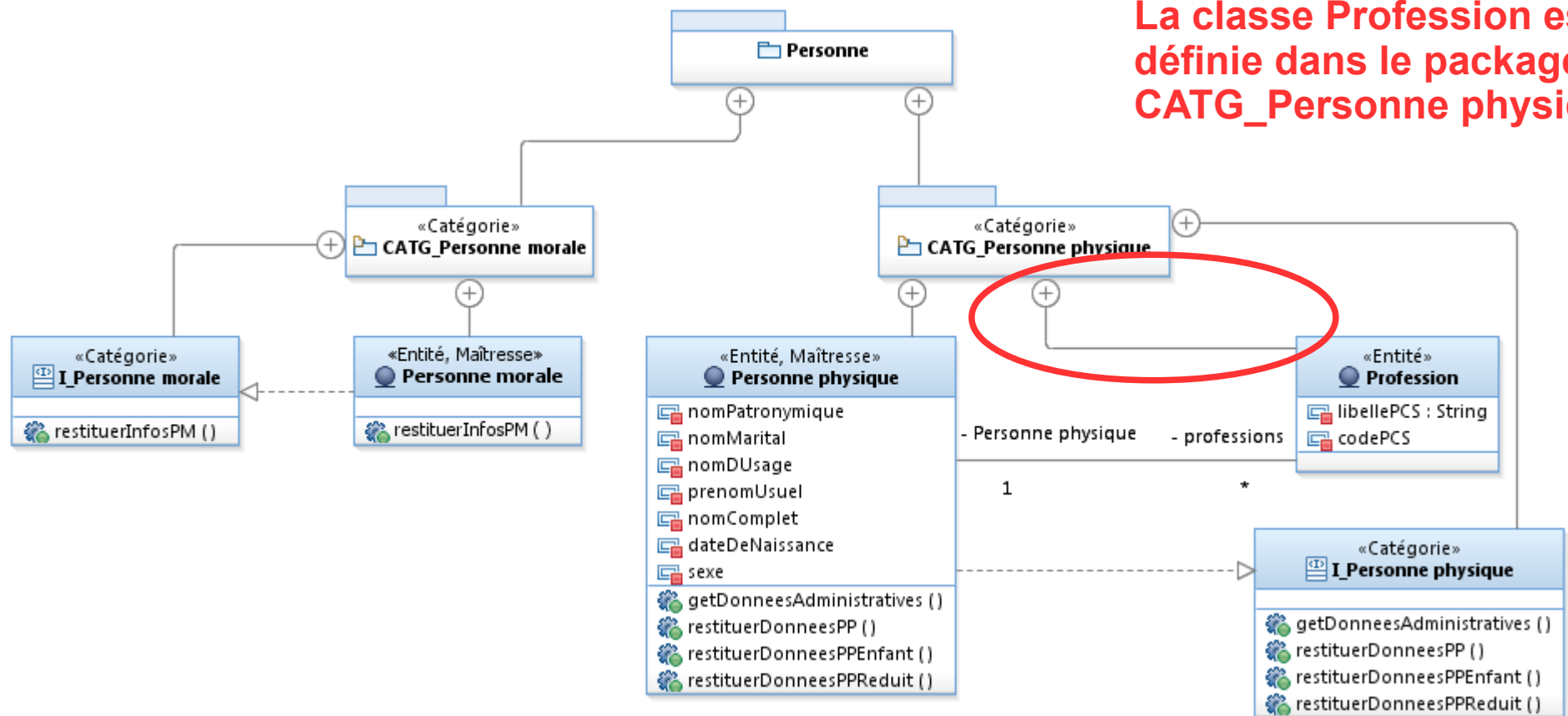
- UML propose d'autres types de relations comme la dépendance
 - La relation de dépendance est un lien non structurel (ne change pas la structure)
 - Il indique une dépendance, quelconque, entre les deux éléments
 - La nature de la dépendance est généralement indiquée par un **stéréotype**
 - Un marqueur entre chevrons << >>



- La relation de **réalisation** indique un lien de contractualisation
 - L'élément source **réalise** les spécifications décrites par la cible
 - On dit aussi qu'il **implémente** la cible
 - Typiquement pour indiquer qu'une classe réalise les opérations d'une interface
 - Une **interface** est une sorte de contrat spécifiant les opérations d'un type

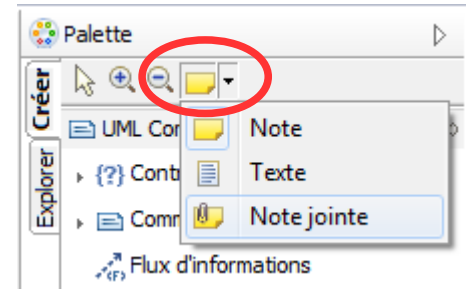


- La relation de confinement indique un lien d'imbrication
 - L'élément cible est **défini** à l'intérieur de l'élément source
 - En général utilisée pour montrer une arborescence de packages et leurs classes



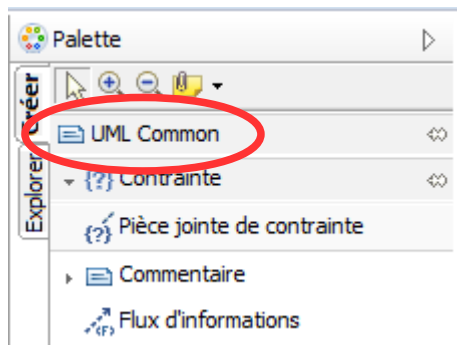
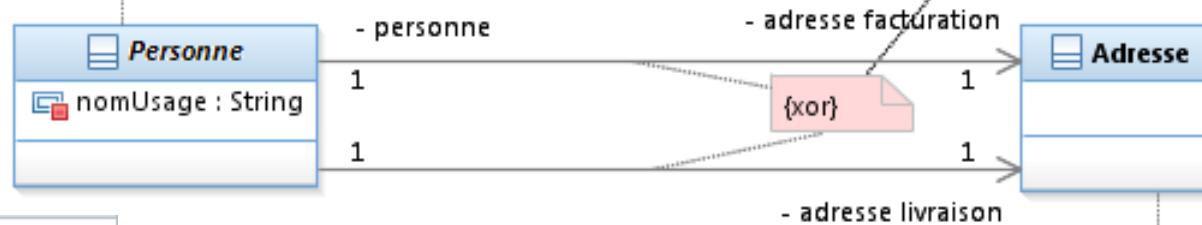
- **Les annotations introduisent un autre type de lien**

- Il se note en pointillés
- Il indique à quel(s) élément(s) se rapporte l'annotation



un commentaire attaché à **Personne**
Ex : on distingue adresse de livraison et de facturation

une note attachée à la contrainte.
L'exclusion indique ici que la même instance d'adresse ne peut pas être utilisée en même temps pour la livraison et la facturation



{Contrainte attachée à Adresse
Ex : si on modifie l'adresse de livraison, ça ne doit pas se répercuter sur l'adresse de facturation sans prévenir}



Les concepts avancés

- **Comment naissent les objets ?**

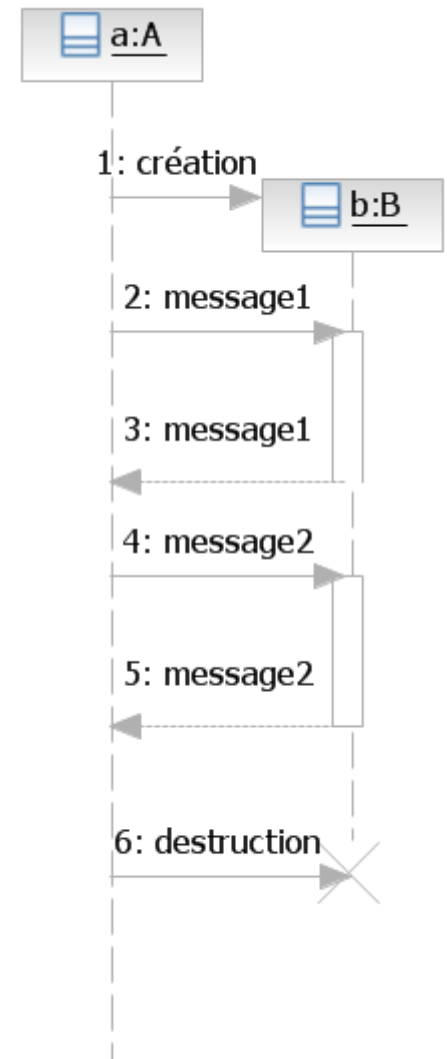
- Ça dépend du système/langage objet
- Création par un opérateur spécial (généralement « new »)
- Ou par un message adressé à la classe

- **Comment disparaissent les objets ?**

- Ça dépend aussi du système/langage objet
- Destruction par un opérateur spécial (destructeur)
- Ou par « GarbageCollector » (gestion automatique)

- **UML reste neutre**

- On montre le moment de création sur la ligne de vie
- Déclenchée par l'invocation d'une opération de création
- On montre la fin par un X en interrompant la ligne de vie
- Éventuellement déclenchée par une opération de destruction



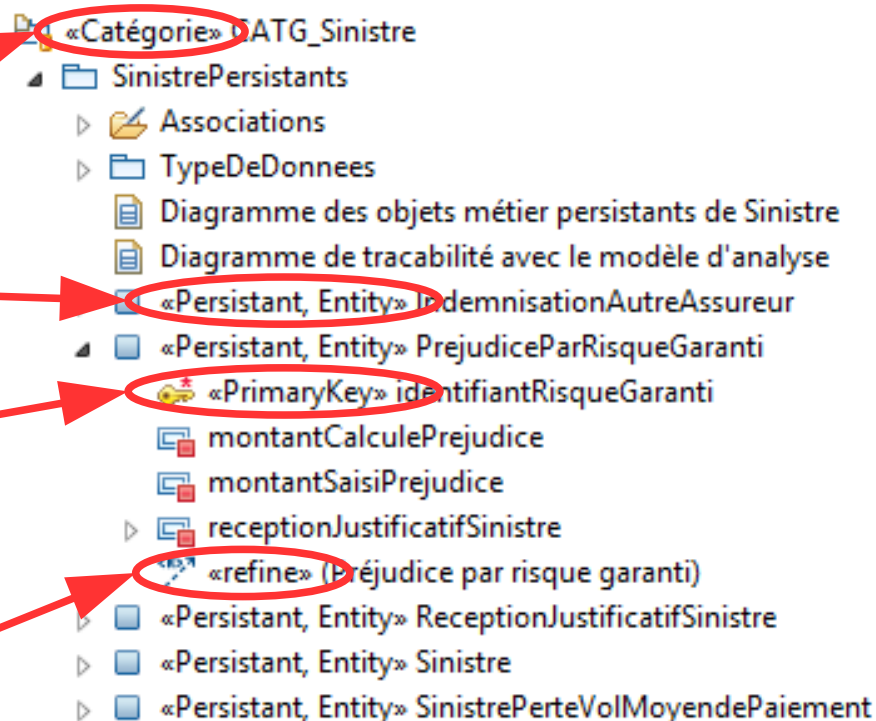
- **Un stéréotype permet de marquer un élément avec un mot clé**
 - Un stéréotype est noté entre chevrons (ex: «Entité», «Persistant», «Maîtresse»)
 - Il permet d'indiquer toutes sortes d'informations standards ou spécifiques
 - Sur toutes sortes d'éléments (classe, attribut, opération, relations, ...)
 - Il rend UML générique, sans multiplier le nombre de symboles

**Package représentant
une Catégorie**

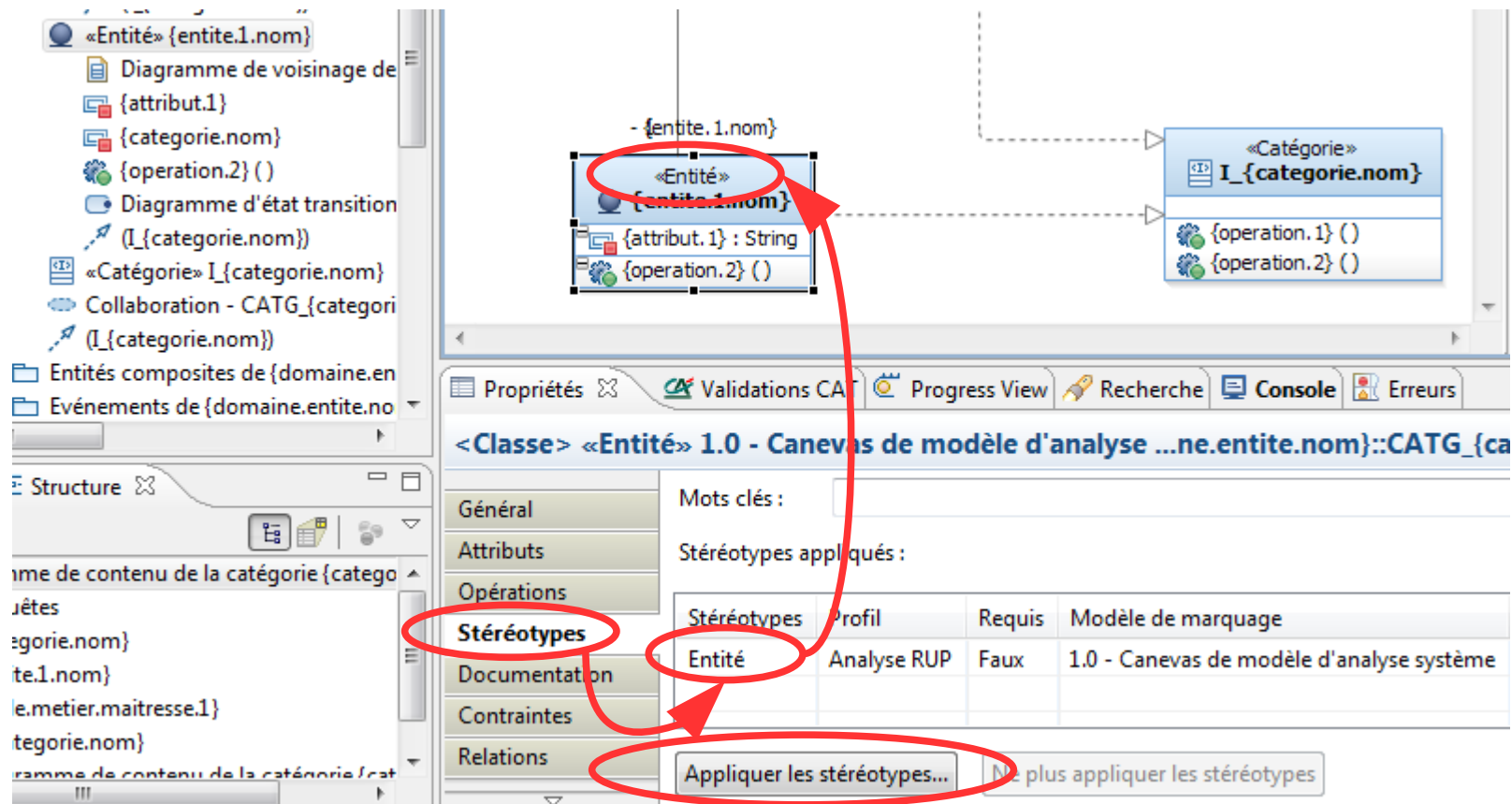
**Classe représentant
une entité persistante**

**Attribut représentant
une clé primaire**

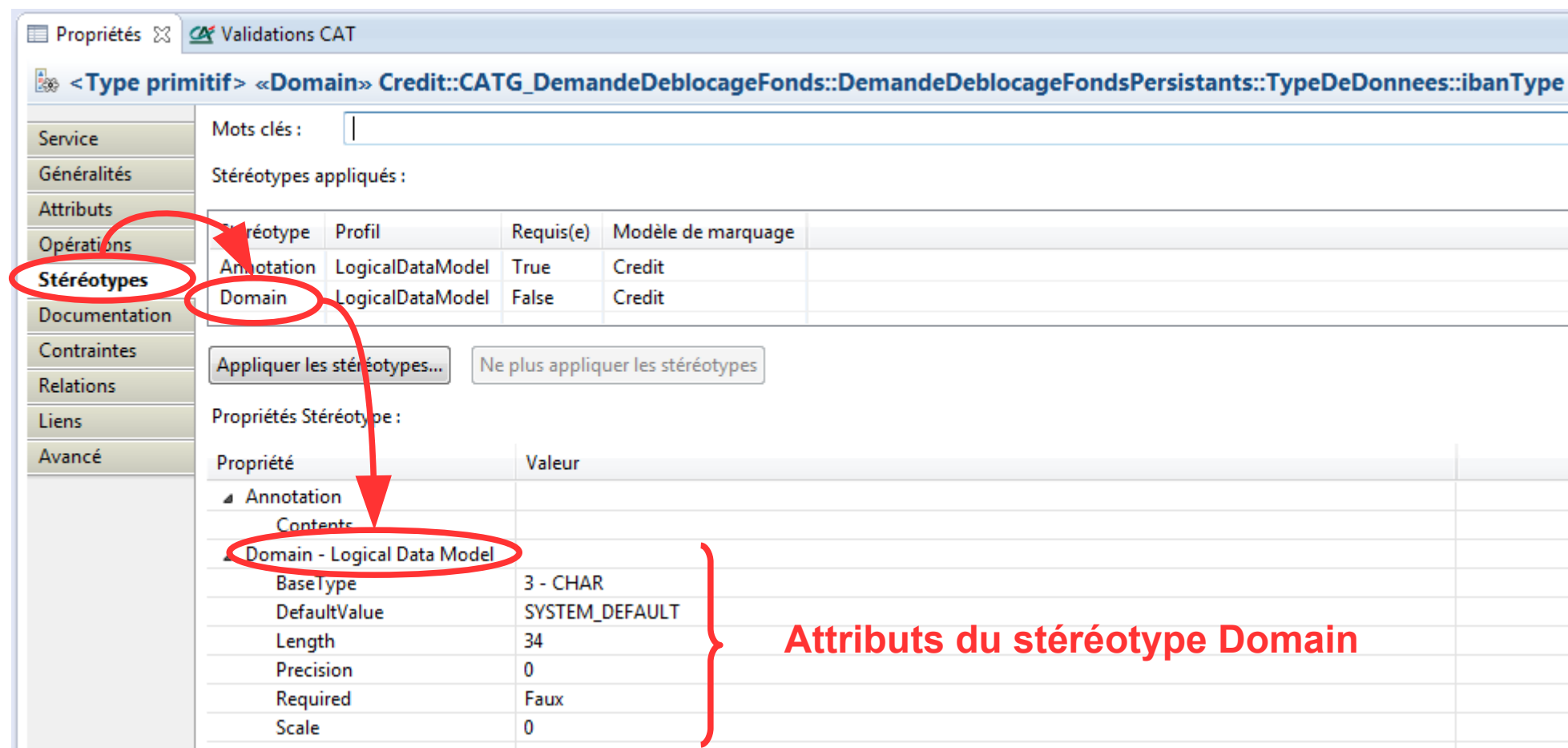
**Dépendance représentant
un raffinement**



- Dans la vue Propriétés, onglet Stéréotypes
 - Bouton **Appliquer les stéréotypes...** : choisir
 - Bouton **Ne plus appliquer les stéréotypes** : enlever le stéréotype



- **Un stéréotype n'est pas qu'un simple mot clé**
 - C'est un objet pouvant lui-même contenir des attributs
 - Ces attributs sont modifiables dans la vue **Propriétés**



Propriétés

<Type primitif> «Domain» Credit::CATG_DemandeDeblocageFonds::DemandeDeblocageFondsPersistants::TypeDeDonnees::ibanType

Mots clés :

Stereotypes appliqués :

Stereotype	Profil	Requis(e)	Modèle de marquage
Annotation	LogicalDataModel	True	Credit
Domain	LogicalDataModel	False	Credit

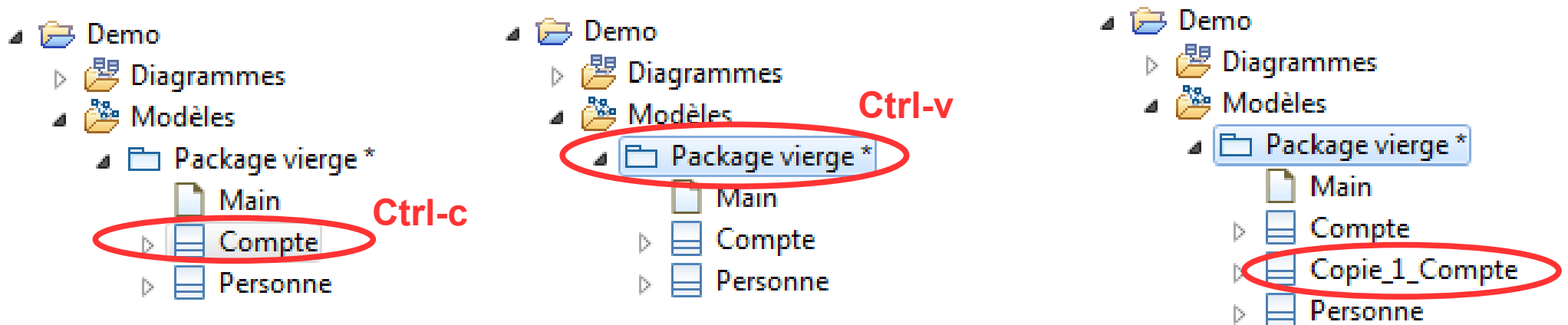
Appliquer les stereotypes... Ne plus appliquer les stereotypes

Propriétés Stereotype :

Propriété	Valeur
Annotation	
Contenu	
Domain - Logical Data Model	
BaseType	3 - CHAR
DefaultValue	SYSTEM_DEFAULT
Length	34
Precision	0
Required	Faux
Scale	0

Attributs du stéréotype Domain

- **Toujours copier/coller dans l'arbre du modèle, pas le diagramme**
 - On veut dupliquer un élément, pas sa représentation
 - Renommer ensuite l'élément copié et le glisser/déposer dans le diagramme
- **La copie est une copie en profondeur**
 - La copie d'une classe copie ses attributs, opérations, stéréotypes, relations, ...
 - La copie d'un attribut copie ses caractéristiques (type, multiplicité, ...)
- **On peut copier plusieurs éléments en même temps**
 - Les liens entre ces éléments seront alors copiés aussi



- **L'héritage pose beaucoup de problèmes**

- Une relation très forte et structurelle
- Une relation complexe, souvent mal comprise et mal utilisée
- Une complexité encore plus particulière avec l'héritage multiple

- **Une classe représente 3 choses**

- La définition d'attributs et de méthodes
- La capacité à créer des instances (le moule pour fabriquer les instances)
- Un type, qui permet de typer des variables, attributs, paramètres, ...

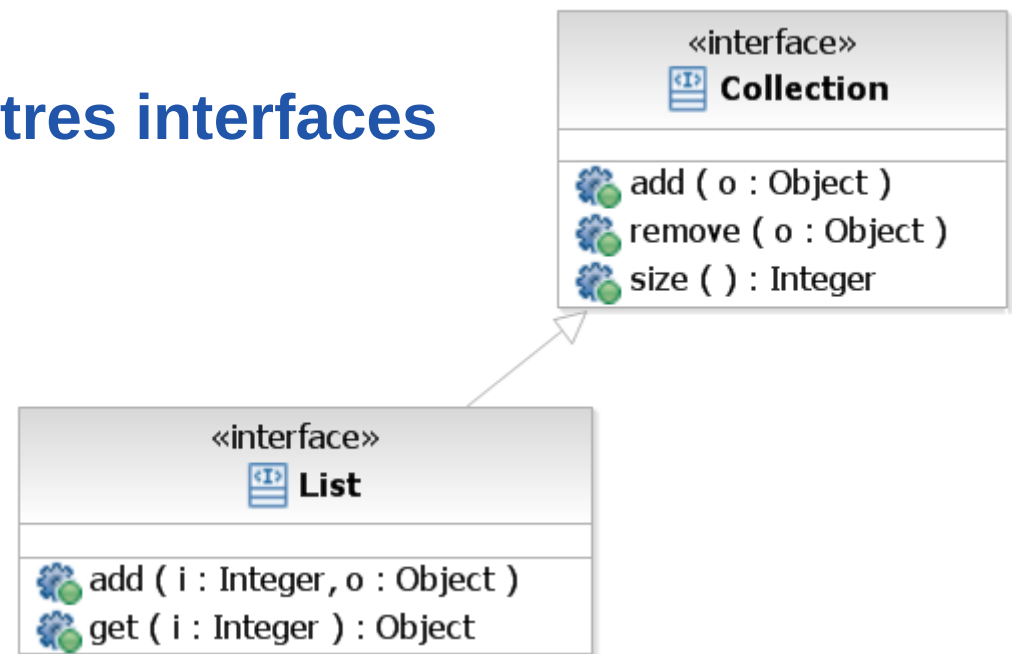
- **Lorsqu'on hérite, on hérite de ces 3 choses**

- De nombreux systèmes ont choisi de rester en héritage simple

- **On a souvent besoin de représenter spécifiquement un type**

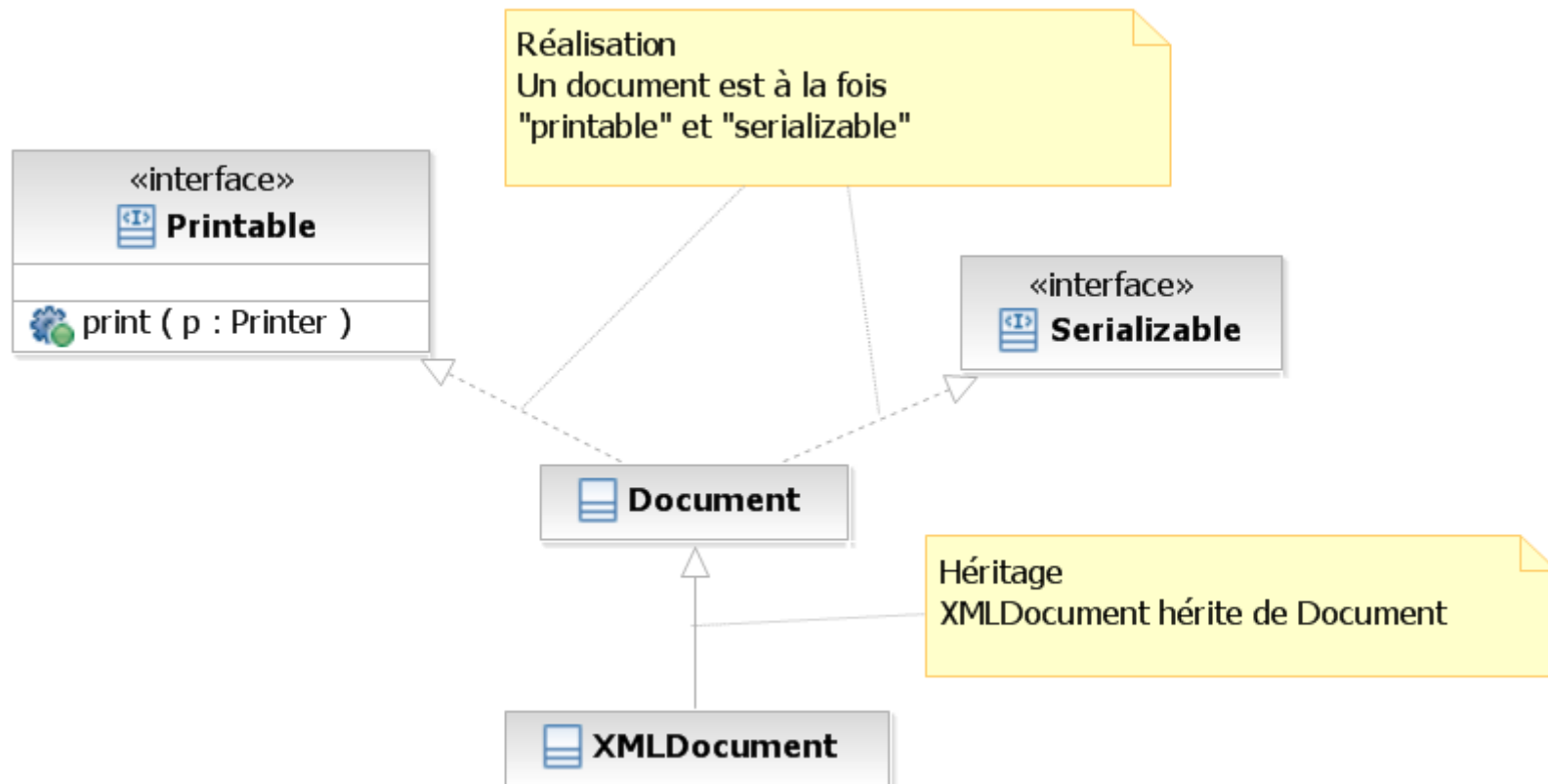
- Théorie aboutie sur les types abstraits
- Les classes abstraites ne suffisent pas : il faut un concept spécifique

- Une « interface » pour représenter un type abstrait
 - Une sorte de classe purement abstraite, popularisée par Java
 - Définit un « contrat » en permettant de spécifier des opérations (abstraites)
 - Permet de typer des variables (attributs, paramètres et retour d'opérations, ...)
 - Techniquement, une classe annotée par un stéréotype « interface »
- Le nom d'une interface est souvent suffixé par « able »
 - Traduit la notion de « capable »
- Une interface peut hériter d'autres interfaces
 - Correspond à un sous-type



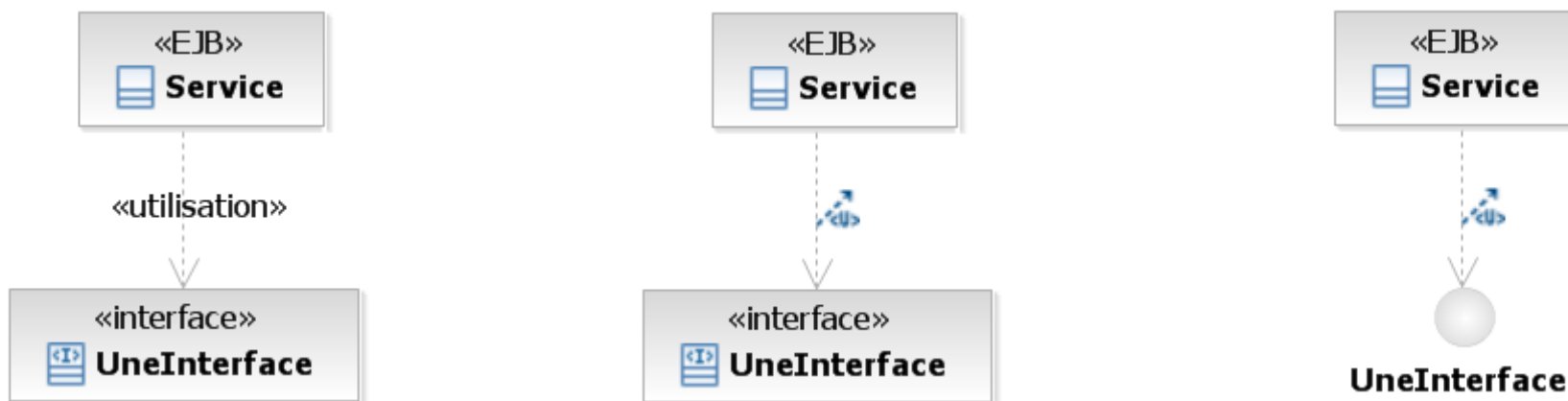
• Réalisation

- Une classe peut « réaliser » des interfaces (en Java : « implémenter »)
 - Elle s'engage à réaliser le « contrat » en implémentant les méthodes de l'interface
 - Les interfaces permettent ainsi de diminuer le besoin pour l'héritage multiple
- Attention : différent de l'héritage



- **Les interfaces permettent de diminuer le « couplage »**
 - En établissant un « contrat » sans rien imposer sur le « fournisseur » du contrat
- **Deux éléments sont couplés lorsqu'ils dépendent l'un de l'autre**
 - Deux classes sont couplées lorsqu'une relation existe entre ces deux classes
 - Concrètement : on ne peut pas extraire et réutiliser l'une sans extraire l'autre
- **Ce couplage peut être plus ou moins fort**
 - Dépend du type et du nombre de références entre les classes
 - Un héritage entre deux classes est plus fort qu'une simple utilisation
 - La force du couplage correspond à l'effort (jH) à fournir pour les séparer
 - Un couplage fort rend difficile maintenance, évolutions futures et réutilisation
- **Diminuer le couplage est un objectif primordial de la conception**

- **Les stéréotypes disposent d'une forme textuelle**
 - Entre chevrons (ex : « utilisation »)
- **Il peuvent disposer aussi d'une forme graphique**
 - Une image qui peut être utilisée sous forme d'icône
 - L'image peut même parfois être utilisée comme forme de boîte



- « énumération »

- Correspond à un type énuméré
- Les valeurs possibles sont énumérées

- « signal »

- Information véhiculée lors d'une communication
- Ne contient pas d'opération
- Attributs = paramètres de la communication
- Émission asynchrone (l'émetteur n'attend pas)
- La réception correspond à un événement pour le récepteur
- Correspond aux exceptions dans Java par exemple

- « type de données », « primitive »

- Représente des types de données « non objet »



- **Difficulté d'exprimer la sémantique sur un diagramme**
 - Les diagrammes doivent être accompagnés d'un document
 - UML prévoit aussi les notes
- **Différentes possibilités pour exprimer les contraintes**
 - Contraintes structurelles : attributs, types de relations, cardinalités, ...
 - Contraintes de type : type des attributs, paramètres, retour, ...
 - Autres : visibilité, abstraction (classe, méthode, interface, ...)
- **S'avère insuffisant pour exprimer toutes les subtilités**
 - C'est pourquoi UML prévoit la notion de contrainte

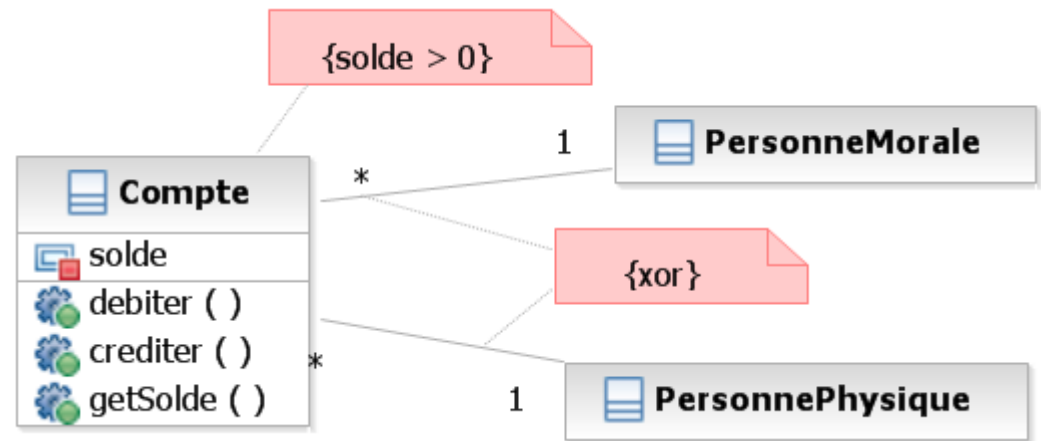
- **Une contrainte exprime une restriction sémantique**
 - On l'exprime à l'aide d'une condition à satisfaire pour une implémentation correcte du système
 - On la note entre accolades (ex : {age > 18})
- **On peut décrire la condition de 3 manières**
 - En langue naturelle : on reste alors à un niveau purement descriptif
 - Directement dans un langage de programmation : le modèle devient alors dépendant du langage
 - En OCL (Object Constraint Language) : la solution standardisée
- **OCL est la solution préconisée par l'OMG**
 - Mais très lourde à utiliser (OCL est un vrai langage dédié aux contraintes)

- **Sur une classe**

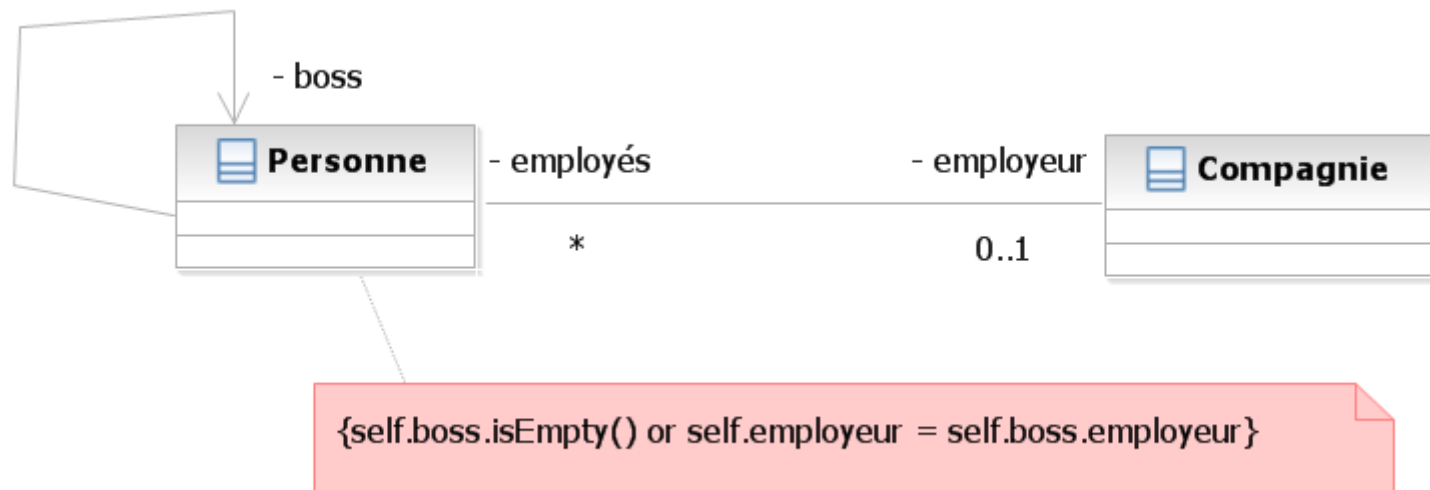
- Ou sur un attribut
- Ici le solde doit être positif

- **Sur une association**

- Ou entre des associations
- Ici un compte appartient à une PersonneMorale ou à une PersonnePhysique
 - Mais pas les deux en même temps



- **Exemple avec OCL**



- **Quelques contraintes que l'on peut rencontrer**
 - **{xor}** : exclusion entre associations
 - **{subset}** : une association est un sous ensemble d'une autre
 - **{ordered}** : les éléments de l'association restent dans l'ordre
 - **{list}** : les éléments de l'association sont indexés par un indice
 - **{addOnly}** : on ne peut pas enlever des éléments de l'association
 - **{readOnly}** : on ne peut pas modifier l'attribut
 - **{frozen}** : la valeur ne change plus après son initialisation
- **On en retrouve certaines dans les spécifications UML 2**
 - Le moment où on a cherché à spécifier UML entièrement en UML

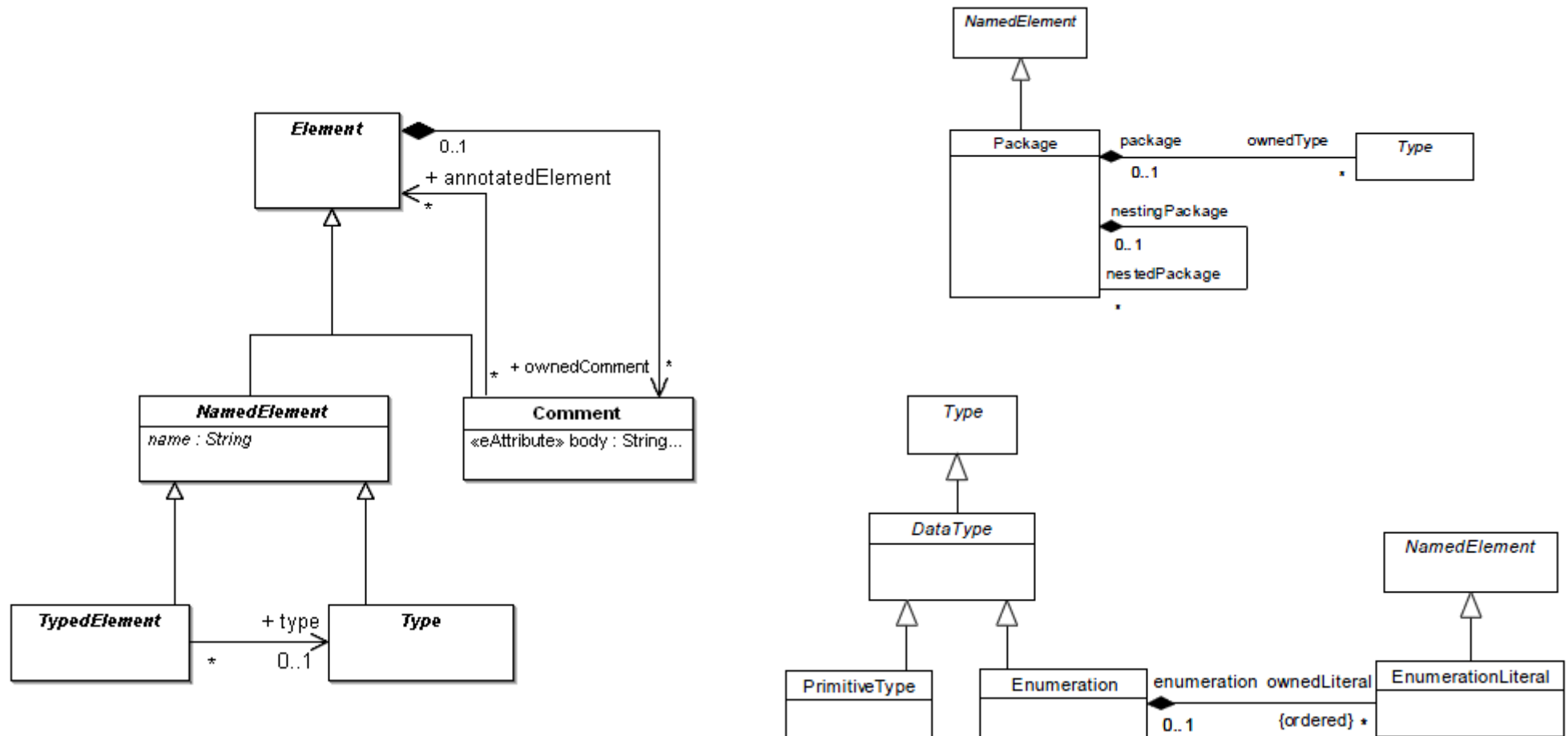
- **Diagramme des PrimitiveTypes**

- Les **UnlimitedNatural** incluent l'astérisque '*' pour désigner 'plusieurs'
- **Remarque** : il n'y a pas de nombre réel dans les spécifications UML

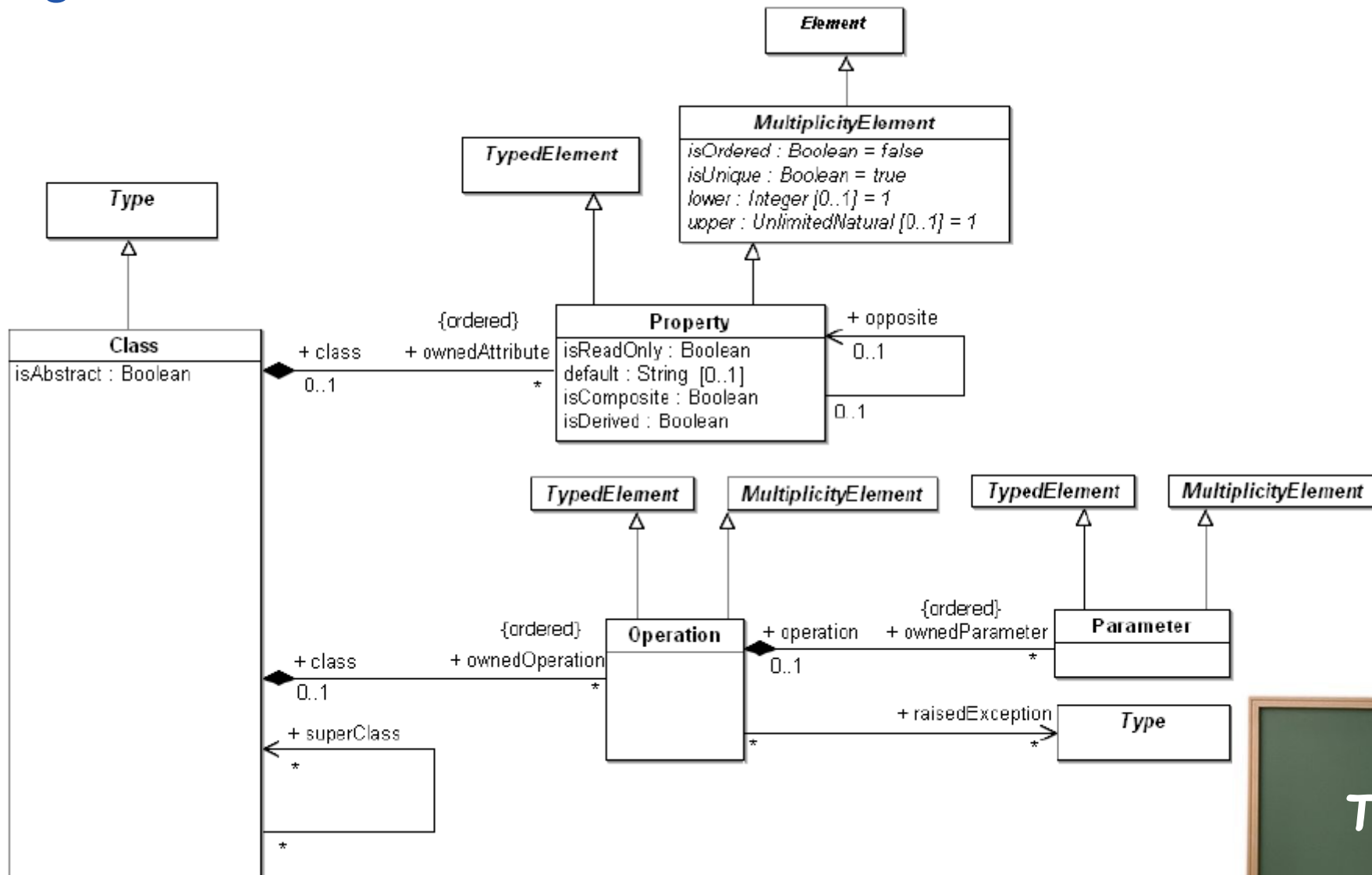


• Diagrammes des Types, DataTypes et des Packages

- Extraits des spécifications



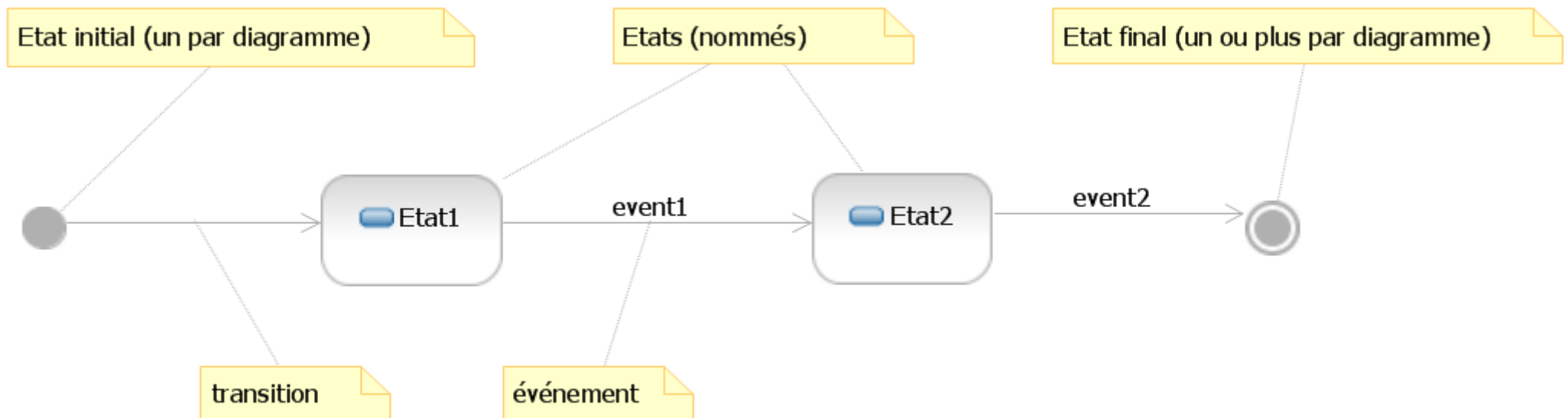
- Diagramme des classes



TP 6

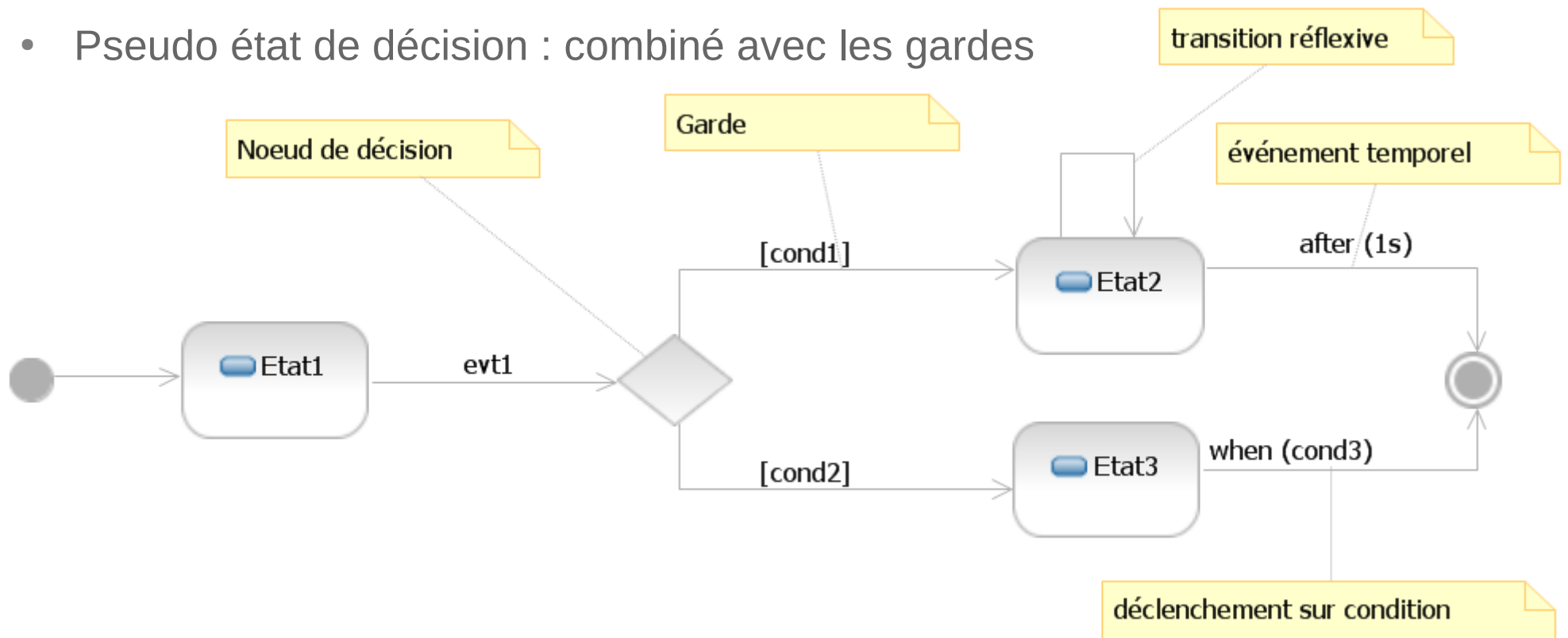
- **Certains objets ont un cycle de vie complexe**
 - L'ordre des messages qu'on leur envoie est important
 - Ils se comportent différemment selon les situations
- **Exemples**
 - Cycle de vie d'une commande (en création, validée, payée, expédiée, reçue, ...)
 - Dialogue avec une base de données (connexion, requêtes, déconnexion)
- **Notion d'état**
 - Un statut remarquable durant la vie d'un objet pendant lequel il satisfait une condition particulière, réalise une activité ou simplement attend un événement
- **Notion d'événement**
 - Un stimulus localisé dans le temps (réception d'un message ou d'un signal ou tout simplement l'atteinte d'un moment donné, la fin d'une activité, ...)

- **Décrit le fonctionnement d'un objet comme machine à états finis**
 - Définit ses états possibles (un nombre fini)
 - Définit les transitions possibles (les passages d'un état à l'autre)
 - Définit les conditions (stimulus ou événement) qui déclenchent ces transitions
- **États et transitions simples**
 - L'événement déclencheur peut être la réception d'un message ou d'un signal
 - L'interception d'une exception en Java par exemple

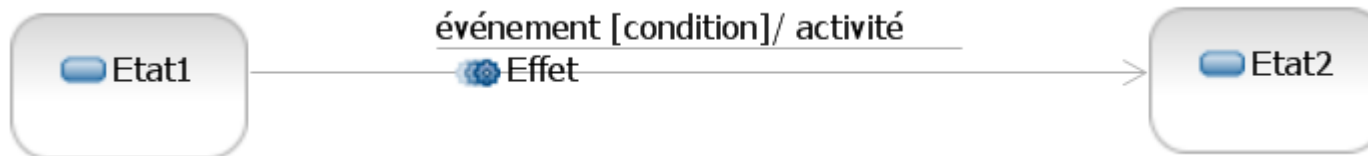


- Plusieurs subtilités sur les transitions

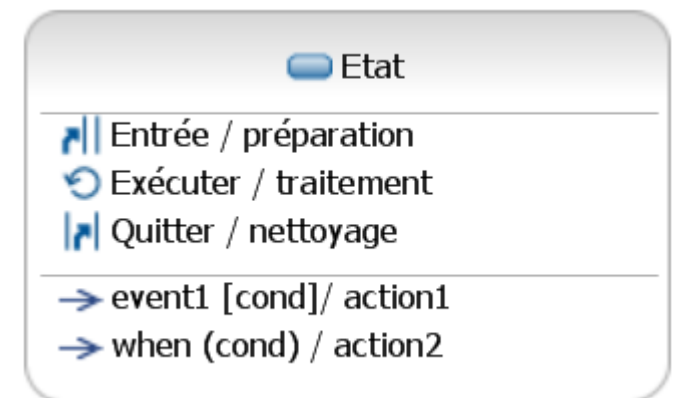
- Événements de déclenchement temporel : après un délai ou à un moment précis
- Événements de déclenchement sur la réalisation d'une condition quelconque
- Transition réflexive : revient sur le même état
- Garde : pour conditionner une transition
- Pseudo état de décision : combiné avec les gardes



- **Transition externe : celles représentées par une flèche**
 - Peut être déclenchée par un événement
 - Peut être gardée par une condition
 - Peut avoir un effet (en général une activité)



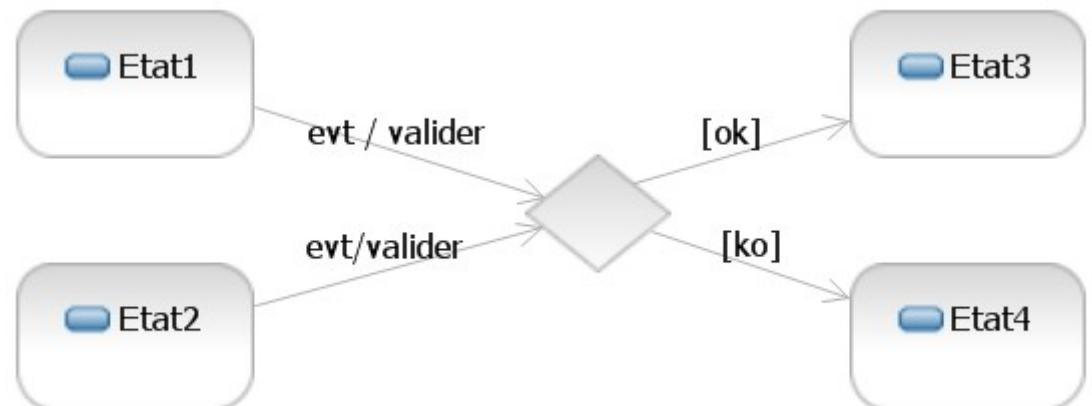
- **Transition interne : à l'intérieur de l'état (sans flèche)**
 - Même caractéristiques qu'une transition externe (événement, garde, activité, ...)
 - Mais ne provoque pas de changement d'état
- **Trois événements spéciaux**
 - « entry » : permet de déclencher une action sur l'entrée dans l'état
 - « do » : permet de lancer une activité qui s'exécute tant qu'on reste dans l'état
 - Interrompue dès qu'une transition se déclenche
 - « exit » : permet de déclencher une action sur la sortie de l'état
- **Remarque**
 - Une transition externe réflexive sort de l'état
 - Puis entre à nouveau
 - Ce qui provoque l'interruption du « do »
 - Ainsi que les événements « exit » et « entry »



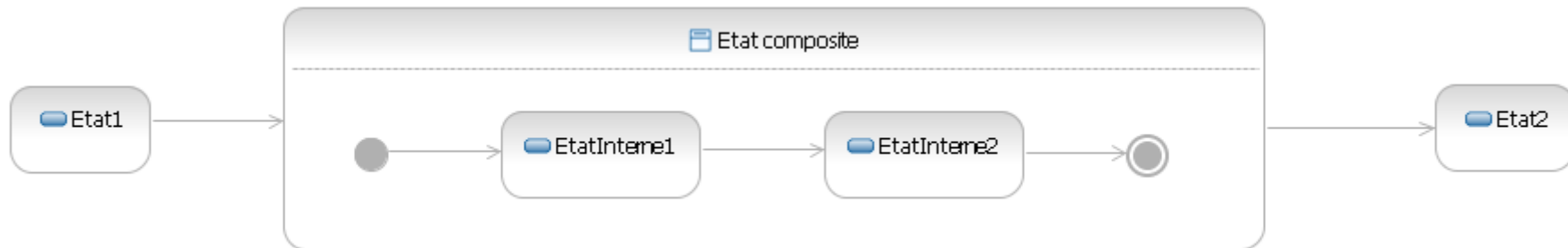
- **Un autre pseudo état pour simplifier un diagramme complexe**
 - Lorsque de nombreuses transitions similaires existent entre plusieurs états



- **Attention : ne pas le confondre avec le point de décision**
 - Qui impose plusieurs sorties
 - Et qui peut baser sa sortie sur le résultat de l'activité entrante



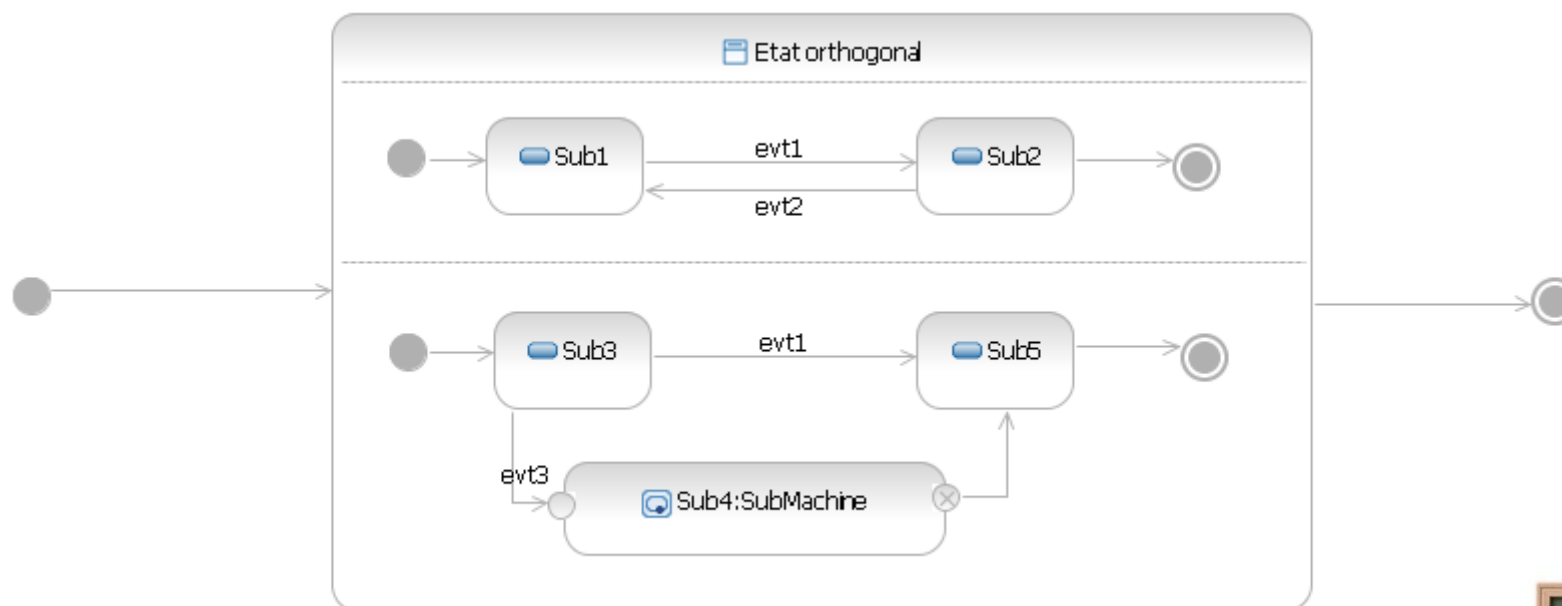
- **Un état composé de plusieurs sous états**
 - Lorsque l'activité de cet état est elle-même complexe



- Il peut éventuellement faire référence à une autre machine
 - On peut montrer les points d'entrée et de sortie



- **Un état composite peut contenir plusieurs « régions »**
 - Correspondent à des sous états « concurrents » (c-à-d en parallèle)
 - Dans ce cas, l'état composite est qualifié d'état « orthogonal »



Identifier les objets

- **Maintenant que nous savons comment représenter les objets**
 - Diagramme de classes, d'objets, de séquences, d'états, ...
- **Il nous faut apprendre à les débuser**
 - Pendant l'analyse et la conception
- **Les plus simples à découvrir : les objets métiers**
 - Ce sont généralement des objets concrets du monde réel
 - Une facture, un client, un produit
- **Objets secondaires**
 - Sont découverts pendant l'exploration des attributs et des relations
 - Seules les questions et mises à l'épreuve du modèle, permettent de l'améliorer
- **La phase de conception amènera des objets de 3ème niveau**
 - Objets techniques, classes abstraites ou intégrations avec des bibliothèques, ...

- **Cette démarche ne convient pas pour des projets réels**
 - On risque fort de passer à côté de certains aspects
 - Ou de se perdre dans des détails inutiles
- **La bonne approche est celle préconisée par UP**
 - Elle consiste en une gestion rigoureuse des exigences du projet
 - Chaque exigence donnant lieu à tout un ensemble d'activités
 - Analyse, conception, estimation, élaboration de tests, documentation, ...
 - Ces activités sont toutes tracées et reliées à l'exigence de départ
- **Le point de départ est d'identifier le périmètre du système**
 - Ce qu'il doit faire (ou ne pas faire) et par qui
 - Ces éléments actifs (qui initient des actions dans le système) sont les Acteurs
 - Les autres objets sont passifs et répondent aux sollicitations des acteurs
- **Cette phase constitue le BusinessModeling**

- **Un acteur est une abstraction d'un rôle**
 - Rôle joué par des entités externes (utilisateur humain, dispositif matériel ou autre système) interagissant directement avec le système
- **Un acteur se différencie d'un utilisateur**
 - En ce sens qu'il capture les besoins d'un utilisateur dans un des rôles spécifiques qu'il joue avec le système : il correspond à un profil utilisateur
- **Un UseCase est la représentation d'un processus métier**
 - C'est un ensemble de séquences d'actions réalisées par le système et produisant un résultat mesurable pour un acteur donné
 - Il doit être identifié par au moins quelques exigences fonctionnelles
 - Chaque exigence doit être associée à au moins un UseCase

- **Identifier les acteurs**

- Les candidats sont les utilisateurs humains directs et les systèmes connexes
- Il convient de répertorier tous les profils
 - Sans oublier administrateur, opérateur de maintenance, ...
- Ainsi que tous les systèmes qui interagissent directement avec le système
 - En général via des protocoles bidirectionnels

- **Identifier les cas d'utilisation**

- Chaque cas d'utilisation correspond à une fonction métier du système
 - Identifiée du point de vue de l'acteur, externe au système
- Pour chaque acteur
 - Rechercher les intentions métiers avec lesquelles il utilise système
 - Rechercher dans les exigences (cahier des charges) les services fonctionnels attendus

- **Chaque UseCase est alors analysé et décrit**
 - D'abord textuellement en général
 - On recense toutes les interactions entre les acteurs et le système
 - On identifie clairement le début et la fin du cas d'utilisation
 - Précondition et post-condition du use case
 - On précise les variantes possibles (cas nominal, cas alternatifs et cas d'erreur)
 - Cela donnera ensuite lieu à une représentation sous forme de scénario
 - Qu'on peut décrire avec le diagramme de séquence

- **Pour recueillir, analyser, organiser les besoins**

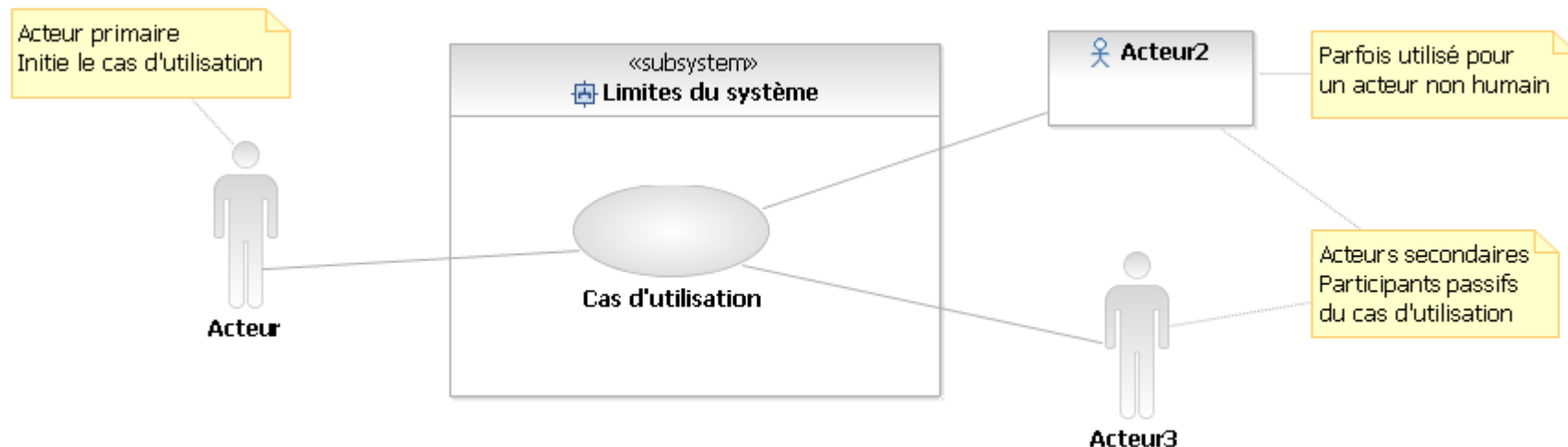
- Permet de recenser les grandes fonctionnalités du système
- Volontairement simple pour être compréhensible par les non informaticiens

- **Acteur**

- Un rôle joué par une personne, un processus ou un élément externe au système

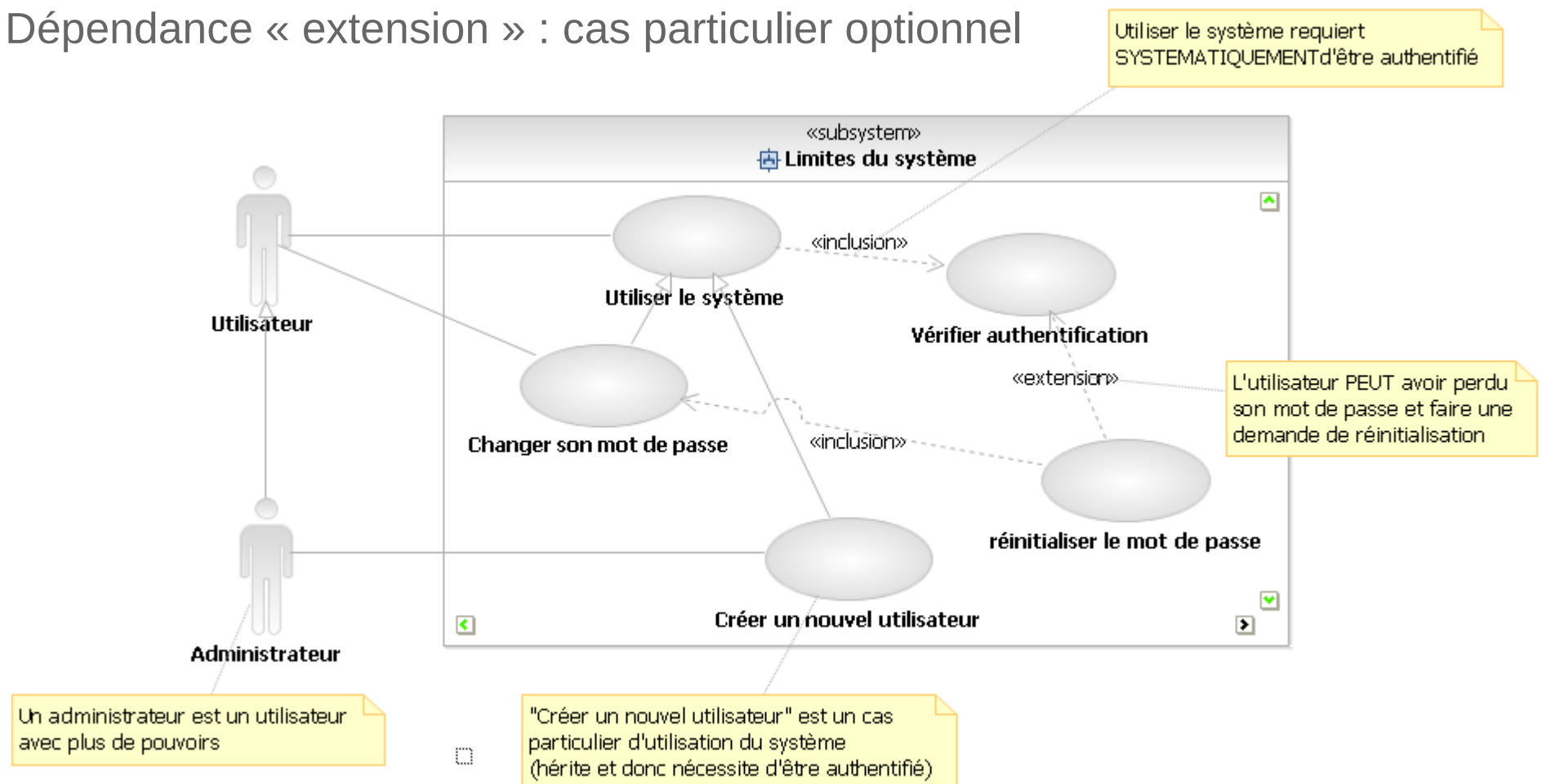
- **Use Case (cas d'utilisation)**

- Un service rendu par le système
- Associé à un acteur qui initie le cas d'utilisation (à gauche)



• Trois liens possibles

- Héritage entre acteurs ou entre cas d'utilisation
- Dépendance « inclusion » : cas « inclus » systématiquement
- Dépendance « extension » : cas particulier optionnel



- **Préciser les pré et post conditions**
 - Dans la description textuelle
- **Ne pas abuser des relations**
 - Elle sont difficiles à interpréter, surtout par des non informaticiens
 - Attention : il n'y a pas de chronologie dans les extensions
- **Ne pas multiplier les UseCase et rester très général**
 - De nombreux concepteurs recommandent moins de 20 UseCases
 - Les démarches agiles plus légères (scrum, XP, ...) recommandent le contraire
 - Beaucoup de petits UseCase (appelés user stories)



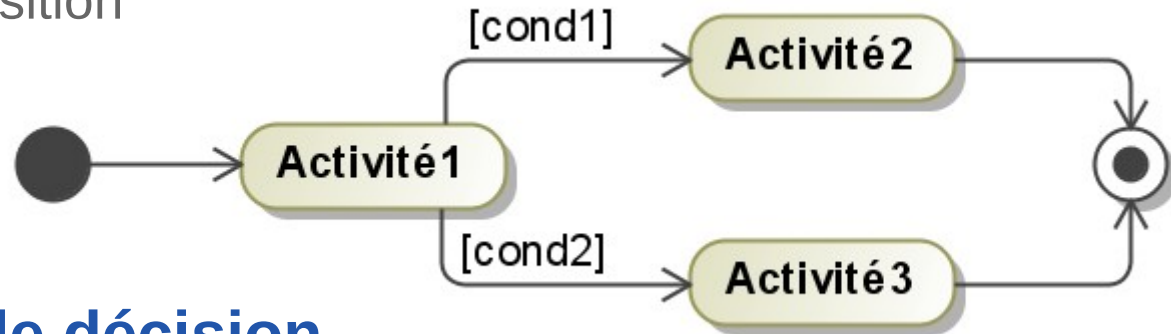
- **La documentation textuelle des UseCases est indispensable**
 - Pour communiquer avec les utilisateurs et se mettre d'accord sur les termes
- **Mais le texte est trop linéaire**
 - Difficile d'exprimer les cas exceptionnels et les cas d'erreurs
- **Il est recommandé de compléter par des diagrammes dynamiques**
 - Tels que les diagrammes de séquences
 - Ou mieux, le diagramme d'activités, assez bien compris des utilisateurs

- **Pour modéliser les flots de contrôle ou de données**
 - Permet de représenter le comportement d'une méthode ou le déroulement d'un cas d'utilisation
- **Représenté sous forme de graphe (proche du diagramme d'états)**
 - Les nœuds représentent des actions ou des activités
 - Les transitions expriment le passage d'une activité à une autre
 - Un nœud initial indique le début du flot
 - Un nœud final (éventuellement plusieurs) indique sa fin
- **Une activité peut éventuellement être structurée**
 - Être elle-même décrite par un diagramme d'activité



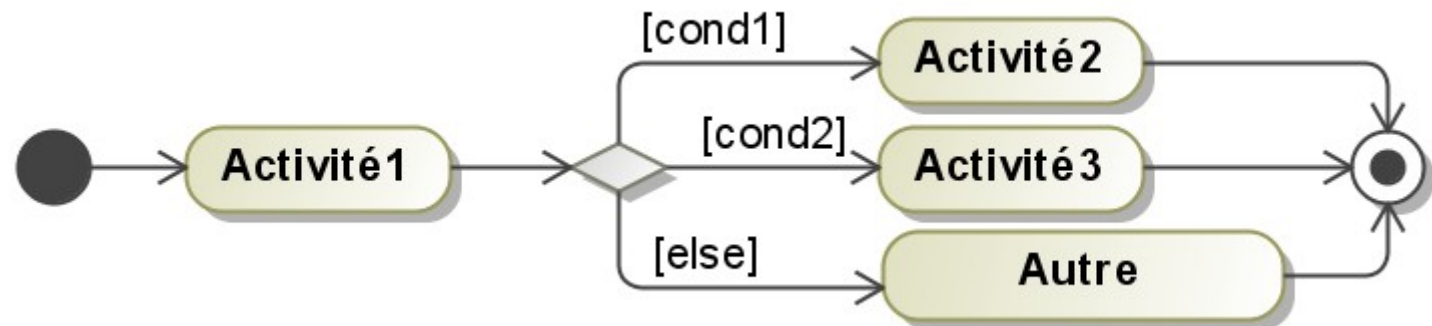
- On retrouve les transitions avec « garde »

- Une condition sur la transition



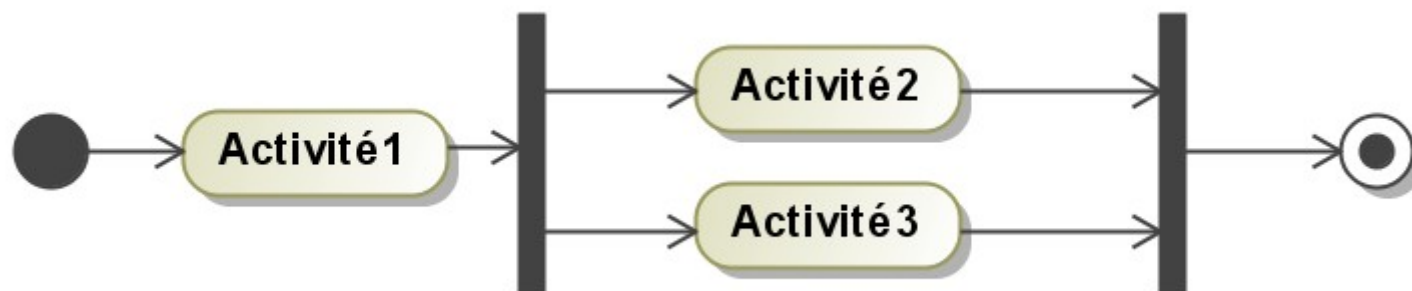
- Ainsi que les nœuds de décision

- Peuvent éventuellement être sécurisés par une garde [else]
 - Valide si aucune autre garde n'est valide
- Peuvent aussi servir de nœud de fusion (recevoir plusieurs transitions)



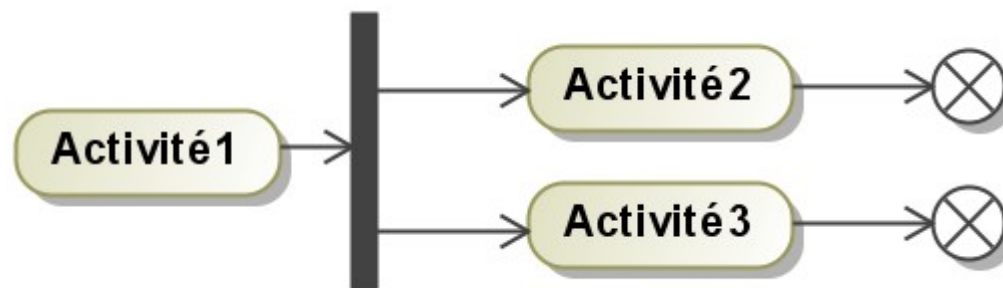
- **Représentées par des chemins concurrents**

- Utilisation de nœud de bifurcation / union



- **Nœud de terminaison de flot**

- L'activité continue pour les chemins concurrents



- **Les nœuds « exécutables »**

- Des actions simples (création d'objet, appel d'opération, levé d'exception, affectation de variable)
- Permettent une description très précise



- **Représentation spéciale**

- Envoi d'un signal
- Réception d'un signal (événement)
- Événement de temps (peut être relatif)



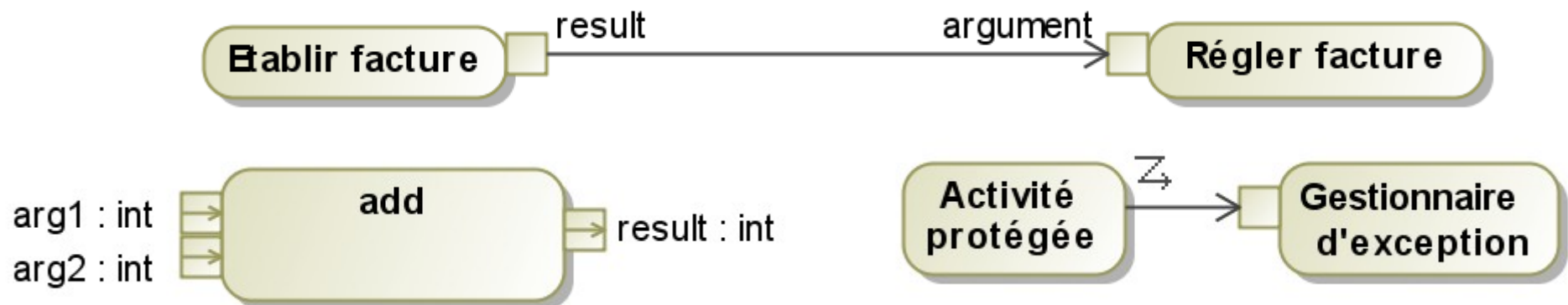
- **Nœud d'objet**

- Représente un objet généré par une action et utilisé par une autre
- Peut être nommé et typé

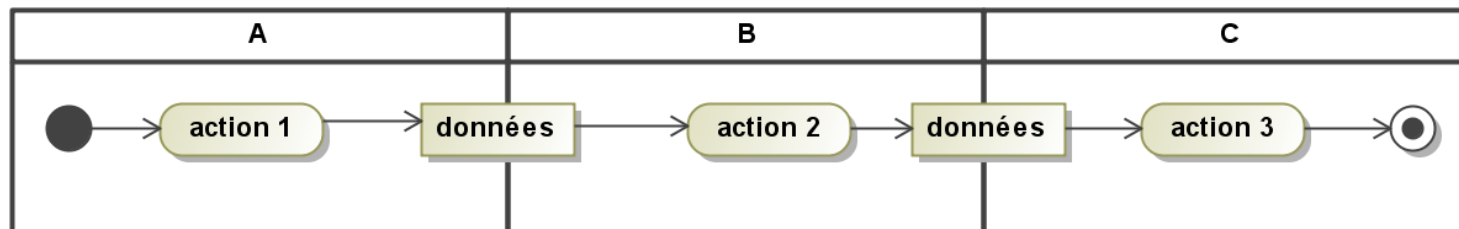


- **On peut aussi utiliser les « pin » d'entrée et de sortie**

- Permet d'exprimer précisément les entrées et sorties
- Permet d'exprimer la gestion des exceptions



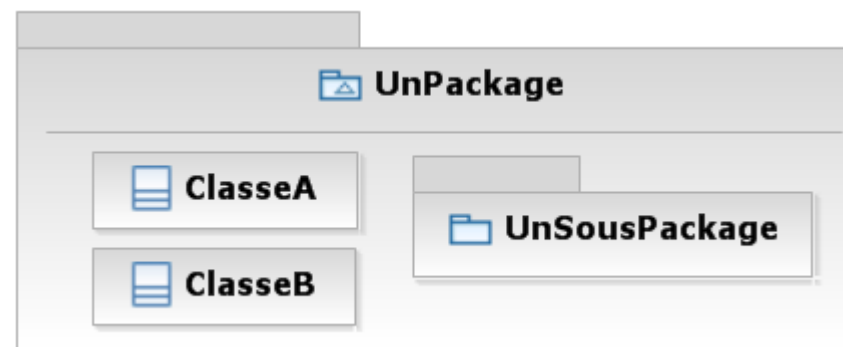
- **Permet de regrouper arbitrairement les activités**
 - Par exemple pour montrer le responsable de leur implémentation
 - (classe, acteur, ...) ou même décrire une organisation
 - Une activité ne peut appartenir qu'à un seul couloir
- **Peut même être multi-dimensionnel**



Les autres diagrammes

- **Les éléments peuvent être organisés dans des packages**

- Un package peut contenir des classes, des diagrammes, d'autres packages, ...
- Comme des répertoires de fichiers
- Les éléments peuvent être privés



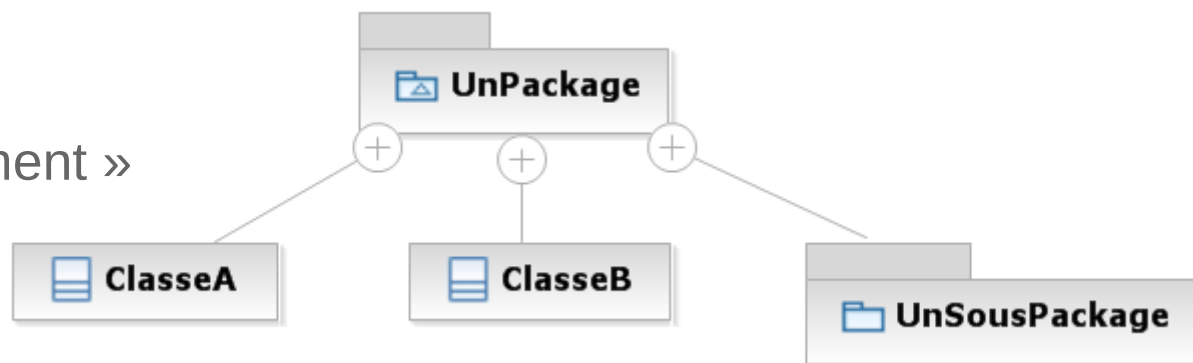
- **Constitue un espace de noms**

- Séparateur « :: »
- Nom qualifié = chemin complet



- **Autre représentation**

- Avec des « liens de confinement »



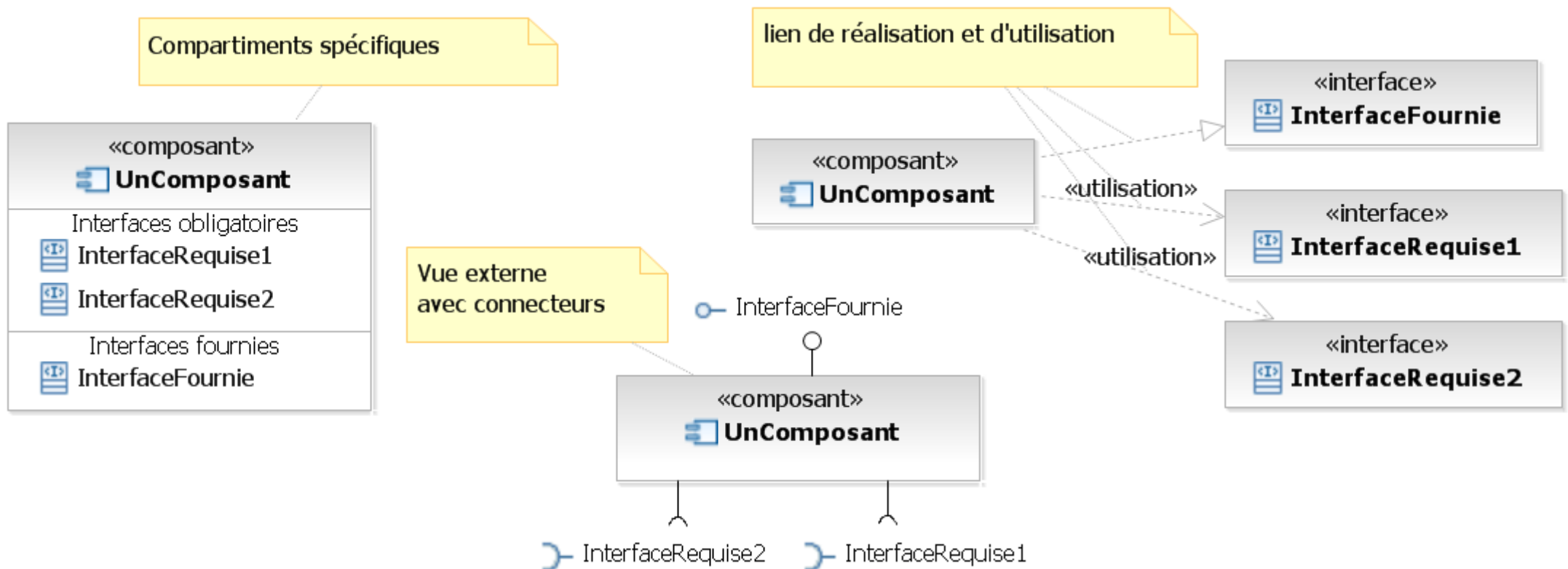
- **Ce type de diagramme permet d'offrir une vue d'ensemble**
 - En montrant l'architecture globale du système
 - En montrant les liens entre ses différents composants
 - Historiquement dédié à la représentation physique du système, son utilisation s'est progressivement élargie
- **Composant**
 - Représenté par un classificateur avec le stéréotype « composant »
 - Le diagramme permet de montrer les dépendances entre composants



- **Un composant fournit et requiert des interfaces**

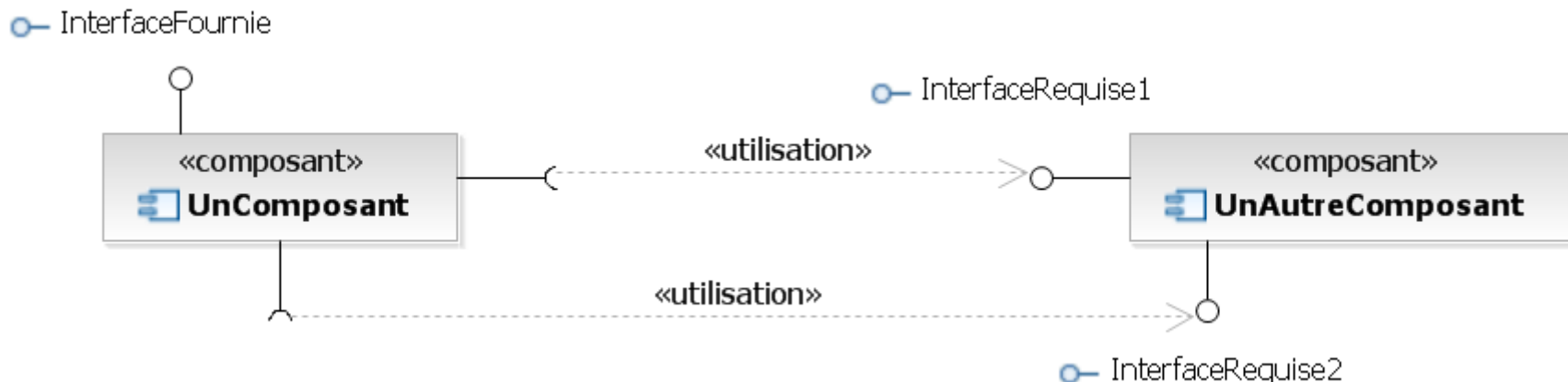
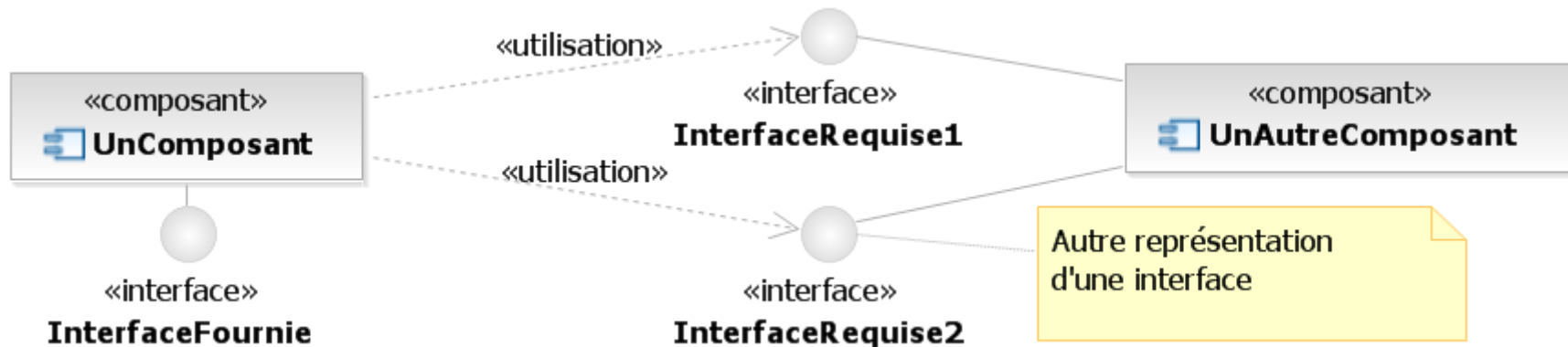
- Il fournit les interfaces qu'il « réalise »
- Il requiert les interfaces qu'il « utilise »
- Remarque : RSA traduit « requises » par « obligatoires »

- **UML propose trois notations pour exprimer cela**



- **Le diagramme de composants pour montrer la « connectique »**

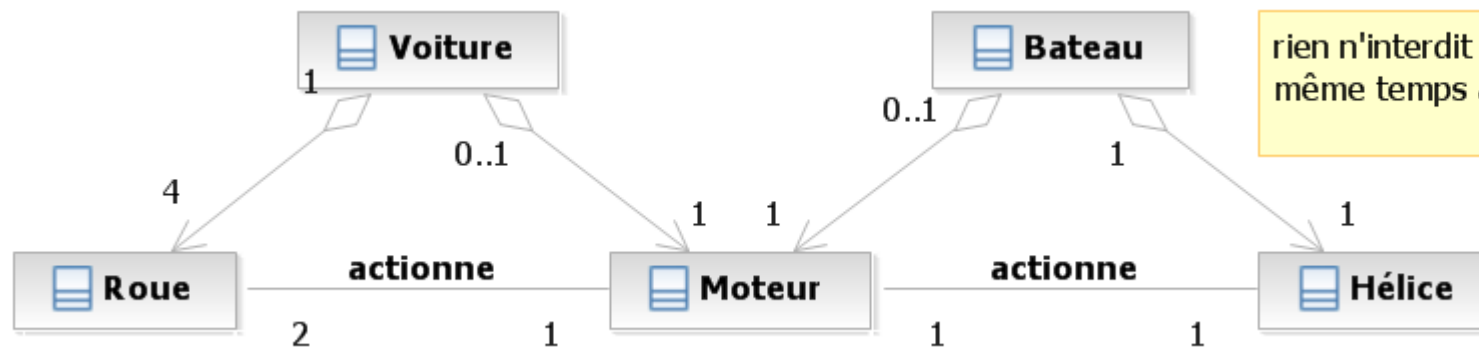
- Ici, **UnAutreComposant** fournit les interfaces requises de **UnComposant**
- On montre comment les deux composants sont connectés ensemble
- L'utilisation des « vue externe » permet un schéma plus compact



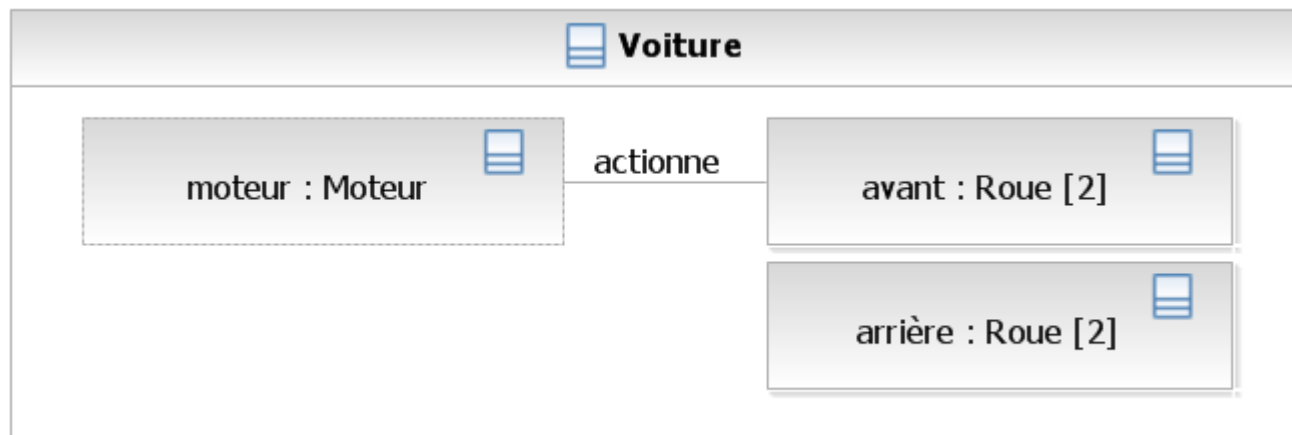
- **Un composant est réalisé de différentes manières**
 - Par d'autres composants ou classes
 - Par une librairie externe, un fichier, ...
- **UML 2 : un rapprochement entre « classe » et « composant »**
 - Correspondent généralement à des niveaux d'abstractions différents
 - Se traduit aussi dans les nouvelles manières de coder depuis les spécifications Java EE puis et les travaux autour du framework Spring
- **Nécessité de spécifier plus finement les structures internes**
 - UML 2 a introduit les diagrammes de « structure composite »
 - La même notation est utilisable dans le compartiment « structure » d'un élément

- **Structure interne de classeur complexe (classe ou composant)**

- Permet de mieux isoler le contenu et le fonctionnement interne
- Un « part » (participant) identifie un élément dans le contexte d'une instance
- Peut permettre de lever des ambiguïtés sur des relations complexes

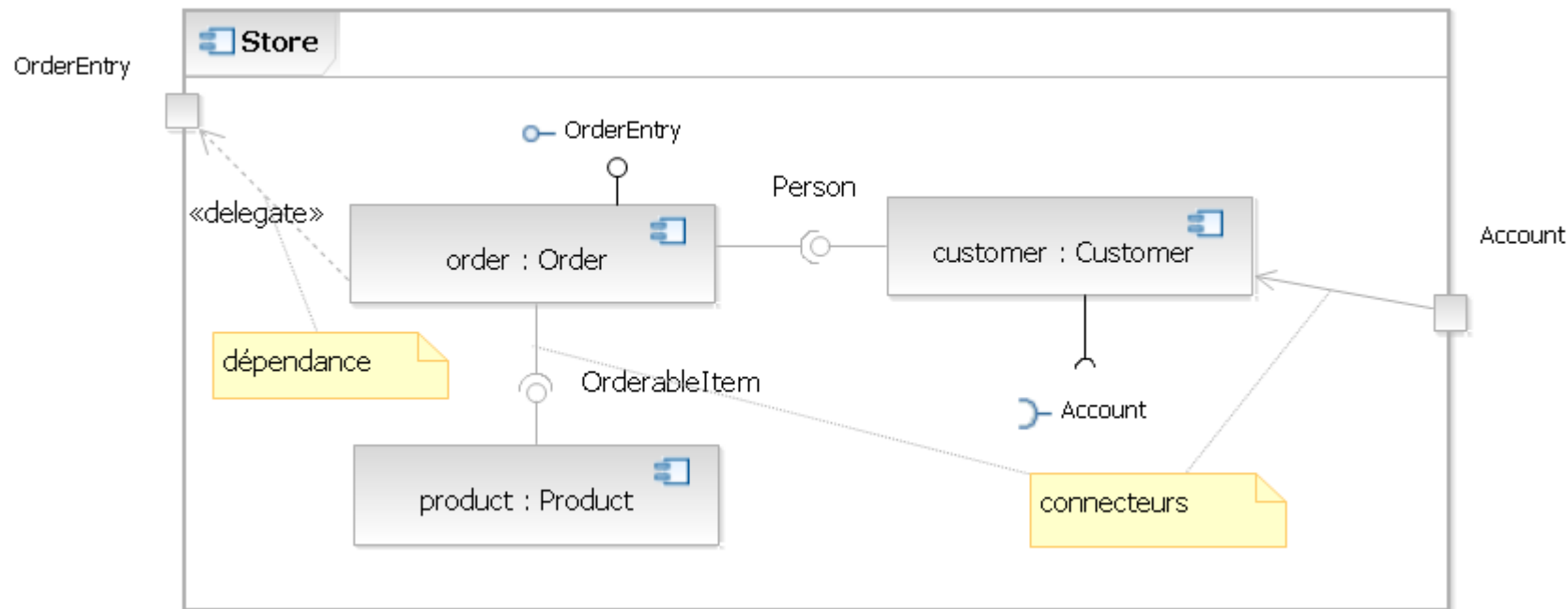


rien n'interdit ici qu'un moteur soit lié en même temps à une voiture et un bateau

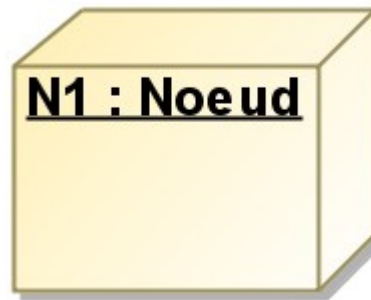


la structure montre les parties internes et leur collaboration à l'aide de connecteurs

- **Port = point de connexion entre le classeur et son environnement**
 - Généralement associé à une interface requise ou offerte
 - Pour montrer les connexions entre les éléments internes et externes
 - Facilite les évolution internes sans impact à l'extérieur
 - Renforce l'isolation (élimine les liens entre intérieur et extérieur du classeur)
 - Permet la réutilisation dans un contexte différent
 - On l'associe à un élément par une dépendance « delegate » ou un connecteur



- **Sert à décrire les ressources matérielles du système**
 - Montre la répartition des composants sur ces matériels
- **Les ressources sont représentées par un Nœud**
 - C'est un classeur : peut posséder des attributs (ses caractéristiques)
 - Peut exister sous forme de spécification ou d'instance

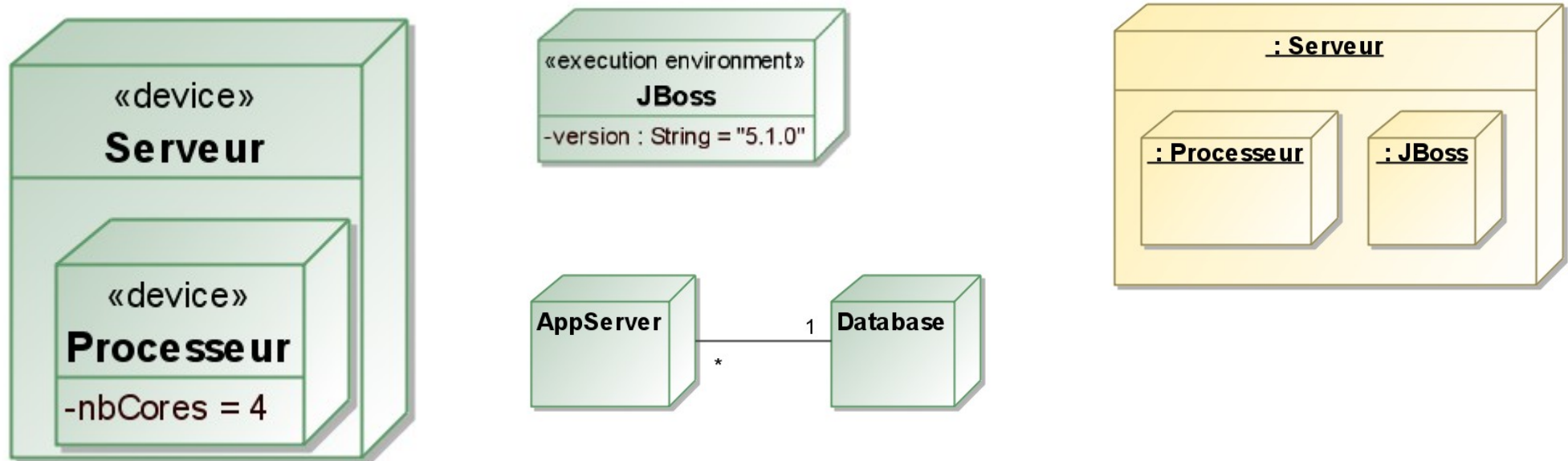


- **Deux variantes**

- « device » : un matériel ou un périphérique
- « execution environment » : un logiciel (OS, conteneur, ...)

- **Pour structurer le système**

- Ces éléments peuvent être imbriqués
- Ils peuvent être connectés pour montrer les communications



- **Un artefact correspond à un élément concret**

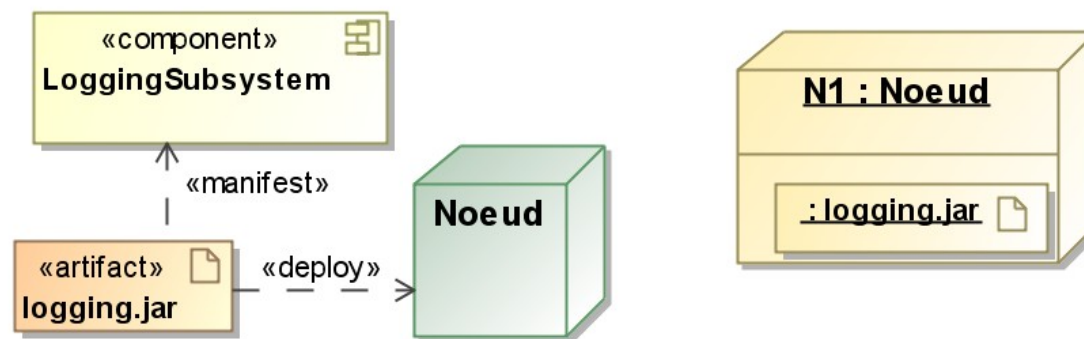
- Document, fichier, exécutable, table d'une base, ...
- Il sert à « manifester » des éléments du modèle
 - C'est-à-dire réaliser et implémenter



- **Il peut être « déployé » sur un nœud**

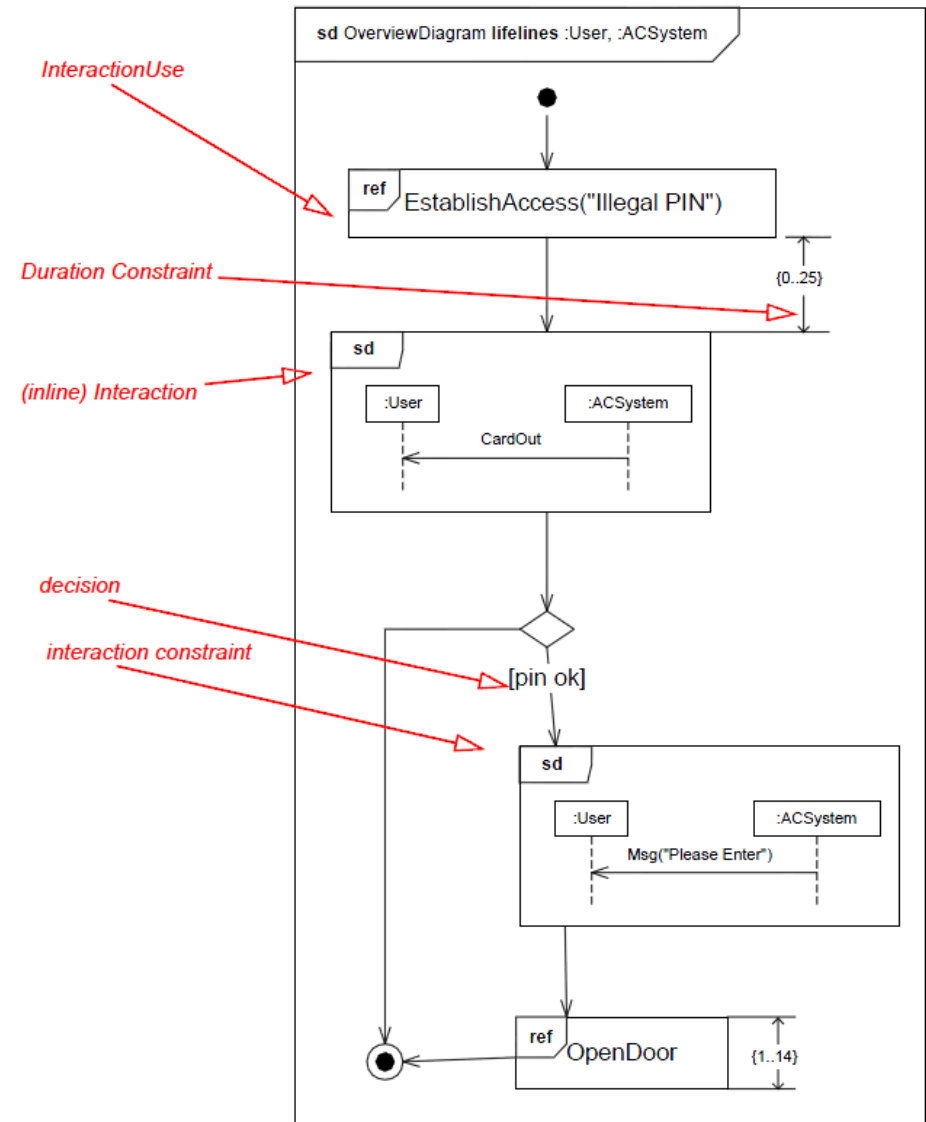
- **Ses instances se déploient sur une instance de nœud**

- On peut représenter ça par un lien « deploy » ou en imbriquant

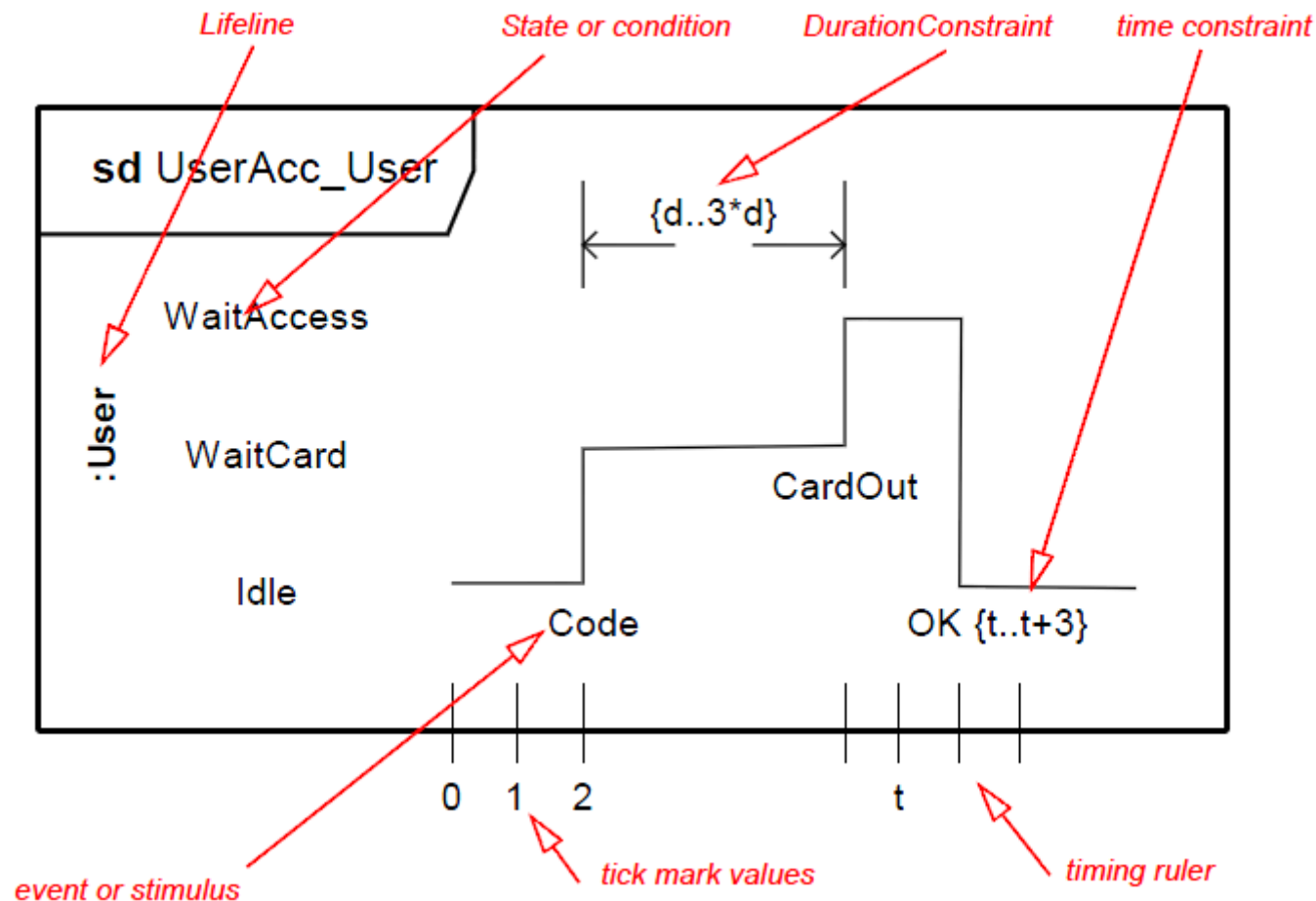


• Une variante du diagramme d'activités

- Permet d'inclure des diagrammes de séquence comme nœud d'activité
 - Soit sous forme de référence
 - Ou directement dans le diagramme
- Exemple extrait des spécifications

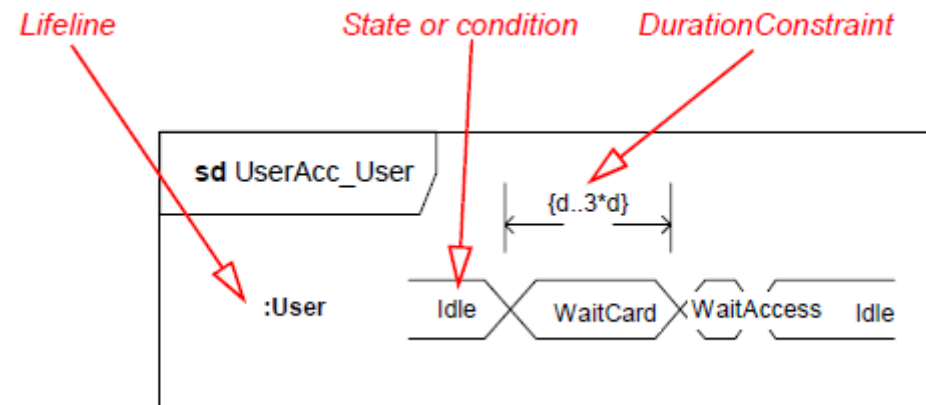


- Décrire précisément les contraintes de temps
 - Très spécifique, pour les systèmes temps-réel par exemple



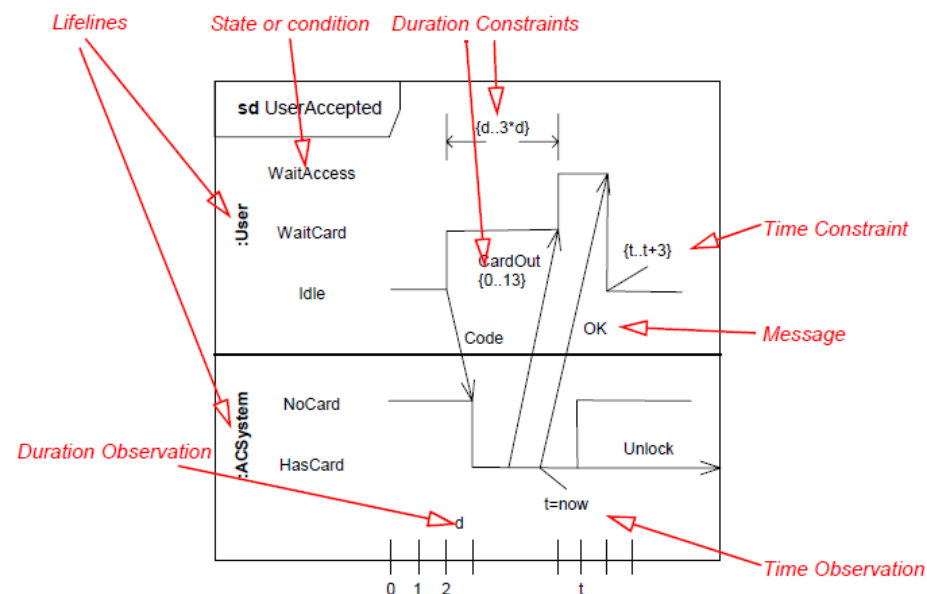
• Présentation compacte

- Montre les changements d'état



• Lignes de vie multiples

- Et messages échangés



- **Fonctionnel**

- Diagramme de Cas d'Utilisation (identifier acteurs et exigences)

- **Statique**

- Diagramme de classe (structure pour analyse et conception)
 - Diagramme de package (organiser, découper)
 - Diagramme d'objet (montrer des exemples, cas particuliers ou contre-exemple)
 - Diagramme de composant (structure pour analyse et conception du système)
 - Diagramme composite (montrer l'intérieur d'un composant ou classe complexe)
 - Diagramme de déploiement (montrer l'architecture de déploiement)

- **Dynamique**

- Diagramme de séquence (pour expliciter un cas d'utilisation ou une méthode)
- Diagramme de communication (autre forme du diagramme de séquence)
- Diagramme d'activité (expliciter des flots de contrôle ou de données)
- Diagramme interaction overview (une combinaison de activités et séquence)
- Diagramme state machine (montrer les contraintes dynamique d'une classe)
- Diagramme de timing (montrer les contraintes de temps)

- **BPMN (Business Process Model and Notation)**

- Pour modéliser les procédures d'entreprise et les processus métier
- Dans l'esprit du diagramme d'activités mais beaucoup plus riche

- **SysML (Systems Modeling Language)**

- Une extension d'un sous ensemble d'UML
- Il simplifie la notation ajoute des diagrammes (mais en supprime d'autres)
- Pour la modélisation de systèmes au sens large (moins centrés sur le logiciel)

Conclusion

- **Un panorama de la notation**
 - Une notation très riche
 - Difficilement maîtrisable complètement
 - Ne pas chercher à tout utiliser (maîtriser les quelques diagrammes principaux)
- **Seulement une notation**
 - Pas de préconisation méthodologique (se tourner vers UP)
- **Intimement liée aux activités d'analyse et de conception**
 - Bien plus complexes
 - Nécessitant la connaissance de la notation pour communiquer efficacement