GÉNIE LOGICIEL
CM2 -
QUALITÉ LOGICIELLE
DIAGRAMME DE CLASSES

L2
A. Prigent
2020-2021

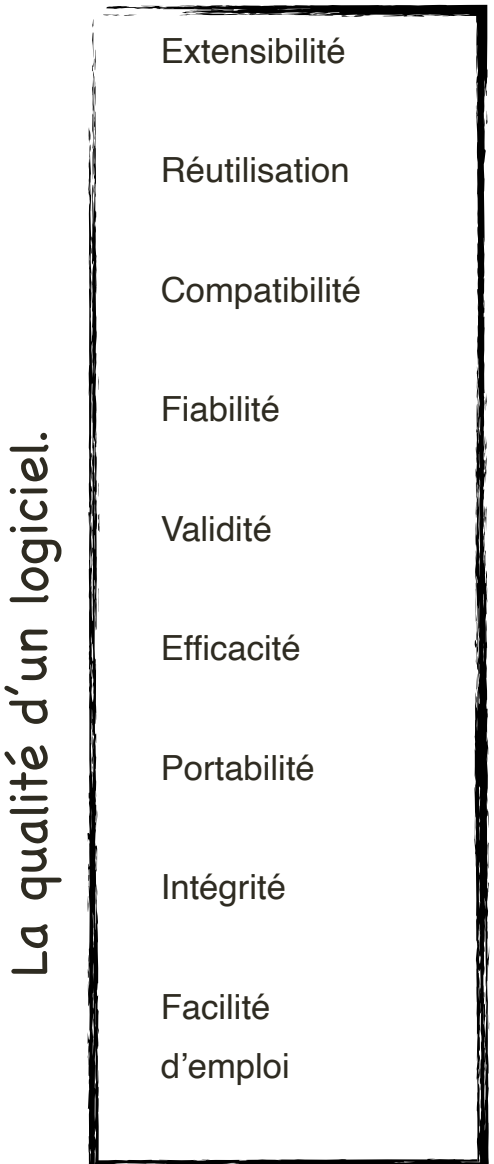
Laboratoire Informatique Image Interaction (L3I)
Université de La Rochelle - Pôle Sciences et Technologie - Avenue Michel Crépeau - 17042 LA ROCHELLE CEDEX 1 France
Tél : +33 (0)5 46 45 82 62 – Fax : 05.46.45.82.42 – Site internet : http://l3i.univ-larochelle.fr/

PARTIE 1

La qualité du logiciel

Laboratoire Informatique Image Interaction (L3I)
Université de La Rochelle - Pôle Sciences et Technologie - Avenue Michel Crépeau - 17042 LA ROCHELLE CEDEX 1 France
Tél : +33 (0)5 46 45 82 62 – Fax : 05.46.45.82.42 – Site internet : http://l3i.univ-larochelle.fr/

LA QUALITÉ DU LOGICIEL



- Qualité de logiciel
  - Les facteurs de qualité :
    - Internes ou de conception : visibles par les développeurs
    - Externes: visibles par les utilisateurs.

- **Problème**
  - Développer des logiciels de qualité à un coût acceptable
- **Solution**
  - Méthode de construction de logiciel *modulaire*
  - *Méthodes de validation*
  - Conception et programmation
    - Programmation par objet:
      - méthodologie de conception de logiciel qui répond bien aux exigences de qualité

LA QUALITÉ DU LOGICIEL (EXTERNE)

La qualité d'un logiciel.

- Extensibilité
- Réutilisation
- Compatibilité
- Fiabilité
- Validité
- Efficacité
- Portabilité
- Intégrité
- Facilité d'emploi

Extensibilité : facilité avec laquelle un logiciel se prête à une modification ou à une extension des fonctions qui lui sont demandées

- Facilité avec laquelle le logiciel peut s'adapter à des changements de spécification
  - Ou à des corrections !
- Pour une librairie de composants
  - Il est rare qu'un composant satisfasse nos besoins à 100%
    - P.ex: printf() affiche un chaîne de caractères; mais s'il nous faut une fonction qui affiche des chaînes et des entiers ?
      - Peut-on aisément étendre la fonction ?

LA QUALITÉ DU LOGICIEL (EXTERNE)

La qualité d'un logiciel.

- Extensibilité
- Réutilisation
- Compatibilité
- Fiabilité
- Validité
- Efficacité
- Portabilité
- Intégrité
- Facilité d'emploi

Réutilisation : aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications

- Possibilité d'utiliser tout ou partie d'un logiciel pour le développement de nouvelles applications.
  - P.ex: les librairies Java
  - Nécessaire pour palier le coût de développement
    - Le cas idéal : le système est entièrement fait de composants existants
  - Réutiliser la connaissance ou l'expérience d'un autre programmeur
    - On n'a pas besoin de comprendre comment le printf() marche pour l'utiliser

LA QUALITÉ DU LOGICIEL (EXTERNE)

La qualité d'un logiciel.

- Extensibilité
- Réutilisation
- Compatibilité
- Fiabilité
- Validité
- Efficacité
- Portabilité
- Intégrité
- Facilité d'emploi

Compatibilité : facilité avec laquelle un logiciel peut être combiné avec d'autres logiciels.

- Composants logiciels sont développés de manière indépendante (vendeur, programmeur)
  - Intégration de composants
  - La partie du développement la plus difficile
- Accord au niveau de l'interface de composants
  - Si un composant possède une fonction printf(), tous les autres composants doivent connaître ce nom pour l'utiliser
- Accord au niveau de formats de données
  - P.ex: la date : 11/3/04 ou 3.11.04

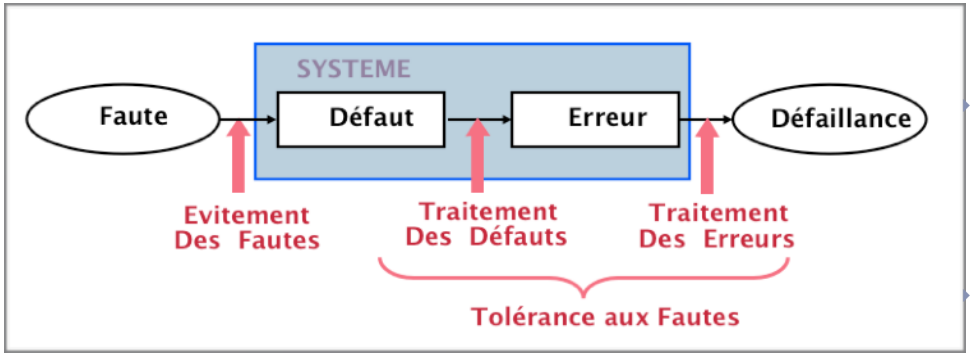
LA QUALITÉ DU LOGICIEL (EXTERNE)

Fiabilité (ou robustesse) : aptitude d'un produit logiciel à fonctionner dans des conditions anormales

La qualité d'un logiciel.

- Extensibilité
- Réutilisation
- Compatibilité
- Fiabilité
- Validité
- Efficacité
- Portabilité
- Intégrité
- Facilité d'emploi

- L 'application doit fonctionner comme prévu
  - Erreurs de conception et/ou programmation
    - Coût en vies humaines
      - Centrales nucléaires, avions
    - Coût financier
      - Banques, commerce électronique
- Aptitude à fonctionner même sous des conditions anormales
  - Cas non spécifiés dans l'analyse des besoins :
    - Ressource non disponible « fichier non-existant », etc.
    - Valeur inattendue, p.ex: division par zéro, date=« 29/**2**/2002 »
  - Les pannes du réseau ou des machines
    - « serveur ne répond pas », etc.
      - Le système doit pouvoir s'adapter et continuer



Définition

- Propriété d'un système qui permet de placer une **confiance justifiée** dans le **service** qu'il délivre

Processus de gestion de défaillance

LA QUALITÉ DU LOGICIEL (EXTERNE)

La qualité d'un logiciel.

- Extensibilité
- Réutilisation
- Compatibilité
- Fiabilité
- Validité
- Efficacité
- Portabilité
- Intégrité
- Facilité d'emploi

Validité : aptitude d'un produit logiciel à remplir exactement ses fonctions, définies par le cahier des charges et les spécifications

- Une application doit respecter les exigences des clients
  - Analyse: comprendre les besoins des clients
  - Faire une application qui marche bien mais qui ne correspond pas aux attentes du client ne sert à rien.

LA QUALITÉ DU LOGICIEL (EXTERNE)

La qualité d'un logiciel.

- Extensibilité
- Réutilisation
- Compatibilité
- Fiabilité
- Validité
- Efficacité
- Portabilité
- Intégrité
- Facilité d'emploi

Efficacité : Utilisation optimales des ressources matérielles.

- Bonne utilisation des différentes ressources
  - Place mémoire utilisée
    - Nombres de variables utilisées
  - Temps CPU utilisé (rapidité d'exécution)
    - Algorithmes de tri plus ou moins rapides
  - Utilisation de la bande passante
    - Échange de données non compressées

LA QUALITÉ DU LOGICIEL (EXTERNE)

La qualité d'un logiciel.

- Extensibilité
- Réutilisation
- Compatibilité
- Fiabilité
- Validité
- Efficacité
- Portabilité
- Intégrité
- Facilité d'emploi

Portabilité : facilité avec laquelle un logiciel peut être transférée sous différents environnements matériels et logiciels

- Facilité avec laquelle un logiciel peut être transféré vers plusieurs environnements
  - Dépendance vis-à-vis du système d'exploitation reste fréquente dans les programmes à cause des appels systèmes
- Idéalement …
  - … un programme doit tourner sur un PC, une station Sun
  - … ou sur un téléphone portable .
- Java
  - les programmes sont indépendants du système d'exploitation

LA QUALITÉ DU LOGICIEL (EXTERNE)

La qualité d'un logiciel.

- Extensibilité
- Réutilisation
- Compatibilité
- Fiabilité
- Validité
- Efficacité
- Portabilité
- Intégrité
- Facilité d'emploi

Vérifiabilité : facilité de préparation des procédures de test

LA QUALITÉ DU LOGICIEL (EXTERNE)

La qualité d'un logiciel.

- Extensibilité
- Réutilisation
- Compatibilité
- Fiabilité
- Validité
- Efficacité
- Portabilité
- Intégrité
- Facilité d'emploi

Intégrité : aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisés

- L'aspect exécution
  - Protection des composants (données, documents, programmes) contre des accès et modifications non autorisées
  - La journalisation (*journalling*)
    - Garder une trace des appels de procédures et d'échange de messages
- L'aspect développement
  - Le droit d'auteur sur le logiciel
  - Le sabotage industriel

LA QUALITÉ DU LOGICIEL (EXTERNE)

La qualité d'un logiciel.

- Extensibilité
- Réutilisation
- Compatibilité
- Fiabilité
- Validité
- Efficacité
- Portabilité
- Intégrité
- Facilité d'emploi

Facilité d'emploi : aptitude d'un logiciel à permettre une utilisation simplifiée par les utilisateurs

- *User friendly*
  - dans le jargon d'informatique
  - Un vrai dialogue entre l'utilisateur et le système
    - Il faut que le système transmette à l'utilisateur suffisamment d'informations ...

… mais sans le surcharger !

LA CRISE DU LOGICIEL

Fin des années 1960

Les symptômes les plus caractéristiques de cette crise sont :

- Les logiciels réalisés ne correspondent souvent pas aux besoins des utilisateurs
- Les logiciels contiennent trop d'erreurs (qualité du logiciel insuffisante)
- Les coûts de développement sont rarement prévisibles et sont généralement prohibitifs
- La maintenance des logiciels est une tâche complexe et coûteuse
- Les délais de réalisation sont généralement dépassés
- Les logiciels sont rarement portables

La crise du logiciel

PRISE DE CONSCIENCE

Comparer l'évolution du coût du logiciel et du matériel :

Le coût du matériel baisse régulièrement;

Le matériel devient de plus en plus puissant, conduisant à la réalisation de logiciels de plus en plus complexes et donc coûteux

Le coût des logiciels représenterait actuellement 85 % des dépenses totales des systèmes informatiques

NAISSANCE DU GÉNIE LOGICIEL

Est un domaine de recherche qui a été défini (fait rare) du 7 au 11 octobre 1968, à Garmisch-Partenkirchen, sous le parrainage de l'OTAN.

Le génie logiciel est la discipline née en réponse à la crise du logiciel.

Le génie logiciel se caractérise par une approche rigoureuse et systématique à la construction de logiciels ne pouvant être maîtrisés par une seule personne.

Le génie logiciel est donc l'art :

- De spécifier, de concevoir, de réaliser, et de faire évoluer des programmes, des documentations et des procédures de qualité
- Avec des moyens et dans des délais raisonnables, en vu d'utiliser un système informatique pour résoudre certains problèmes.

LE GÉNIE LOGICIEL

C'est le regroupement de "règles" en vue d'assurer un bon développement d'une application :

- Travail en équipe,
- Analyse du problème
- Techniques de programmation
- Gestion de versions
- Définition de normes (Langage, sécurité...)
- Définition des caractéristiques d'une "bonne approche"
- Définition de méthodes

ORGANISER LE DÉVELOPPEMENT

Equipe de développement :

- définition de rôles spécifiques (spécifieur, concepteur, développeur,)
- reconnaissance de qualifications
- formation complémentaire
- introduction d'un plan qualité : mise en place de procédures très strictes de contrôles

Introduire des cycles de vie :

- choix de méthodes rigoureuses
- choix de notations spécialisées selon la nature des problèmes
- utilisation d'environnements riches et d'outils supports

PARTIE 2

Les différents cycles de vie

Laboratoire Informatique Image Interaction (L3I)
Université de La Rochelle - Pôle Sciences et Technologie - Avenue Michel Crépeau - 17042 LA ROCHELLE CEDEX 1 France
Tél : +33 (0)5 46 45 82 62 – Fax : 05.46.45.82.42 – Site internet : http://l3i.univ-larochelle.fr/

MODÈLES DE CYCLE DE VIE

Définition :

- Modèles généraux de développement décrivant les enchaînements et les interactions entre activités.

Objectif :

- Obtenir des processus de développement rationnels, reproductibles et contrôlables. Utilisés pour mieux maîtriser le processus développement en permettant de prendre en compte, en plus des aspects techniques, l'organisation et les aspects humains

Remarques :

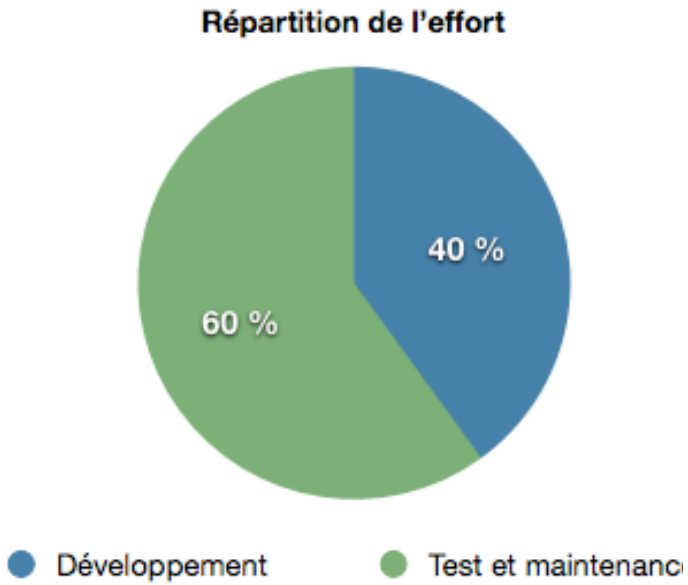
- Une étape, telle que la conception, peut faire intervenir plusieurs activités, comme celles de la spécification globale, du prototypage et de la validation

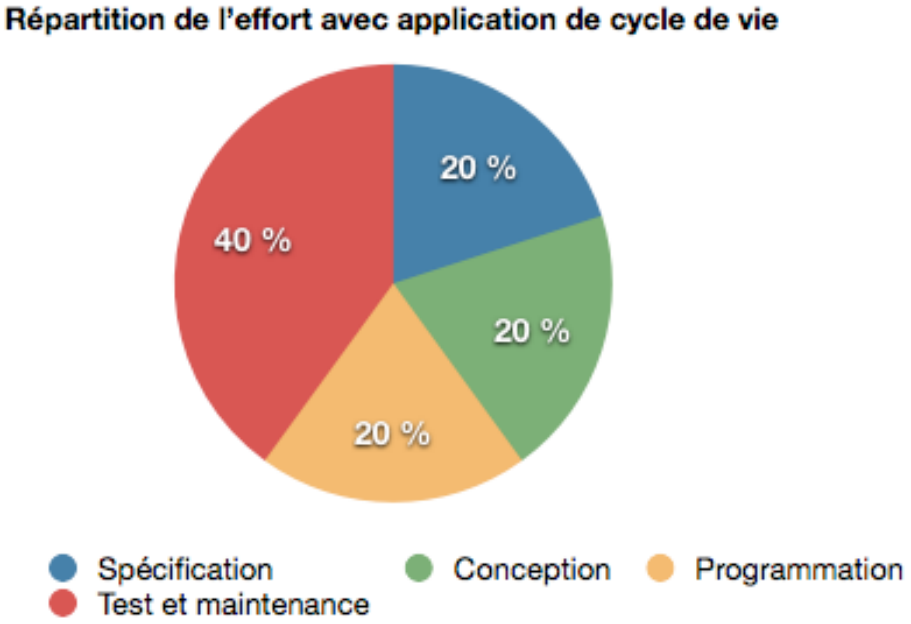
- une activité comme la documentation peut se dérouler pendant plusieurs étapes

CYCLE DE VIE DU LOGICIEL

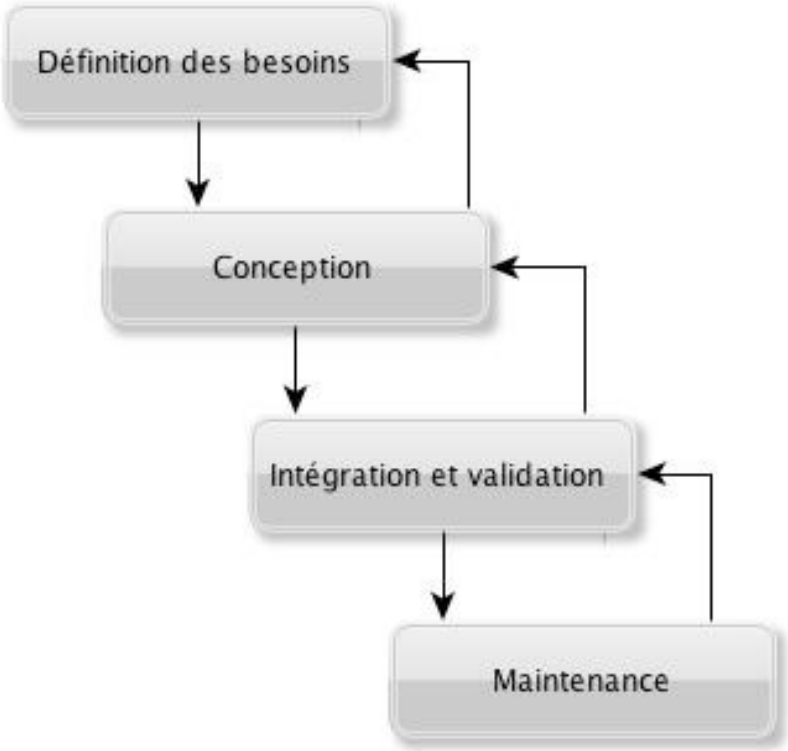
Dans un projet « bien conduit », l'effort se répartit comme suit :

- 15 à 20% programmation,
- 40% spécification et conception,





MODÈLE EN CASCADE



Dans ce modèle, chaque **phase** se termine à une date précise par la production de certains documents ou logiciels.

On ne passe à la phase suivante que **si les résultats de l'étape précédente sont jugés satisfaisants**

D'autres activités interviennent, par exemple **le contrôle technique et la gestion de la configuration** sont présents tout au long du processus.

MODÈLE EN CASCADE

Modèle de la cascade (waterfall)

- Le modèle original ne comportait pas de possibilité de retour en arrière.
- Celle-ci a été rajoutée ultérieurement sur la base qu'une étape ne remet en cause que l'étape précédente, ce qui, dans la pratique, s'avère insuffisant.

Les développements récents de ce modèle font paraître de la validation-vérification à chaque étape :

- faisabilité et analyse des besoins : validation ;
- conception du produit et conception détaillée : vérification ;
- intégration : test d'intégration et test d'acceptation ;
- installation : test du système.

MODÈLE EN V

Contrairement au modèle de la cascade, ce modèle fait apparaître le fait que le **début du processus de développement conditionne ses dernières étapes.**

Toute description d'un composant est accompagnée de **tests** qui permettront de s'assurer qu'il correspond à sa description.

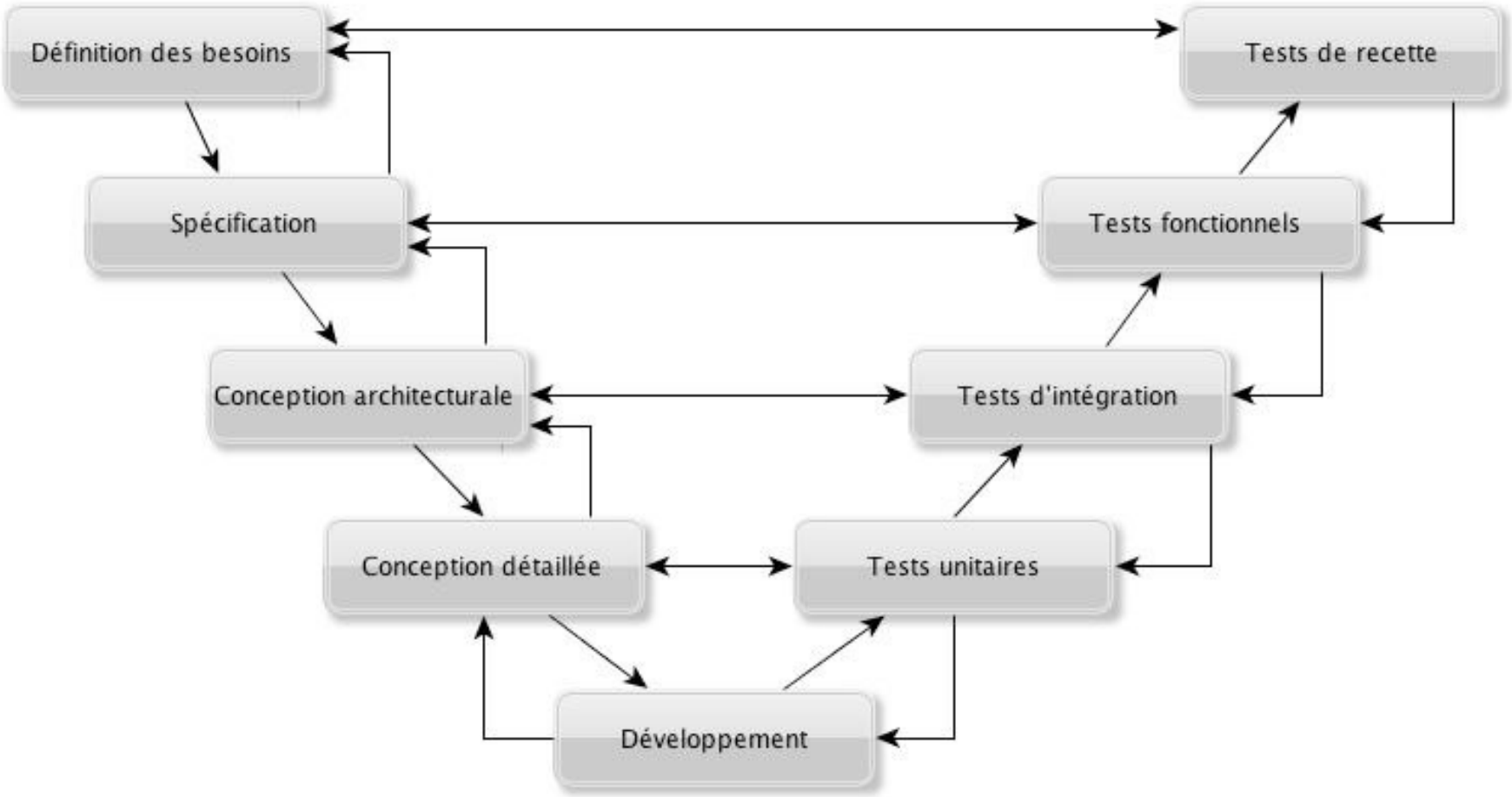
Ceci rend explicite la préparation des dernières phases (validation-vérification) par les premières (construction du logiciel),

Ce modèle permet ainsi d'éviter un écueil bien connu de la spécification du logiciel :

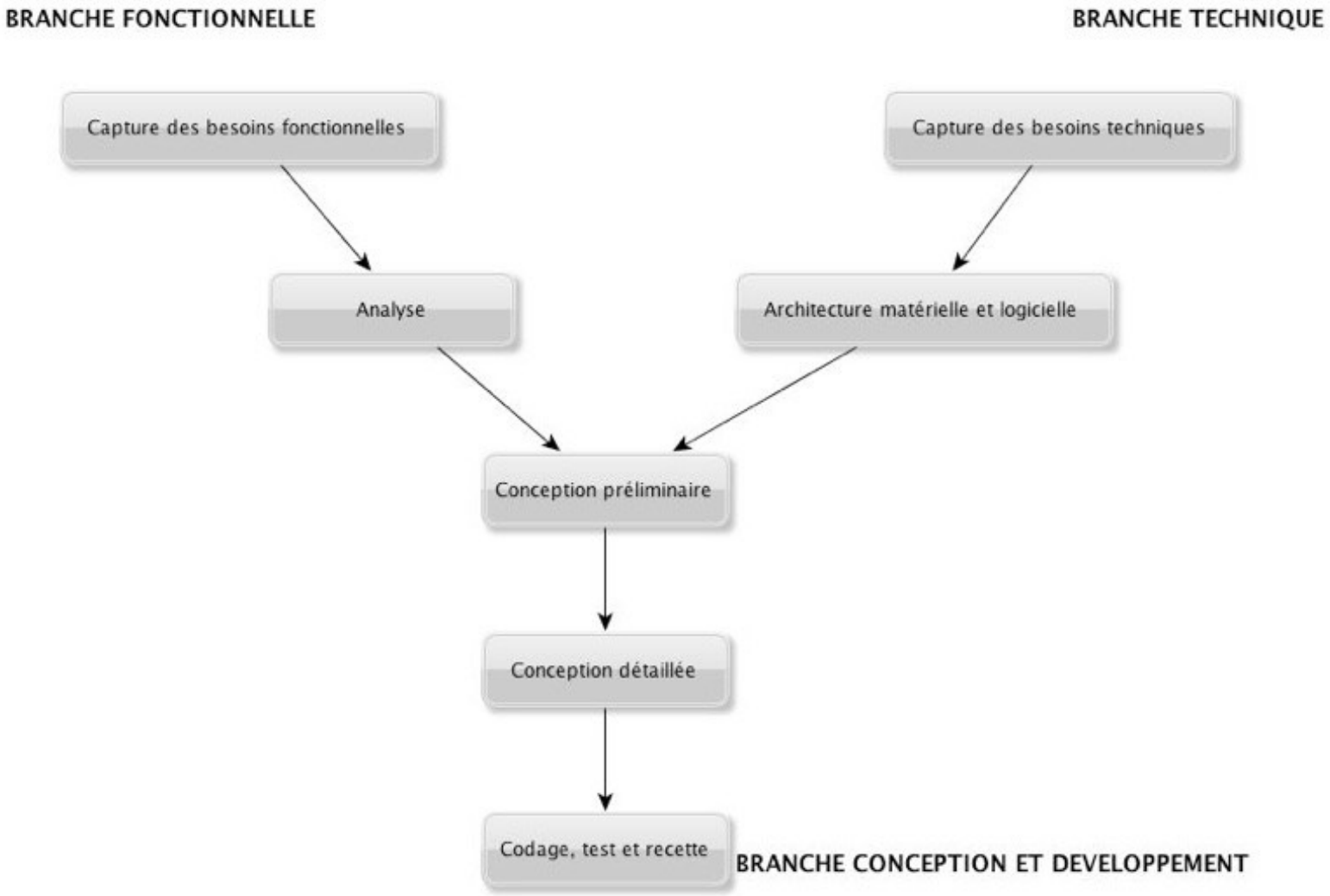
- énoncer une propriété qu'il est impossible de vérifier objectivement après la réalisation.

Le cycle en V est le cycle qui a été normalisé

MODÈLE EN V


Définition des besoins
Tests de recette
Spécification
Tests fonctionnels
Conception architecturale
Tests d'intégration
Conception détaillée
Tests unitaires
Développement

MODÈLE EN Y


BRANCHE FONCTIONNELLE
BRANCHE TECHNIQUE
Capture des besoins fonctionnelles
Capture des besoins techniques
Analyse
Architecture matérielle et logicielle
Conception préliminaire
Conception détaillée
Codage, test et recette
BRANCHE CONCEPTION ET DEVELOPPEMENT

MODÈLE PAR INCRÉMENT

Dans les modèles en V ou en cascade, un logiciel est décomposé en composants développés séparément et intégrés à la fin du processus

Dans le modèle par incréments, seul un sous ensemble est développé à la fois

Dans un premier temps un logiciel noyau est développé, puis successivement, les incréments sont développés et intégrés. Chaque incrément est développé selon l'un des modèles précédents.

MODÈLES PAR INCRÉMENT

Avantages

- Chaque développement est moins complexe ;
- Les intégrations sont progressives ;
- Possibilité de livraisons et de mises en service après chaque incrément ;
- Meilleur lissage du temps et de l'effort de développement à cause de la possibilité de recouvrement des différentes phases. l'effort est constant dans le temps par opposition au pic pour spécifications détaillées pour les modèles en cascade ou en V.

Risques

- La remise en cause du noyau de départ remet en cause celle des incréments précédents
- L'impossibilité d'intégrer un nouvel incrément

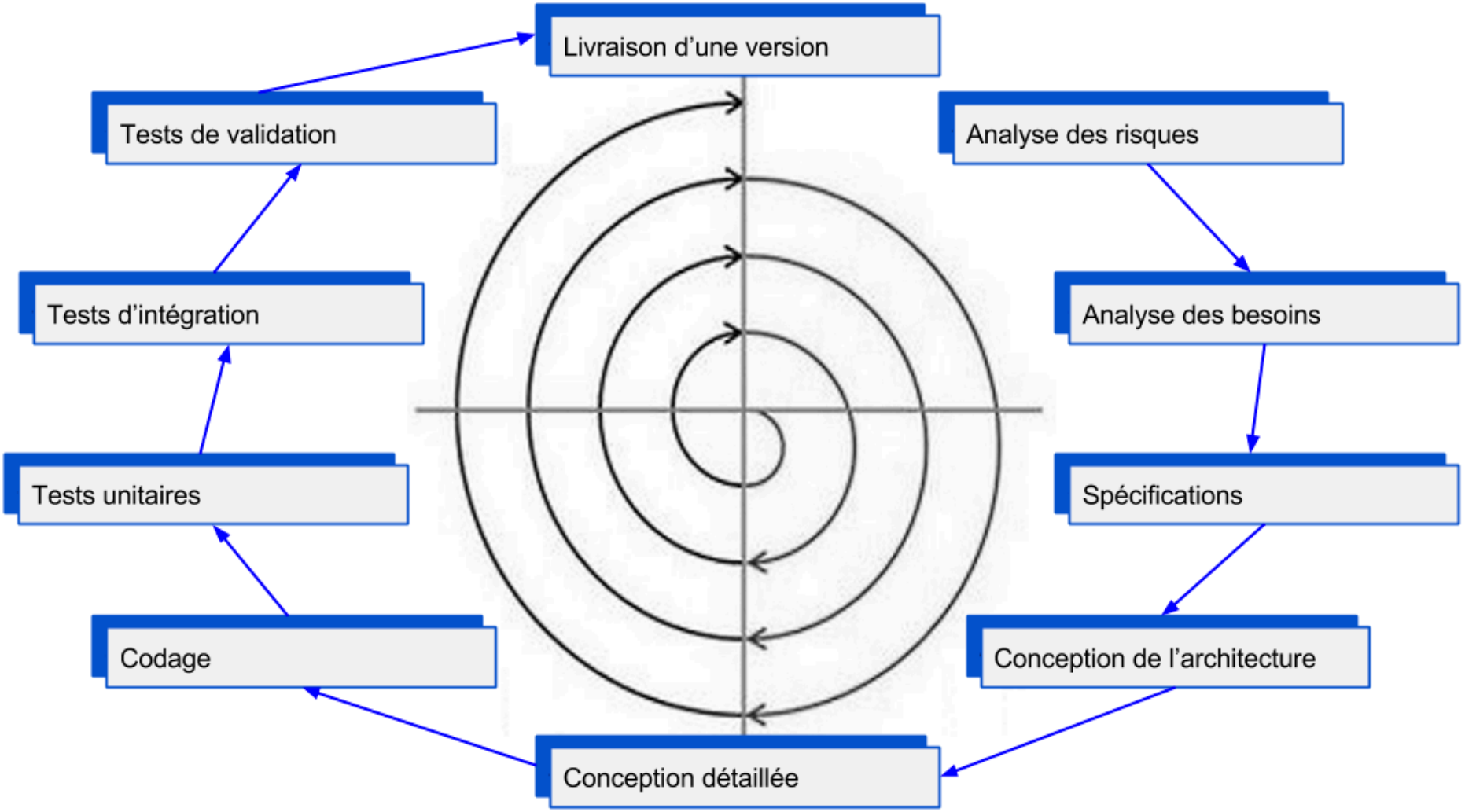
MODÈLE EN SPIRALE

Proposé par B. Boehm en 1988, ce modèle est beaucoup plus général que le précédent.

Il met l'accent sur l'activité d'analyse des risques : chaque cycle de la spirale se déroule en quatre phases :

- détermination, à partir des résultats des cycles précédents --ou de l'analyse préliminaire des besoins, des objectifs du cycle, des alternatives pour les atteindre et des contraintes ;
- analyse des risques, évaluation des alternatives et, éventuellement maquettage ;
- développement et vérification de la solution retenue, un modèle « classique » (cascade ou en V) peut être utilisé ici ;
- revue des résultats et vérification du cycle suivant.

LE CYCLE EN SPIRALE



ANALYSE DES RISQUES

La mise en œuvre demande des compétences managériales et devrait être limitée aux projets innovants à cause de l'importance que ce modèle accorde à l'analyse des risques. Citons, par exemple

- risques humains:
  - défaillance du personnel ; surestimation des compétences
  - travailleur solitaire, héroisme, manque de motivation
- risques processus
  - pas de gestion de projet
  - calendrier et budget irréalistes ;
  - calendrier abandonné sous la pression des clients
  - composants externes manquants ;
  - tâches externes défaillantes ;
  - insuffisance de données
  - validité des besoins ;
  - développement de fonctions inappropriées
  - développement d'interfaces utilisateurs inappropriées

- risques technologiques
  - produit miracle, "plaqué or";
  - changement de technologie en cours de route
  - problèmes de performance
  - exigences démesurées par rapport à la technologie
  - incompréhension des fondements de la technologie

MATURITÉ DU CYCLE DE VIE

Une notion de maturité du processus de développement a été définie en 1989 puis en 1991 par le SEI: Software Engineering Institude - DoD + Carnegie Mellon.

Une classification selon cinq niveaux a été ainsi définie. Le processus est dit:

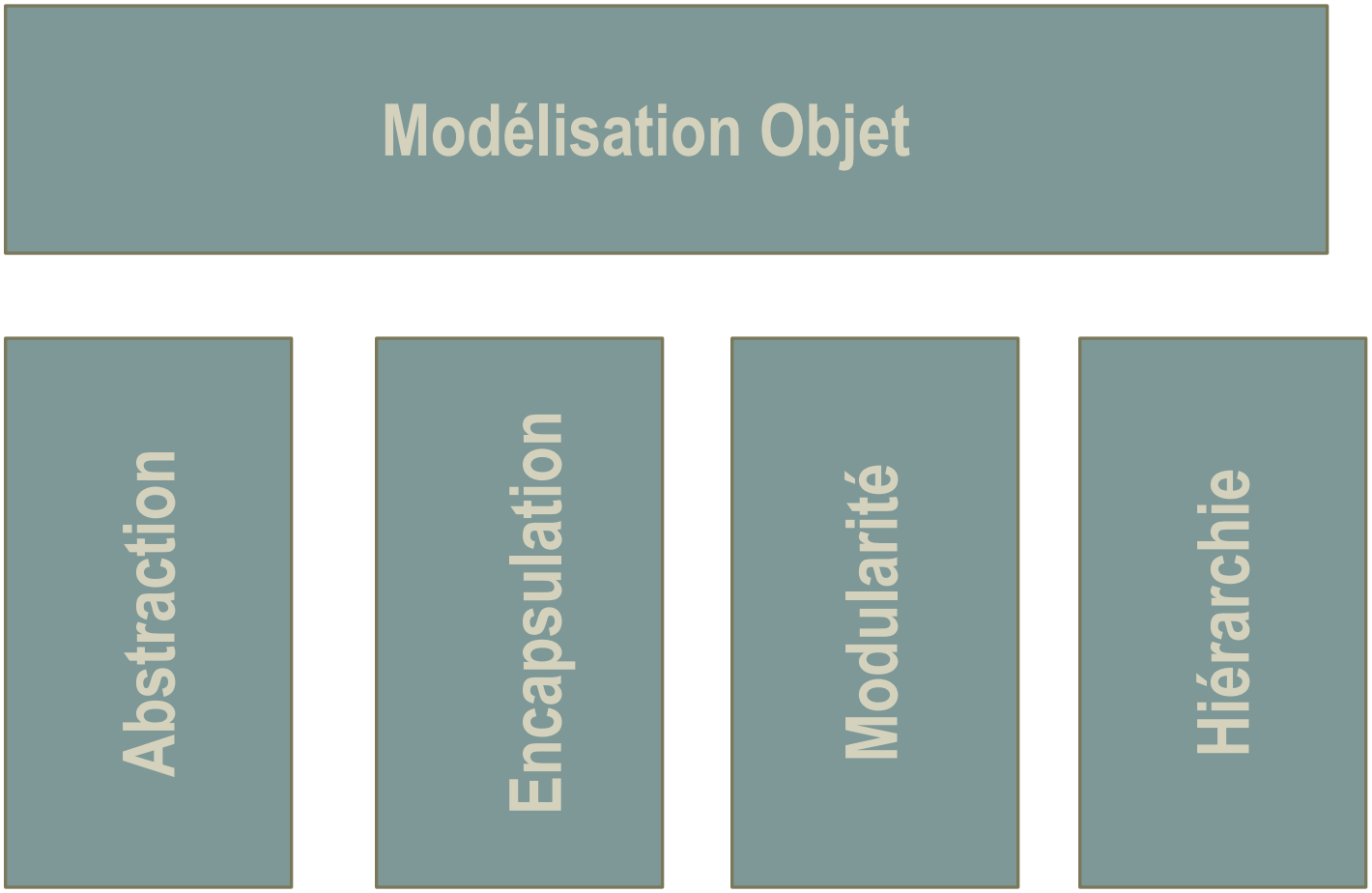
- **initial** si le développement est chaotique: les coûts, les délais et la qualité sont imprévisibles.
- **reproductible** si le processus de développement est artisanal et dépendant beaucoup des individus: les coûts et la qualité sont variables. Les méthodes sont mal définies ou mal suivies;
- **défini** si le processus de développement est bien suivi mais, pour l'essentiel de manière qualitative: les délais et les coûts sont fiables mais la qualité est variable.
- **géré** si le processus est contrôlé et mesuré. La qualité est fiable;
- **optimisé** si une analyse de chaque projet est effectuée à des fins de l'amélioration des coûts, des délais et de la qualité.

PARTIE 2

UML - le diagramme de classes

Laboratoire Informatique Image Interaction (L3I)
Université de La Rochelle - Pôle Sciences et Technologie - Avenue Michel Crépeau - 17042 LA ROCHELLE CEDEX 1 France
Tél : +33 (0)5 46 45 82 62 – Fax : 05.46.45.82.42 – Site internet : http://l3i.univ-larochelle.fr/

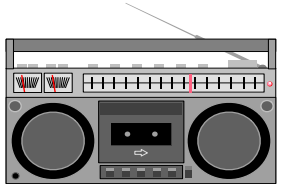
PRINCIPES DE LA MODÉLISATION OBJET



ABSTRACTION

Vendeur

Client

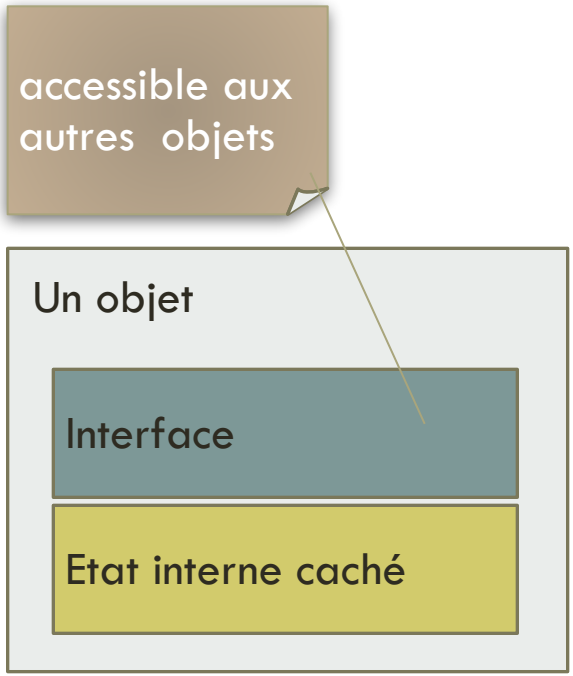


Produit

classe
type de données abstrait, caractérisé par des propriétés (attributs et méthodes) communes à toute une famille d'objets permettant de créer (instancier) des objets possédant ces propriétés.

ENCAPSULATION

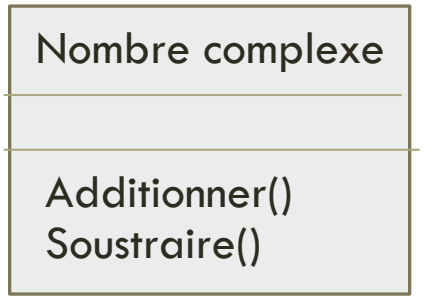
Réduit le couplage dans le modèle et favorise la modularité et la maintenance des logiciels



Masquer les détails d'implémentation d'un objet, en définissant une interface.

Interface : la vue externe d'un objet, elle définit les services accessibles aux utilisateurs de l'objet.

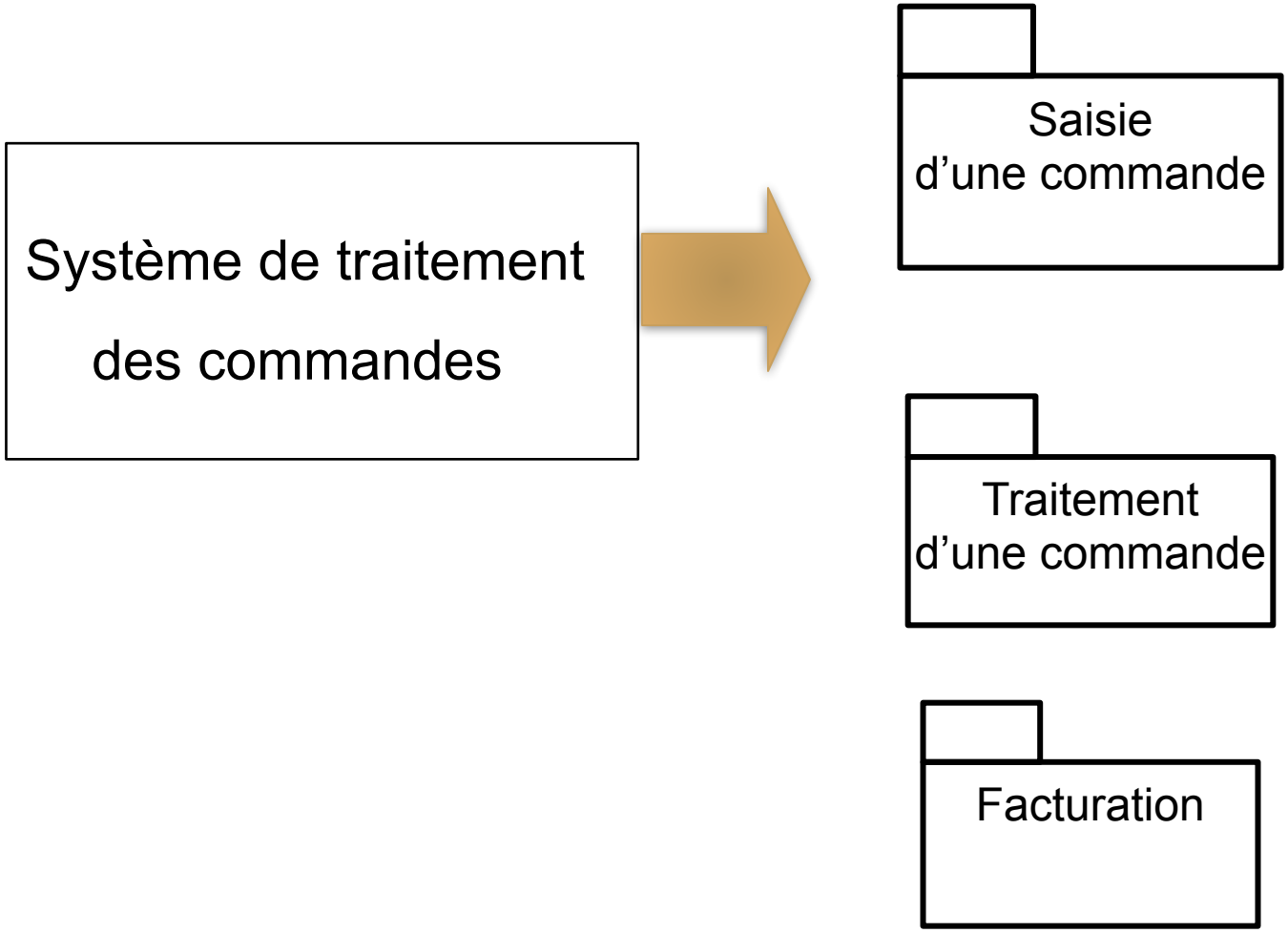
L'encapsulation:
facilite l'évolution d'une application (stabilisation de l'utilisation des objets) => modifier l'implémentation des attributs d'un objet sans modifier son interface
garantit l'intégrité des données, car elle permet d'interdire, ou de restreindre, l'accès direct aux attributs des objets.



La connaissance de la représentation interne du nombre complexe
cartésienne : $z = a + ib$
polaire : $z = \rho e^{i\theta}$
n'est pas nécessaire pour utiliser les opérations décrites

MODULARITÉ

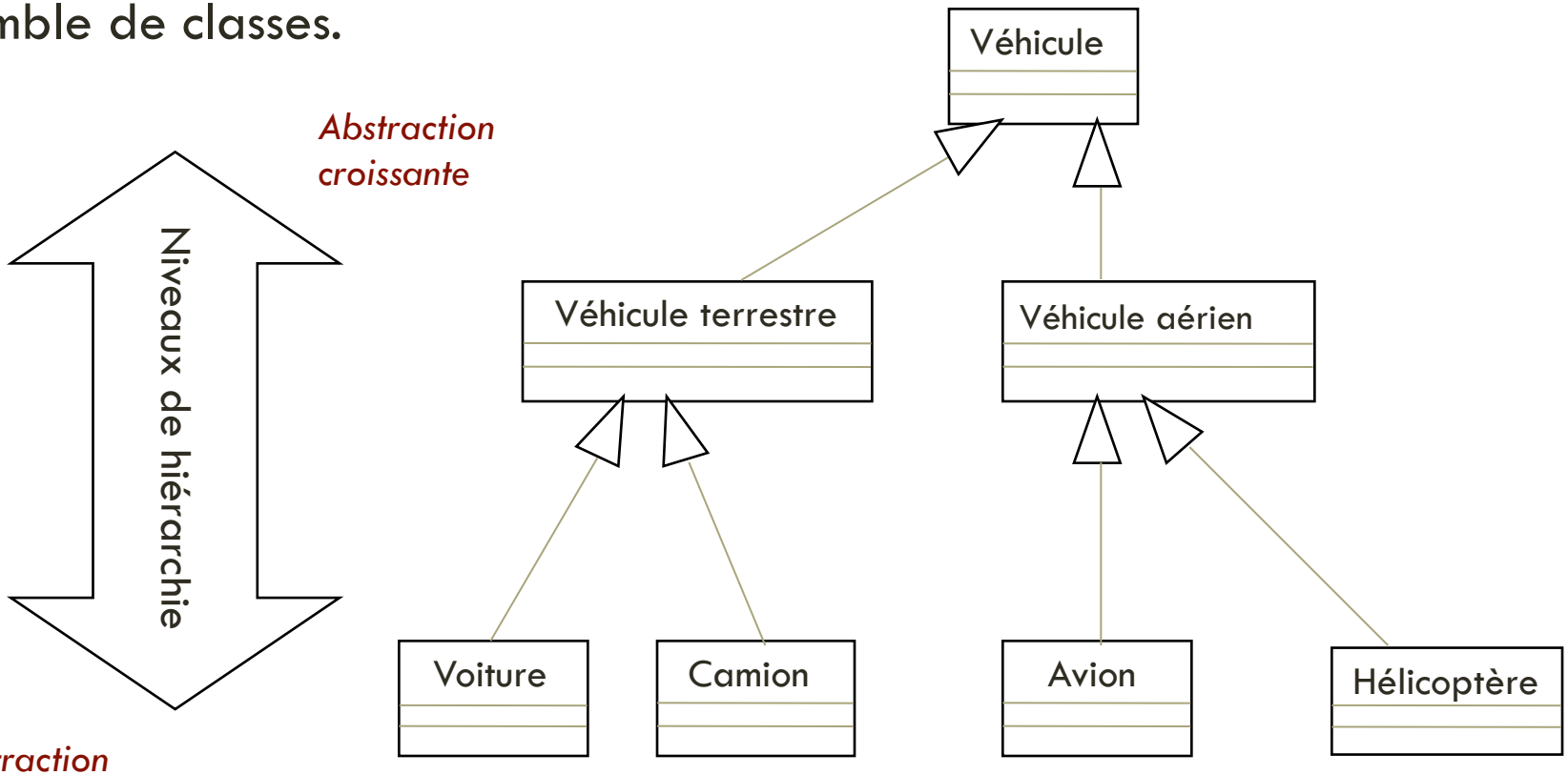
Décomposition d'un problème complexe en sous-problèmes plus simples



HIÉRARCHIE

mécanisme de transmission des propriétés d'une classe (ses attributs et méthodes) vers une sous-classe.

Plusieurs classes peuvent être généralisées en une classe qui les factorise, afin de regrouper les caractéristiques communes d'un ensemble de classes.



Les classes ayant le même niveau de hiérarchie devraient avoir le même niveau d'abstraction

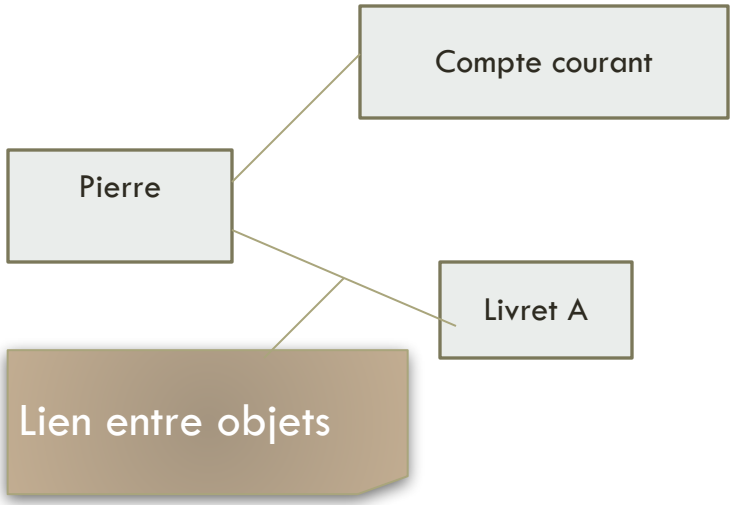
LES OBJETS

Entité physique

Concept

Objets anonymes

Liens entre objets



Caractéristiques fondamentales d'un objet

- Identité unique → identifiant = clé naturelle
- Etat = { valeurs de tous les attributs d'un objet }
- Comportement = { compétences d'un objet } = { opérations }

LE DIAGRAMME DE CLASSES

collection d'éléments de modélisation statiques (**classes, paquetages**...), qui montre la structure d'un modèle.

Non représentation des aspects dynamiques et statiques

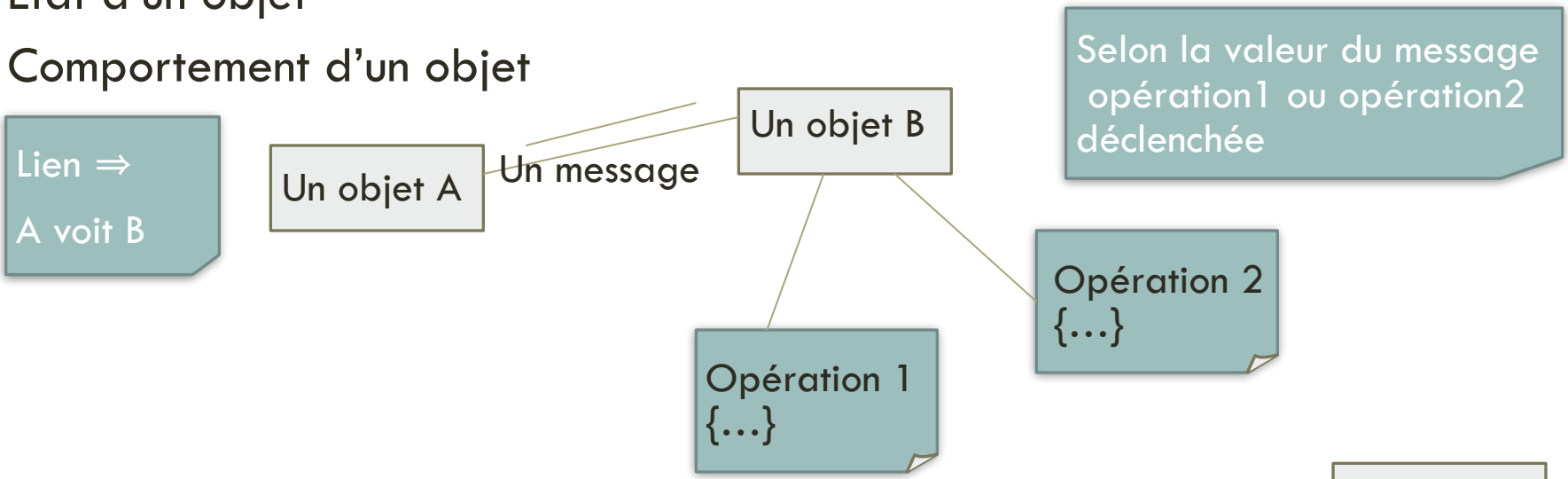
Pour représenter un contexte précis, un diagramme de classes peut être instancié en **diagrammes d'objets**.

Diagramme « central » dans un développement orienté objet

CARACTÉRISTIQUES D'UN OBJET

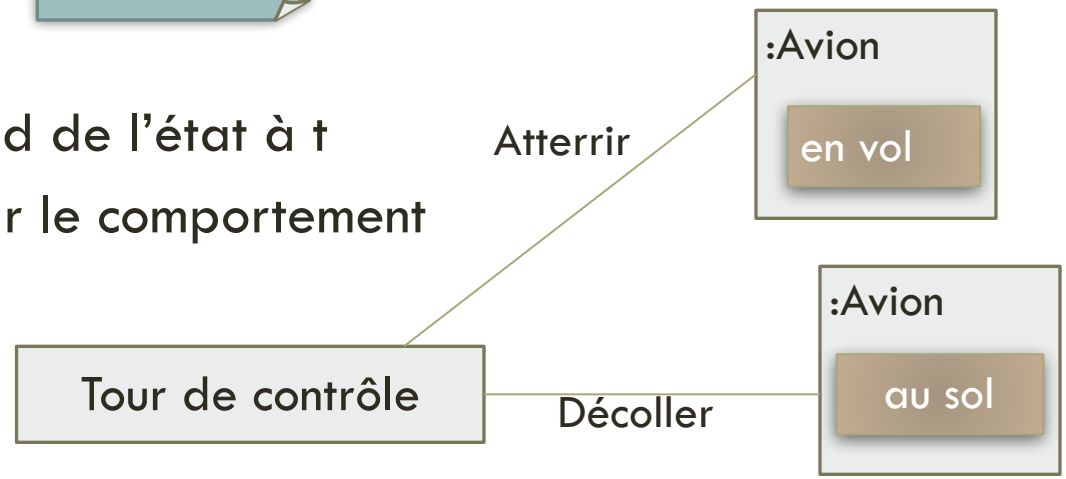
Etat d'un objet

Comportement d'un objet



Etat et comportement son liés

- Le comportement à t dépend de l'état à t
- L' état peut être modifié par le comportement



MISE EN ŒUVRE D'UN DIAGRAMME DE CLASSE

- Des classes
  - Des attributs : informations contenues dans une classe
  - Des opérations (méthodes): un service fournit par la classe (comportement)

Description abstraite d'un ensemble d'objets ayant les mêmes caractéristiques.

Ex: la classe nombre entiers représente l'ensemble des objets 1, 2, 3….

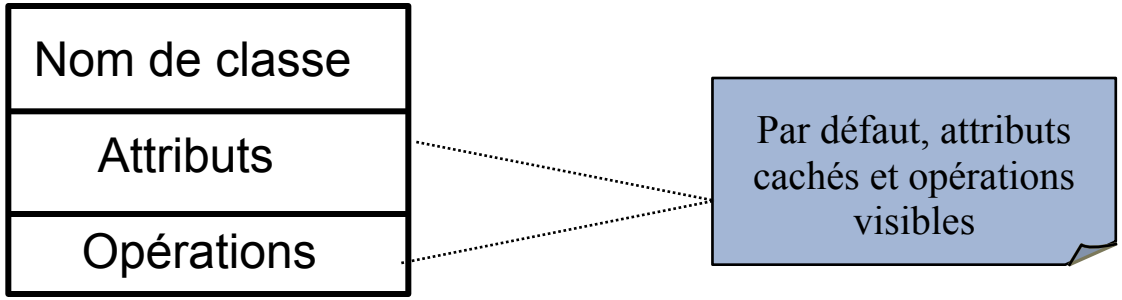
- Reliées entre elles par
  - Des associations
  - Des généralisations
  - Des agrégations …..

Liens sémantiques durables entre les classes.

Ex: la classe A contient des objets de la classe B, la classe B est une spécialisation de la classe D, la classe D a un accès à un objet de la classe E

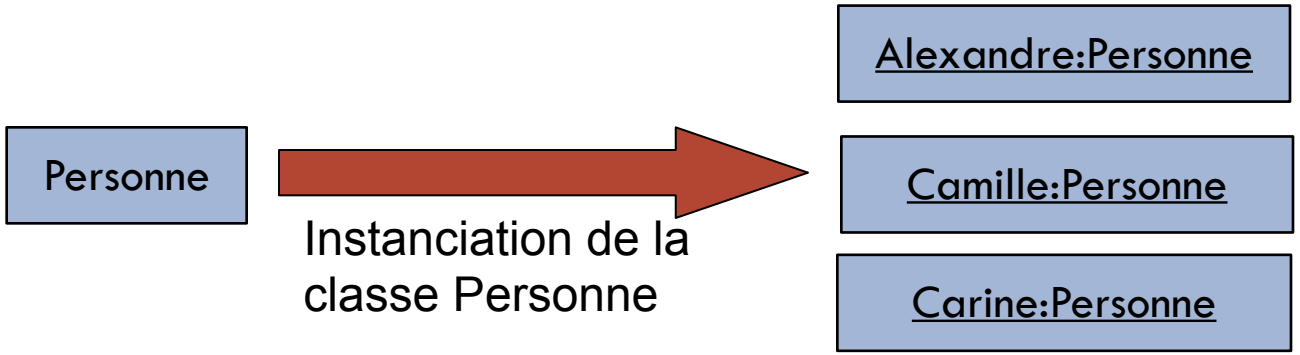
LES CLASSES

Une classe = Groupe d'objets
→Démarche d'**abstraction** pour identifier les caractéristiques communes à un ensemble d'entités



Généralités → classe et particularités → objets

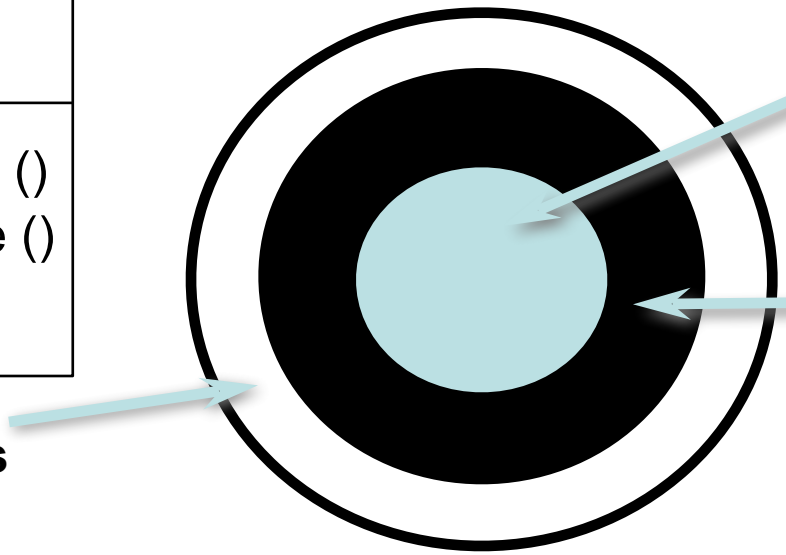
Objets informatique →
construits à partir de la classe →
instanciation : tout objet = instance de classe



VISIBILITÉ DES ATTRIBUTS ET DES OPÉRATIONS

- 3 niveaux de visibilité pour les attributs et les opérations
  - **+ public**, qui rend l'élément visible à tous les clients de la classe
  - **# protégé**, qui rend l'élément visible aux sous-classes de la classe
  - **- privé**, qui rend l'élément visible à la classe seule

Classe
- Attribut privé # Attribut protégé
+ Opération publique () # Opération protégée () - Opération privée ()



Opérations privées

Opérations protégées

Opérations publiques

RELATIONS ENTRE CLASSES

- Liens **sémantiques** particuliers entre objets → vus de manière abstraite dans le monde des classes
- Famille de **liens** entre objets → **relation** entre les classes de ces objets
- Liens entre objets = instances des relations entre classes

Le concepteur doit:

- Définir les liens sémantiques entre classes
- Choisir les relations adaptées
- Définir la multiplicité

TYPES DE RELATION ENTRE LES CLASSES

Association:
- Simple
- Association n-aire (ou multiple)
- Classe d'association
- Qualification (ou restriction)

Contenance
- Agrégation
- Composition

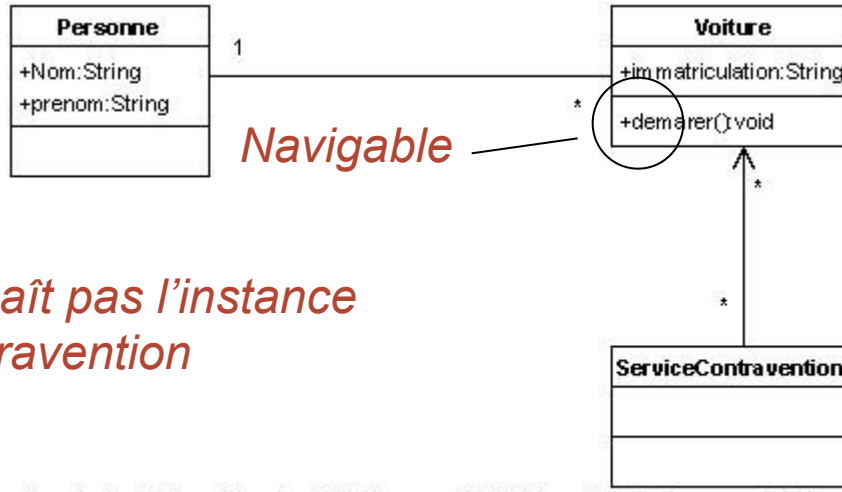
Héritage
- Spécialisation
- Généralisation

Indication de multiplicité:
Nombre d'instances d'objets qui participent à la relation

LES ASSOCIATIONS

- Connexion sémantique bidirectionnelle entre deux classes

- **Navigabilité**
  - Par défaut une association est navigable dans les deux sens
    - Chaque instance de voiture a un lien vers le propriétaire
    - Chaque instance de Personne a un ensemble de lien vers les voitures
  - Restriction de la navigabilité:
    - Définir quelles classes ne connaissent pas les autres

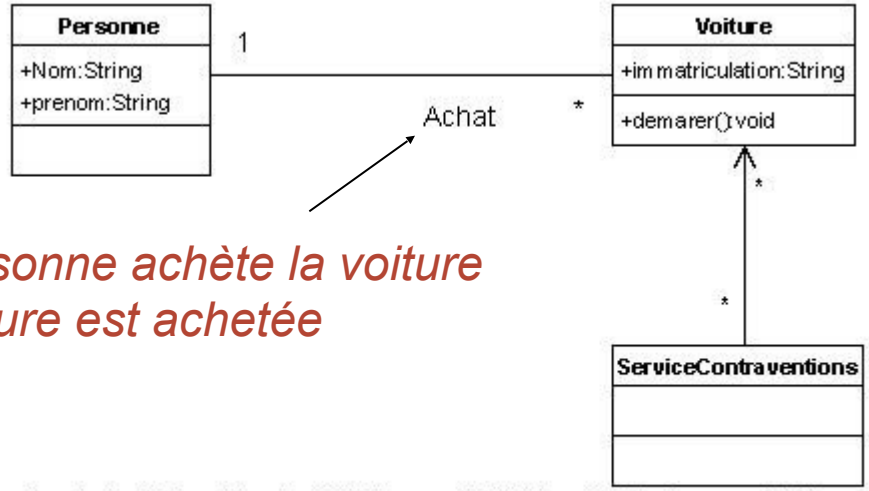


La voiture ne connaît pas l'instance de service de contravention

DOCUMENTER UNE ASSOCIATION

Donner un nom à l'association

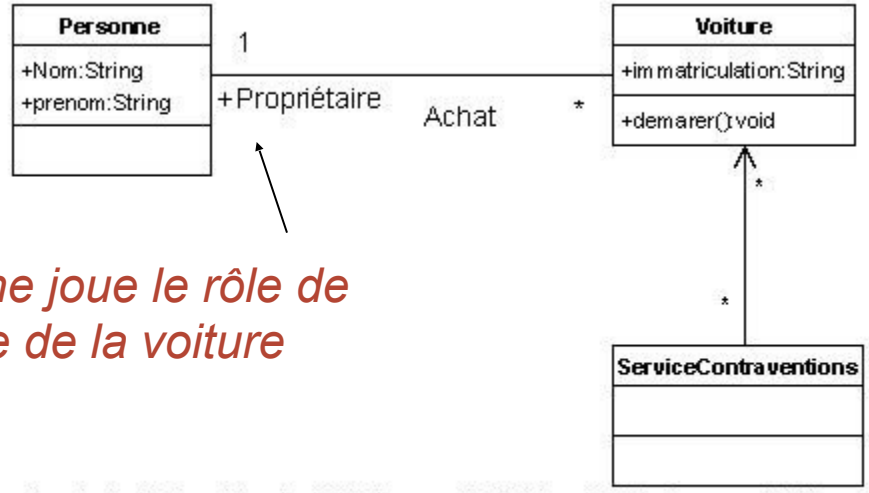
Précise le lien sémantique existant entre les classes



La personne achète la voiture
La voiture est achetée

Attribuer un rôle

Spécifier le rôle d'une classe dans une association donnée



La personne joue le rôle de propriétaire de la voiture

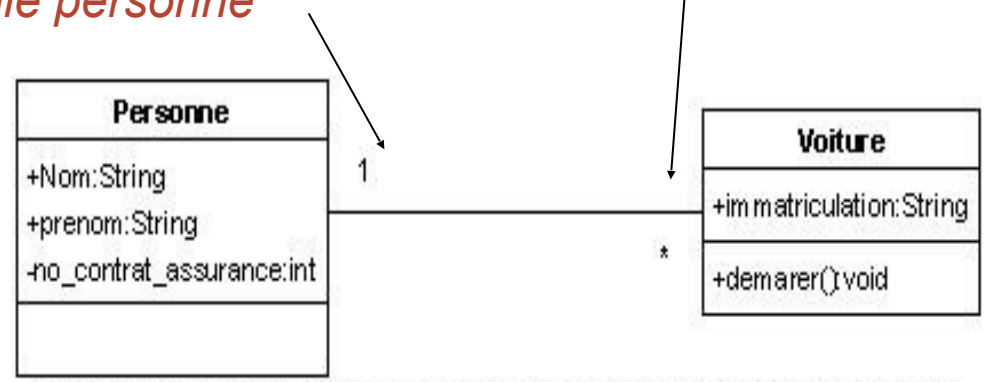
DÉFINIR LA MULTIPLICITÉ

1 : la classe est en relation avec un et un seul objet de l'autre classe

- **1..*** : la classe est en relation avec au moins un objet de l'autre classe
- **0..*** : la classe est en relation avec 0 ou n objets de l'autre classe
- **0..1** : la classe est en relation avec au plus un objet de l'autre classe

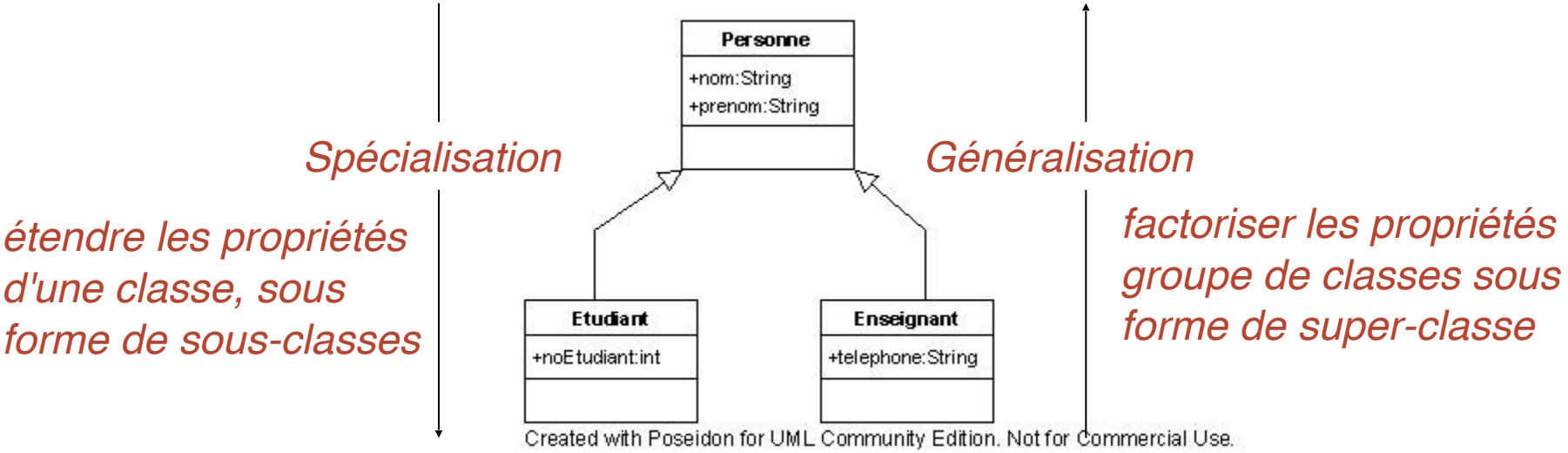
Une voiture est achetée par une et une seule personne

Une personne peut acheter 0 ou n voitures



HÉRITAGE

La **super-classe** est une classe générale liée à des **sous-classes** plus spécialisées qui « héritent » des propriétés de la super-classe et peuvent comporter des propriétés supplémentaires



- *Chaque personne de l'université est identifiée par son nom, prénom*
- *Les étudiants ont plus un noEtudiant*
- *Les enseignants ont un numéro de téléphone interne*

LA RELATION DE CONTENU/CONTENANT

Cas particulier d'association exprimant une relation de contenance

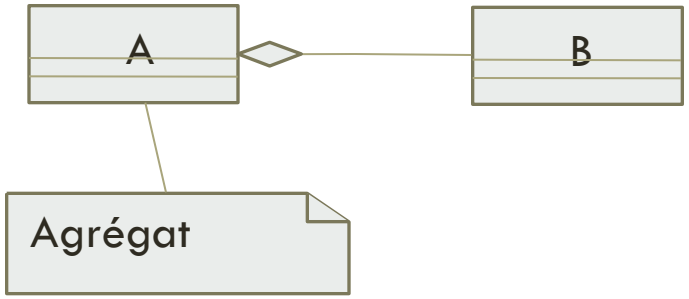
Exemple:

- Une voiture a 4 roues
- Un dessin contient un ensemble de figures géométriques
- Une présentation PowerPoint est composé de transparents
- Une équipe de recherche est composée d'un ensemble de personnes

Deux types de relations en UML

- Agrégation
- Composition (Agrégation forte)

AGRÉGATION



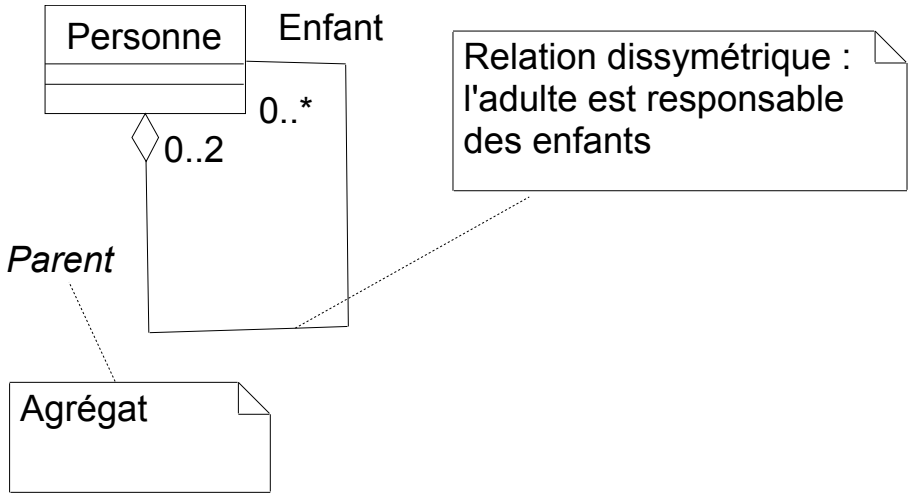
Type de relations

- A « contient » des instances de B,
- A « est composée » d'instances deB

Relation dissymétrique

notion de responsabilité

d'une classe envers une autre



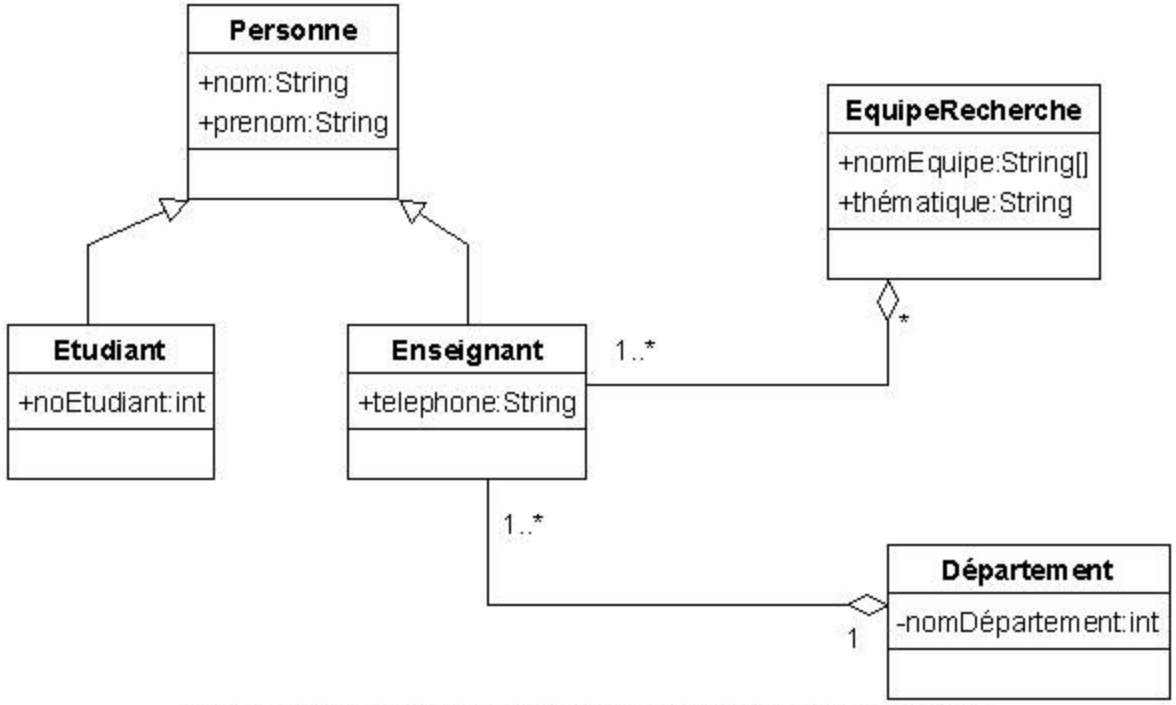
PROPRIÉTÉS DE L'AGRÉGATION

A un même moment, une instance d'élément agrégé peut être liée à plusieurs instances d'autres classes (l'élément agrégé peut être partagé).

Une instance d'élément agrégé peut exister sans agrégat (et inversement) : les cycles de vies de l'agrégat et de ses éléments agrégés peuvent être indépendants

L'enseignant est un composant d'une (ou plusieurs) équipe de recherche
d'un seul département

La disparition d'une équipe de recherche n'entraine pas
la disparition d'un enseignant



LA COMPOSITION

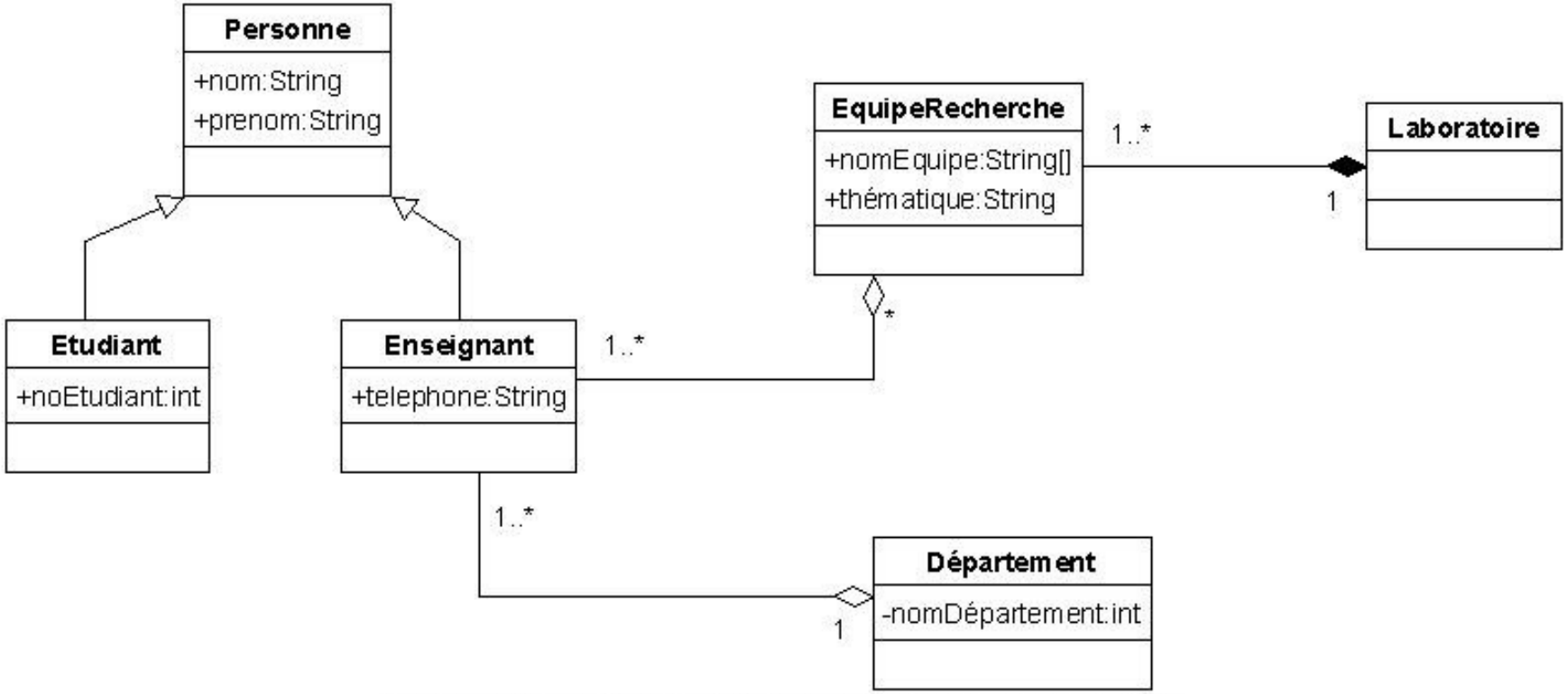
Exemple: « Une présentation PowerPoint est composé de transparents »

- La suppression de la présentation entraine la disparition des transparents qui la compose

Propriétés de la composition:

- A un même moment, une instance d'élément agrégé ne peut appartenir qu'à un seul agrégat (agrégation non partagée).
- Les durées de vie coïncident : la destruction de l'agrégat entraine la destruction des instances qui la compose

SUITE DE L'EXEMPLE

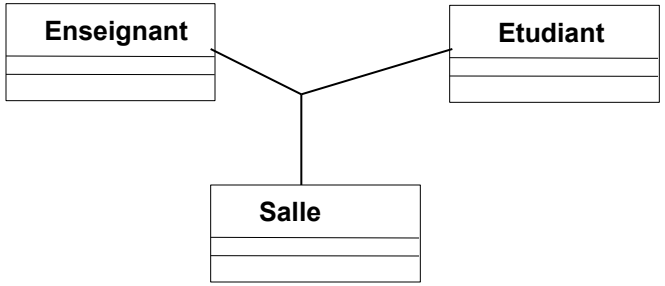


RETOUR SUR LES EXEMPLES

- **Un dessin contient un ensemble de figures géométriques**
  - Composition
- **Une présentation PowerPoint est composé de transparents**
  - Composition
- **Une équipe de recherche est composée d'un ensemble enseignants**
  - Agrégation
- **Une voiture a 4 roues**
  - Agrégation ou composition???
  - Dépend de la sémantique que l'on souhaite donner au modèle
  - Le choix dépendra du type de système traité (Ex: une entreprise spécialisée en pièce détachée ou une entreprise de destruction de véhicule)

ASSOCIATIONS N-AIRES

Type particulier d'association qui relie plus de deux classes



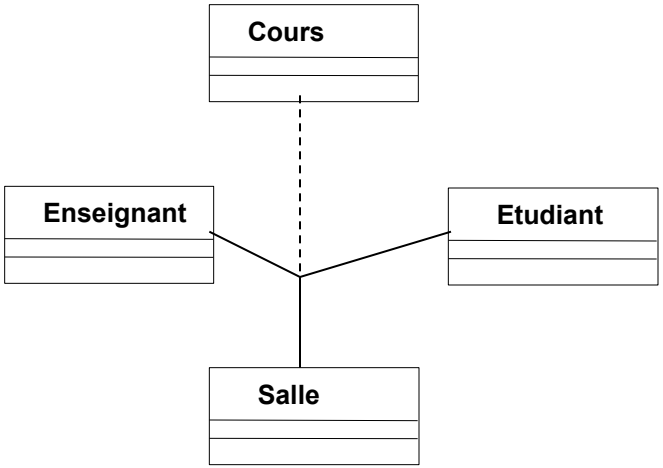
Attention:

- Sémantique floue
- Source d'erreur

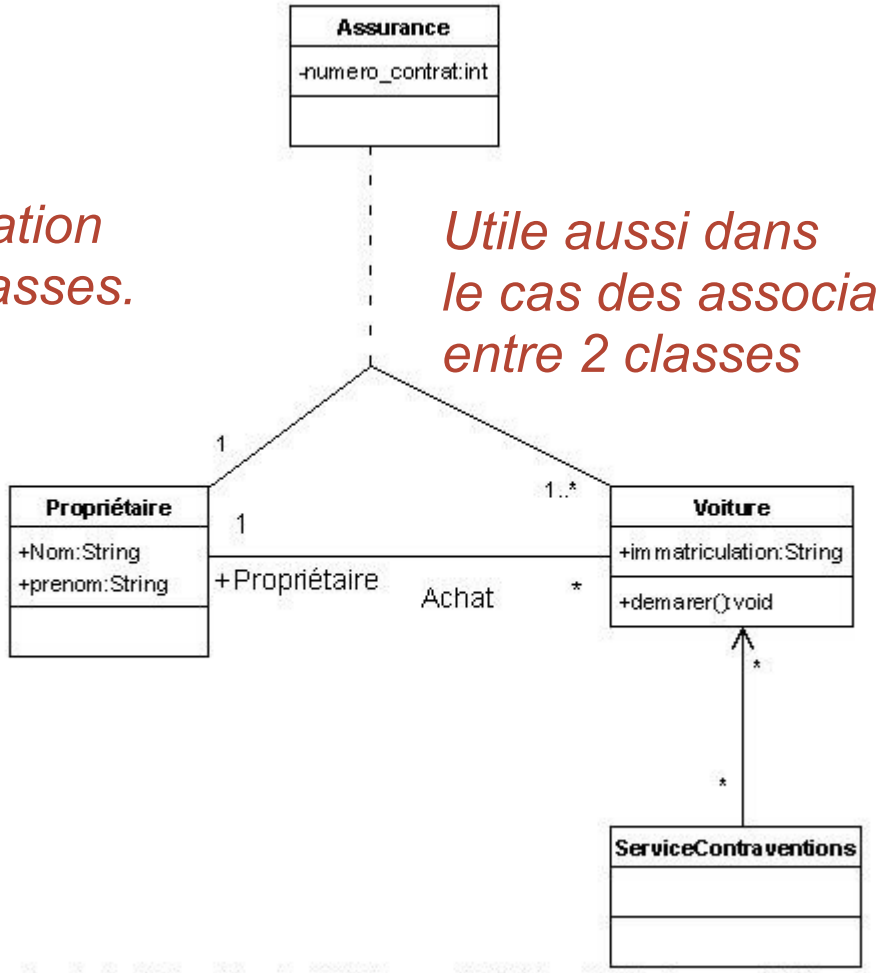
LES CLASSES D'ASSOCIATIONS

Alternative aux associations n-aires

Une classe qui réalise la navigation entre les instances d'autres classes.



Utile aussi dans le cas des associations entre 2 classes



DE LA MODÉLISATION UML À LA CONCEPTION

L'intérêt d'une modélisation statique précise

- Génération d'un « framework » du code source
- Support pour le concepteur au moment de l'implémentation

La déclaration des classes va dépendre:

- Du choix de la séparation des objets en classes
- De la multiplicité
- Des relations entre les classes

```
public class Propriétaire {

   public String Nom;
   public String prenom;
   public java.util.Collection voiture =
                new java.util.TreeSet();
}
```

Association non navigable dans le sens SC-Voiture

```
public class Voiture {
        public String immatriculation;
        public Propriétaire Propriétaire;
        public void demarer() {   }
}
```

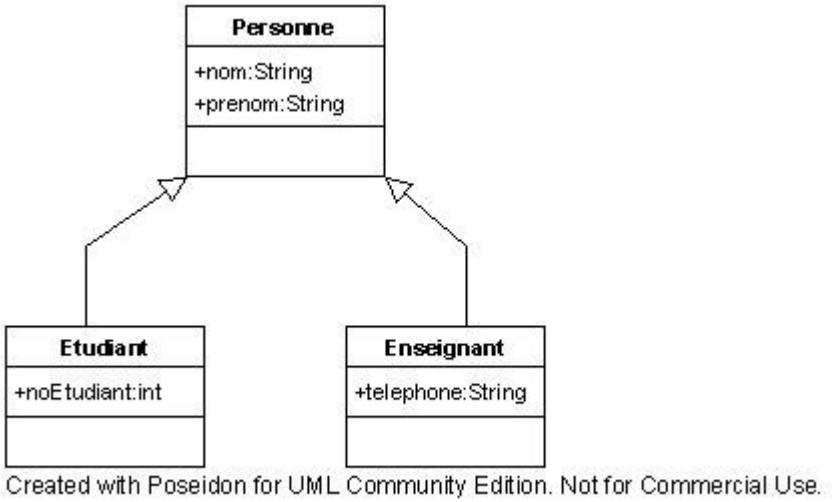


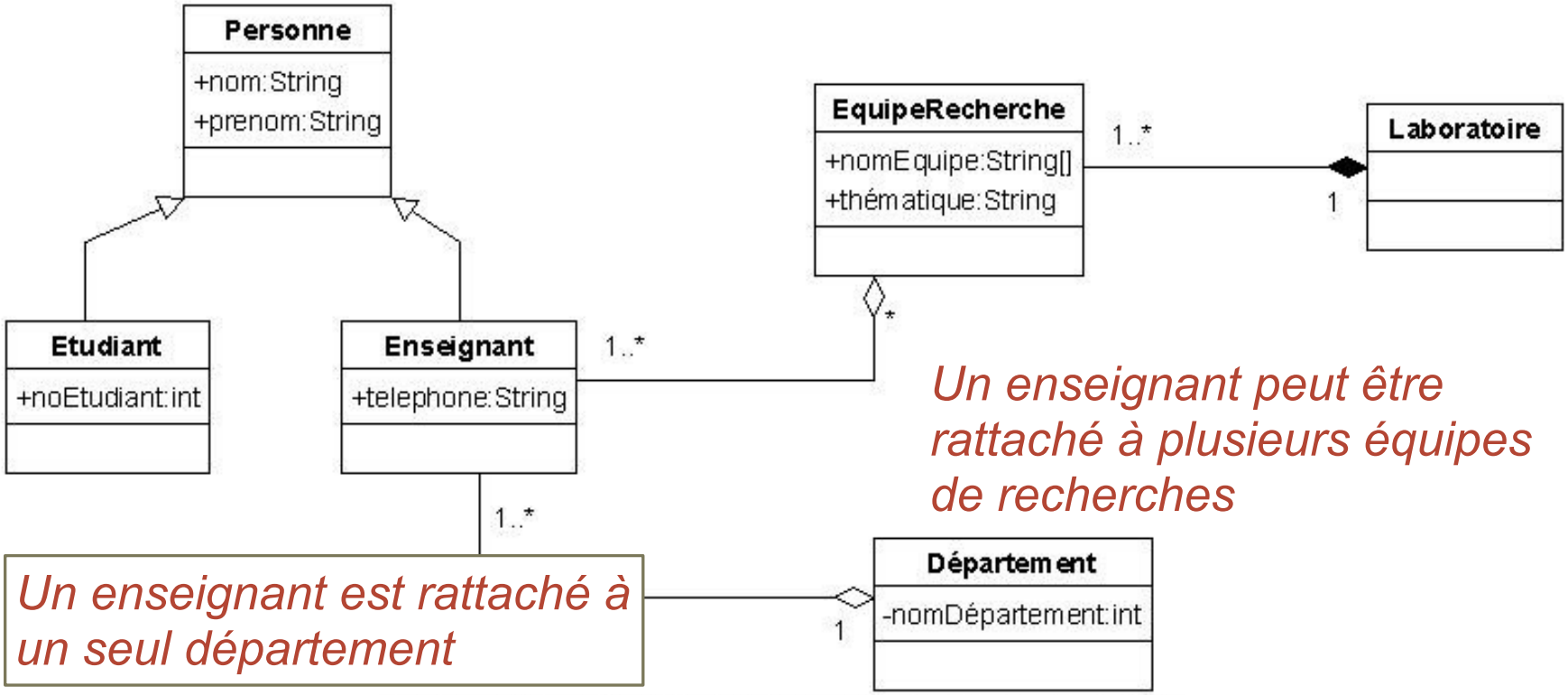
```
public class ServiceContraventions {

public java.util.Collection voiture = new java.util.TreeSet();
 }
```

TRADUIRE L'HÉRITAGE

```
public class Etudiant extends Personne {
        public int noEtudiant;
 }
```

```
public class Personne {
        public String nom;
        public String prenom;
 }
```

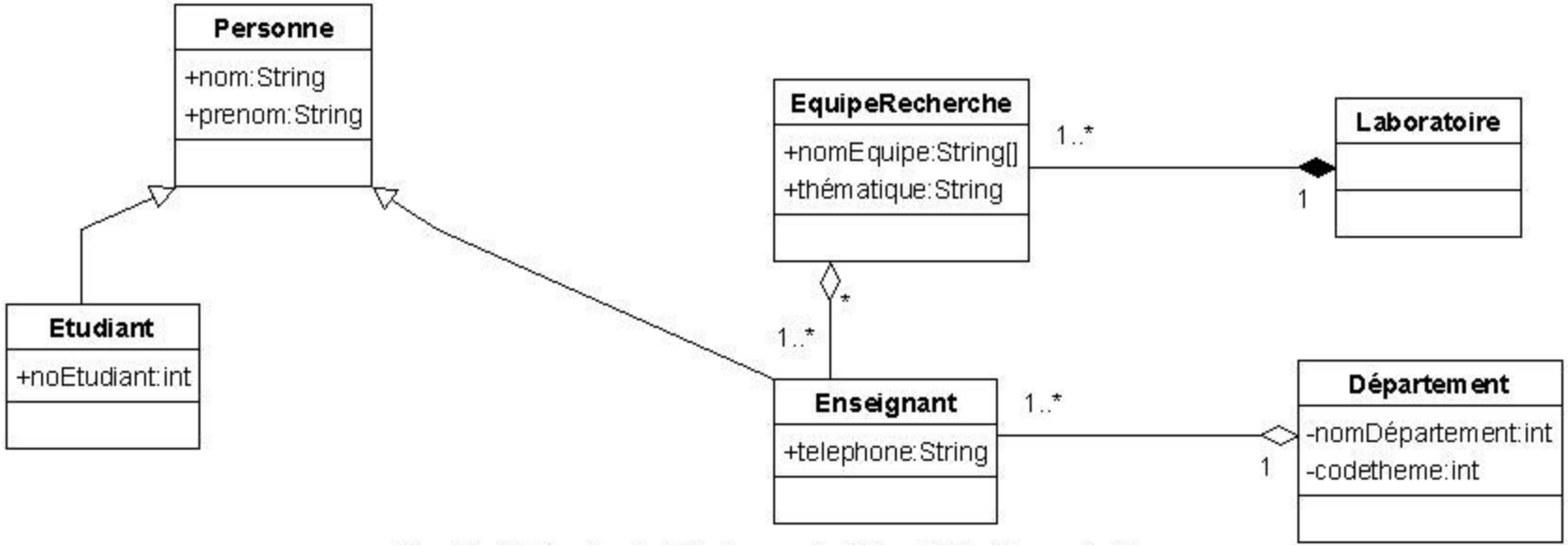





```
public class Enseignant extends Personne {
  public String telephone;
  public java.util.Collection equipeRecherche = new java.util.TreeSet();
  public Departement departement;
 }
```

```
public class Département {
private int nomDépartement;
private int codetheme;
public java.util.Collection enseignant = new java.util.TreeSet();
}
```

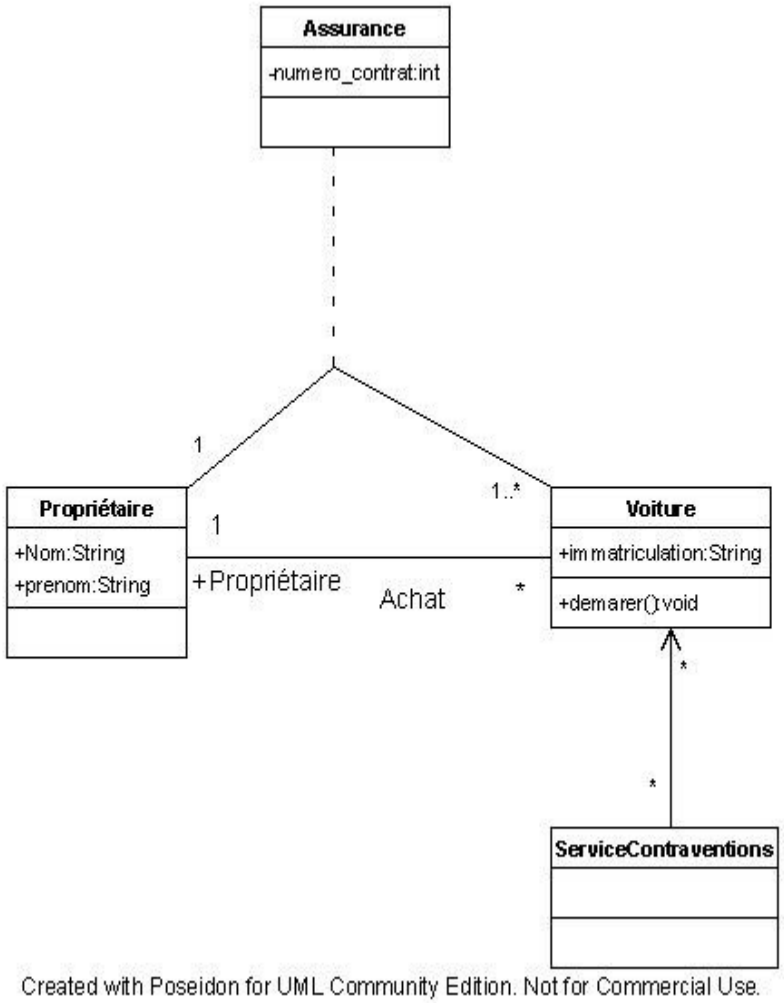
```
public class EquipeRecherche {
public String[] nomEquipe;
public String thématique;
public java.util.Collection enseignant = new java.util.TreeSet();
public Laboratoire laboratoire;
 }
```




```
public class Laboratoire {

public java.util.Collection equipeRecherche = new java.util.TreeSet();
 }
```

LES CLASSES D'ASSOCIATIONS

```
public class Propriétaire {

   public String Nom;
   public String prenom;
   public java.util.Collection voiture =
               new java.util.TreeSet();
   public java.util.Collection assurance_1
               new java.util.TreeSet();
}
```




```
public class Voiture {
        public String immatriculation;
        public Propriétaire Propriétaire;
        public Assurance assurance;

        public void démarrer() {  }
}
```

```
public class Assurance {
   private int numero_contrat;
   public Voiture voiture;
   public Propriétaire propriétaire;
 }
```

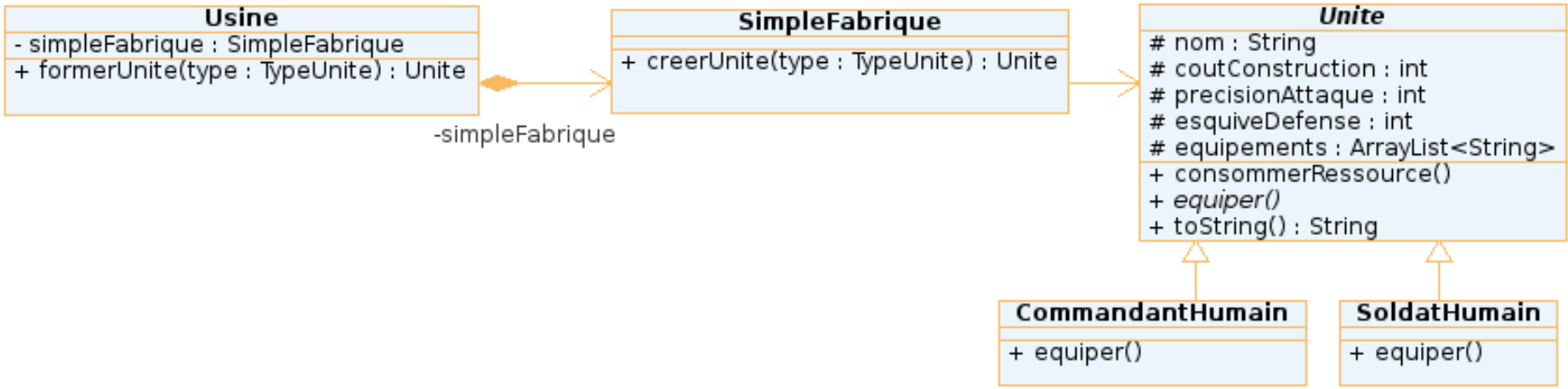
Un exemple de polymorphisme : la Fabrique

- **Problème** : on souhaite pouvoir construire différents types d'objets en fonction d'un paramètre/contexte donné.
  - Ensemble de conditions et choix de la classe à instancier ? Couplage fort … Que se passe-t-il si le catalogue d'objets à créer évolue?
- **Solution** :
  - Utilisation d'une fabrique en capsulant la logique de création sur un objet générique
  - Le polymorphisme garantit que les traitements se réalisent sur l'instance réellement créée.
  - Permet de déterminer dynamiquement quel objet d'un ensemble de sous-classes doit être instancié.
- **Utilité**
  - Le client ne peut déterminer le type d'objet à créer qu'à l'exécution
  - Il y a une volonté de centraliser la création des objets

Une première version simple

- Création des instances au travers d'une simple fabrique

Implémentation d'un début de solution avec une Simple Fabrique



http://design-patterns.fr/fabrique-en-java

La classe Unite

Implémentation de la classe abstraite Unite

```
// Classe abstraite dont toutes les unités du jeu hériteront.
public abstract class Unite
{
    protected String nom;// Nom de l'unité.
    protected int coutConstruction;// Coût de construction de l'unité.
    protected int precisionAttaque;// Précision de l'attaque de l'unité.
    protected int esquiveDefense;// Faculté d'esquiver une attaque de l'unité.
    protected ArrayList equipements;// Tableau des équipements de l'unité.

    // Méthode qui consomme les ressources pour créer une unité.
    public void consommerRessource()
    {
        System.out.println("Consomme "+this.coutConstruction+" ressources pour la création de l'unité.");
    }

    // Méthode abstraite qui permet d'équiper l'unité.
    public abstract void equiper();

    // Méthode générique pour l'affichage de l'unité.
    public String toString()
    {
        String str = "Nom : "+this.nom+"\n";
        str += "Coût de construction : "+this.coutConstruction+"\n";
        str += "Précision d'attaque : "+this.precisionAttaque+"\n";
        str += "Esquive en défense : "+this.esquiveDefense+"\n";
        str += "Equipements : ";
        for(int i=0; i<this.equipements.size(); i++)
        {
            str += this.equipements.get(i)+" ";
        }
        return str;
    }
}
```

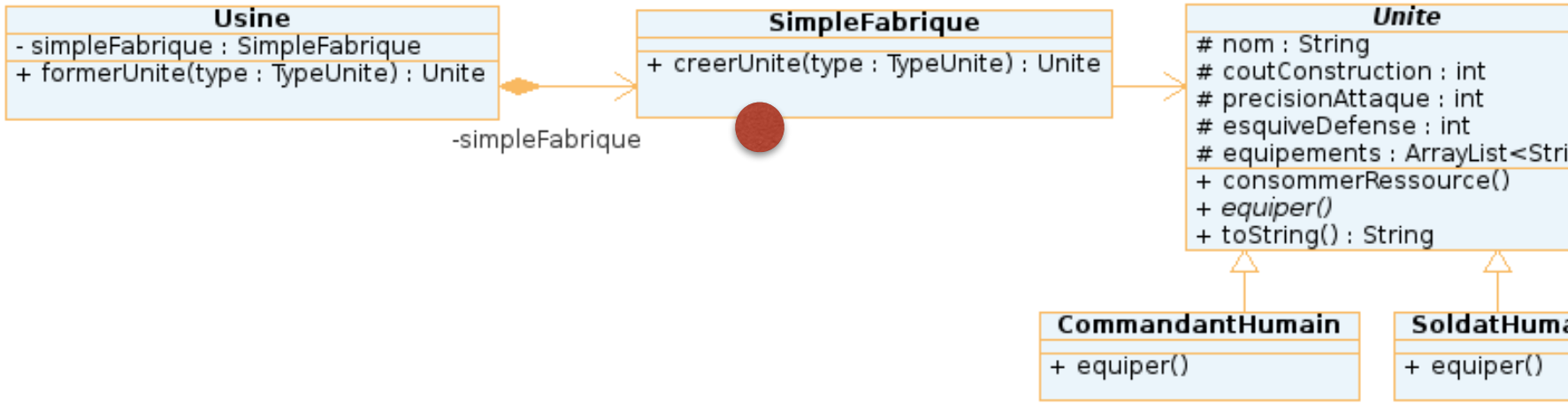
http://design-patterns.fr/fabrique-en-java

Les classes de soldat

```java
// Classe représentant un soldat humain.
public class SoldatHumain extends Unite
{
    // Constructeur pour un soldat humain.
    public SoldatHumain()
    {
        this.nom = "Fantassin";
        this.coutConstruction = 5;
        this.precisionAttaque = 1;
        this.esquiveDefense = 2;
        this.equipements = new ArrayList();
    }

    // Equiper un soldat humain.
    public void equiper()
    {
        this.equipements.add("Pistoler");
        this.equipements.add("Bouclier");
        System.out.println("Equipement d'un soldat humain (Pistoler, Bouclier).");
    }
}
```

```java
// Classe représentant un commandant humain.
public class CommandantHumain extends Unite
{
    // Constructeur pour un commandant humain.
    public CommandantHumain()
    {
        this.nom = "Lieutenant";
        this.coutConstruction = 14;
        this.precisionAttaque = 5;
        this.esquiveDefense = 2;
        this.equipements = new ArrayList();
    }

    // Equiper un commandant humain.
    public void equiper()
    {
        this.equipements.add("Uzi");
        this.equipements.add("Bouclier");
        System.out.println("Equipement d'un commandant humain (
    }
}
```

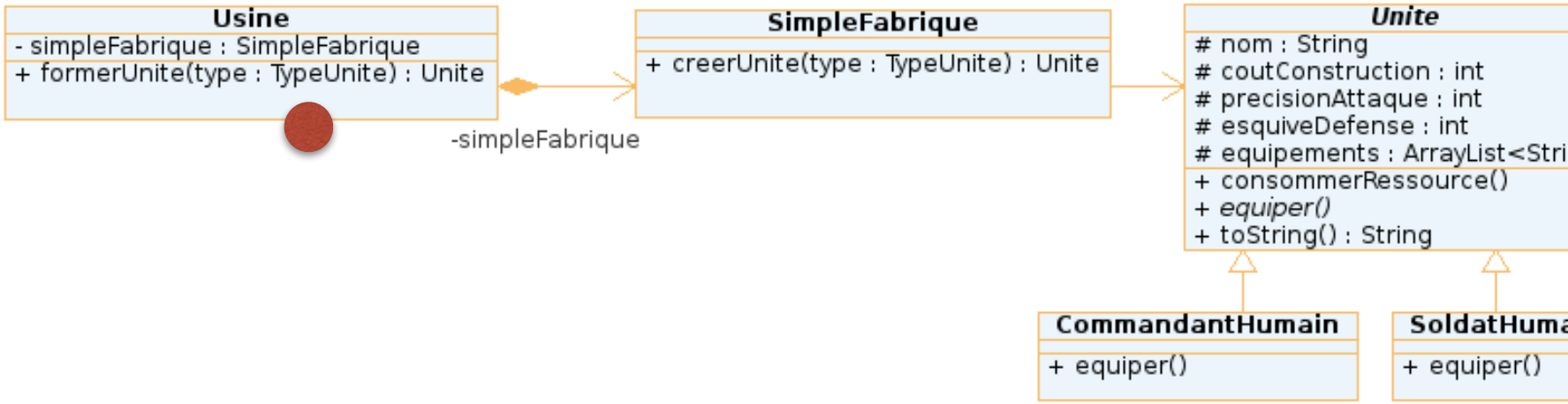
Implémentation d'un début de solution avec une Simple Fabrique




```
{
    // La création d'une unité en fonction de son type est encapsulée dans la fabrique.
    public Unite creerUnite(TypeUnite type)
    {
        Unite unite = null;;
        switch(type)
        {
            case SOLDAT:unite = new SoldatHumain();break;
            case COMMANDANT:unite = new CommandantHumain();break;
        }
        return unite;
    }
}

// Enumération des types d'unités.
public enum TypeUnite
{
    SOLDAT,
    COMMANDANT
}
```

http://design-patterns.fr/fabrique-en-java

Implémentation d'un début de solution avec une Simple Fabrique



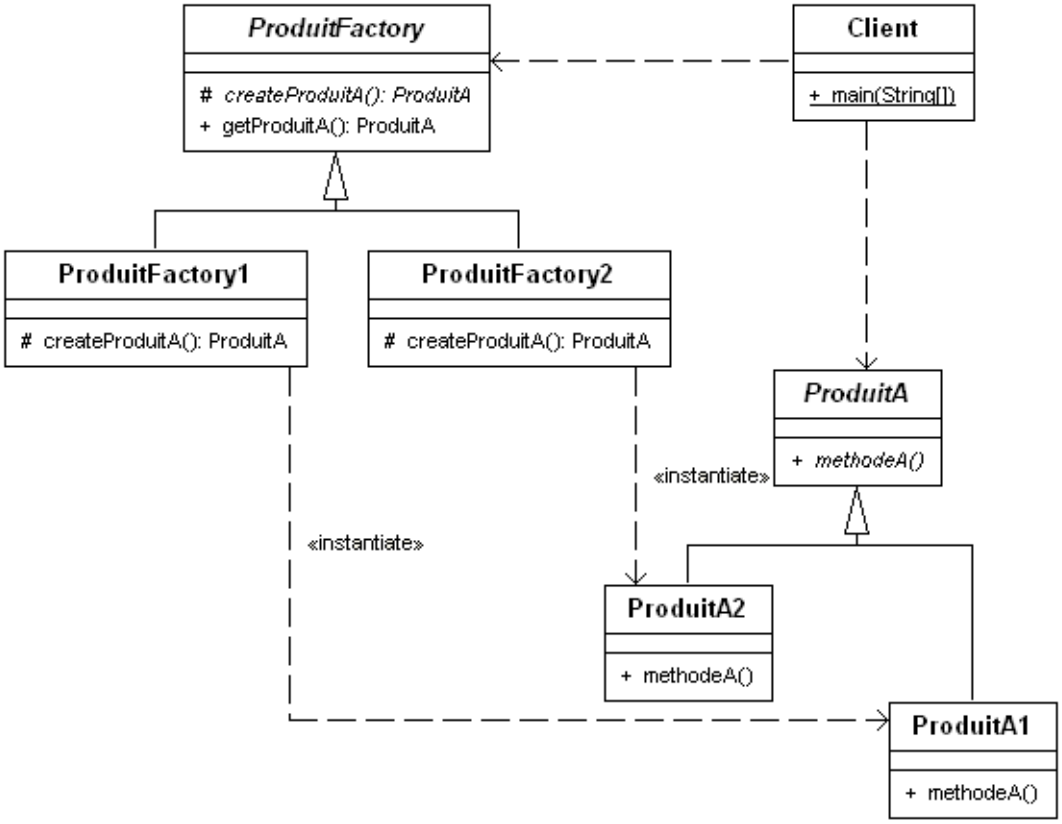
Implémentation de la classe Usine

```
// Classe usine qui représente un bâtiment capable de construire des unités.
public class Usine
{
    private SimpleFabrique simpleFabrique;// Attribut contenant la fabrique simple.

    // Le constructeur permet de sélectionner la fabrique à utiliser.
    public Usine()
    {
        this.simpleFabrique = new SimpleFabrique();
    }

    // Méthode qui permet de construire l'ensemble des unités.
    public Unite formerUnite(TypeUnite type)
    {
        Unite unite = this.simpleFabrique.creerUnite(type);
        unite.consommerRessource();
        unite.equiper();
        return unite;
    }
}
```

http://design-patterns.fr/fabrique-en-java

Fabrique (C) - modèle UML

- Extension à la problématique suivante :
  - Que faire lorsque l'on a plusieurs types de produits et qu'on doit gérer des modes de fabrication différents ?



http://www.jmdoudoux.fr/java/dej/chap-design-patterns.htm

Les produits

```
public abstract class ProduitA {
   public abstract void methodeA();
}
```

```
public class ProduitA2 extends ProduitA {

   public void methodeA() { System.out.println("ProduitA2.methodeA()"); }
}
```

```
public class ProduitA1 extends ProduitA {

   public void methodeA() { System.out.println("ProduitA1.methodeA()"); }

}
```

Les fabriques

```
public abstract class ProduitFactory {


   public abstract ProduitA createProduitA();

   public ProduitA getProduitA() {
      return createProduitA();

   }

}
```

```
public class ProduitFactory1 extends ProduitFactory {

   public ProduitA createProduitA() {

      return new ProduitA1();
   }
}
```

```
public class ProduitFactory2 extends ProduitFactory {

   public ProduitA createProduitA() {

      return new ProduitA2();
   }
}
```

Le client

```
public class Client {

  public static void main(String[] args) {

    ProduitA produitA = null;
    produitA = new ProduitFactory1().getProduitA();
    produitA.methodeA();

    produitA = null;
    produitA = new ProduitFactory2().getProduitA();
    produitA.methodeA();


  }

}
```