UML

Yannick Prié
UFR Informatique – UCB Lyon 1
2004-2005

Avertissement

- Présentation UML1 adaptée pour UML2
- Nécessiterait une refonte complète

1
Système d'information et modélisation

Des exemples de SI

- Une application de gestion de stocks d'une épicerie, d'un supermarché
- Un site web de vente en ligne
- Une bibliothèque numérique
- Un portail pour une association
- Un intranet pour l'UFR informatique
- ...

Les SI évoluent

- Au départ
  - systèmes centralisés (mainframe) propriétaires
  - applications indépendantes, données redondantes
  - utilisateurs hors système d'information
- Ensuite
  - systèmes hétérogènes
  - applications reliées, données dans SGBD
  - utilisateurs sur des stations dédiées (saisie / consultation)
- Maintenant
  - architectures hautement connectées, client/serveur généralisé
  - multiples SGBD, modules indépendants interopérables
  - utilisateurs partout, décideurs, tâches évoluantes

Les méthodes de conception évoluent aussi

- Au départ
  - conception par découpage en sous-problèmes, analytico fonctionnelle
  - méthodes d'analyse structurée
- Ensuite
  - conception par modélisation : « construire le SI, c'est construire sa base de données »
  - méthodes globales qui séparent données et traitements (MERISE)
- Maintenant
  - conception pour et par réutilisation : Frameworks, Design Patterns, bibliothèques de classes
  - méthodes
    - exploitant un capital d'expériences
    - unifiées par une notation commune (*UML*)
    - procédant de manière incrémentale
    - validant par simulation effective

(Morand)

Rappel : objectifs du concepteur

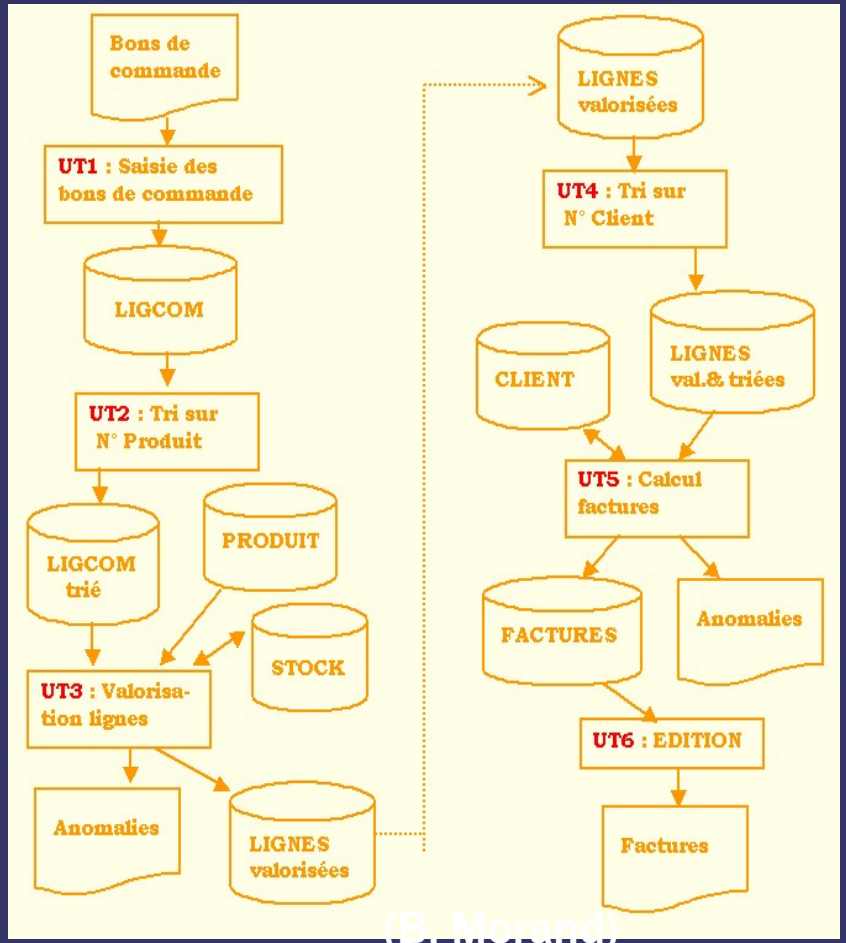
- Concevoir une application qui réponde aux besoins des utilisateurs
- Dont on puisse prévoir à l'avance les fonctionnalités principales
- Dont on puisse vérifier qu'elle fait bien ce qui avait été prévu
- Capable d'évoluer, sécurisée, documentée, …

La modélisation ?

- Pour B. Morand :
  - créer un modèle avec
    - figuration : représenter les concepts comme figures
    - imitation : copier les relations perçues
    - formalisation : mettre de l'ordre
  - pour
    - communiquer
    - préparer la réalisation
      - ce que l'application devra faire (spécification)
      - comment elle est organisée du point de vue de l'utilisateur (réalisation)

Modélisation par les fonctions

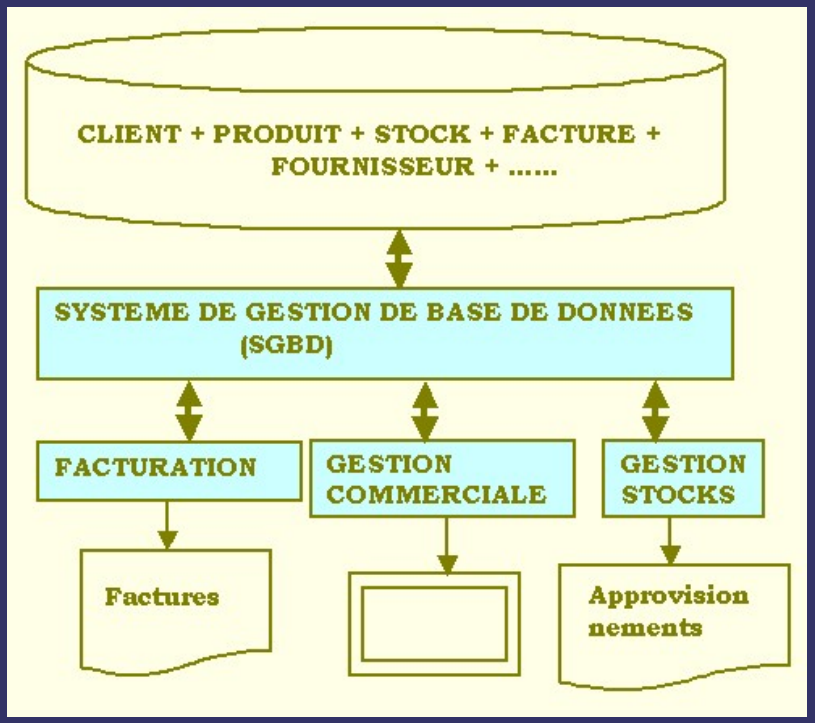
- Décomposition en
  - systèmes/sous-systèmes
  - fonctions / sous-fonctions
- Avec fonctions entrées, sorties, contrôles (proche du fonctionnement de la machine)
- Fonctions contrôlent la structure : si la fonction bouge, tout bouge
- Données non centralisées



(B. Morand)

Modélisation par les données

- Privilégie les flots de données et les relations entre structures de données (apparition des SGBD)
- Traitements = transformations de données dans un flux (notion de processus)
- Acteurs mis en évidence
- Exemple : MERISE
  - plusieurs niveaux d'abstraction
  - plusieurs modèles
  - mais cycles de développement trop figés (cascade), pas d'ingénierie concourante, avenir incertain, coopération de systèmes, etc.



Modélisation orientée-objet (1)

- Dépasser les méthodes pour la gestion classique pour intégrer des types d'applications nouvelles, interactives (CAO, bureautique, télécommunications, …), de plus en plus complexes
- Intervient au niveau *analyse* et *conception*
- Passer du monde des objets (du discours) à celui de l'application en *complétant des modèles* (pas de transfert d'un modèle à l'autre)
- Intégration des constituants d'un système à la fois de façon statique et dynamique.
- Système = objets collaborant → étudier ce que le système est (données), et en même temps ce qu'il fait (fonctions)

Modélisation orientée-objet (2)

- Les fonctions deviennent des collaborations entre objets composant le système : il peut y avoir évolution fonctionnelle sans remise en cause de la structure statique du logiciel.
- Démarche à la fois ascendante et descendante, récursive, encapsulation
- Abstraction forte
- Orienté vers la réutilisation : notion de composants, modularité, extensibilité, adaptabilité (objets du monde), souplesse

Un foisonnement de méthodes

OOD	Object Oriented Design (G. Booch)	1991
HOOD	Hierarchical Object Oriented Design (Delatte & al.)	1993
OOA	Object oriented analysis (Schlaer ,Mellor)	1988, 92
OOA/OOD	(Coad & Yourdon)	1991
OMT	Object Modeling Technique	1991
OOSE	Object oriented software engineering (Jacobson & al.)	1992
OOM	Object oriented Merise (Bouzeghoub & Rochfeld)	1993
Fusion	(Coleman & al.)	1994

De nombreuses méthodes (>50) ayant des avantages et des inconvénients différents, autant de notations différentes (cf. objets, classes, relations jusqu'ici)

Histoire d'UML

1994 Rumbaugh (OMT) rejoint Booch (OOD) chez Rational Software

but : créer une *méthode* en commun (*méthode unifiée*)

1995 : présentation v0.8

1995 Arrivée de Jacobson (OOSE)

1996 L*angage* unifié UML 0.9 (Unified Modeling Language), implication de l'OMG

1997

Présentation de UML à l'OMG (Object Management Group)

UML 1.1 adopté par la plupart des compagnies

1999 UML 1.3

2003

Utilisation croissante, UML1.4, puis UML1.5

2005

UML 2.0 quasi-publié

Unified Modeling Language

- Famille de notations graphiques
- Combinaison de principes à succès
  - Modélisation de données (E/A) → modèles de classes
  - Modélisation de l'activité
  - Modélisation objet
  - Modélisation en composants
- Visualisation / spécification / construction / documentation des artefacts d'une application complète
- Un méta-modèle
- Pas de méthode préconisée
- Utilisation possible au travers de différentes technologies : impératif, objet, relationnel…

II
Vues générales des diagrammes et mécanismes de base d'UML

Modélisation visuelle (Rational)

- Modéliser en utilisant une notation standard, pour capturer les parties essentielles d'un système
- Moyen de communication entre acteurs du développement de l'application
- Passage des objets et de la logique de fonctionnement du domaine à ceux de l'application : couplage entre concepts et artefacts exécutables
- Surmonter la complexité (7 +/-2) : abstraction
- Définir l'architecture de l'application
- Promouvoir la réutilisation

Modélisation en diagrammes

- Modèle = ensemble d'éléments de modélisation vus dans un ensemble de diagrammes
- Diagramme

= mise ensemble d'éléments de visualisation représentant des éléments de modélisation (graphe)

Montre une partie des éléments de modélisation, avec le niveau de détail utile (vues)

Modélisation en diagrammes

Modèle

Vues dynamiques

Chat — effectue — Toilette

Vue statique

Animal

Chat

Toilette

Départ → Laver le museau → Laver le ventre → Laver derrière les oreilles → Dormir → Fin

Et la souris ?

Notation et méta-modèle

- syntaxe : la notation elle-même, ce que l'on peut décrire → les diagrammes
- sémantique : méta-modèle, ie. ce qui est signifié en termes de description objet → ce qui est sous-jacent

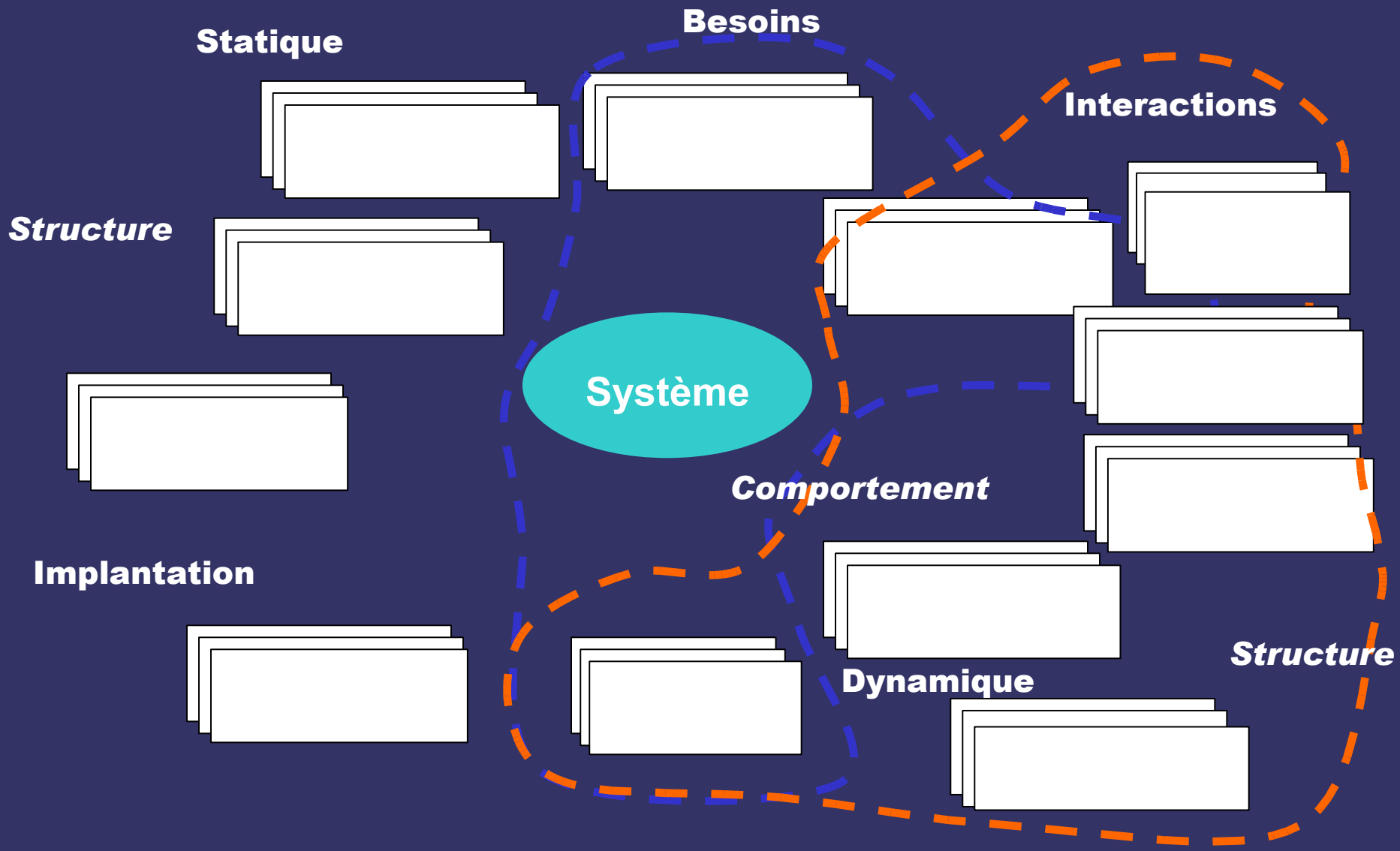
Trois modes d'utilisation d'UML (Fowler/Mellor)

- **Esquisse**
  - Conception / communication
  - Incomplètude
- Plan
  - Exhaustivité, outils bidirectionnels
- Programmation
  - Model Driven Architecture / UML exécutable
  - Implantation automatique. Réaliste ?

Insistance sur les diagrammes

Insistance sur le méta-modèle

UML2.0 : les 13 types de diagrammes



UML : objectifs

- *Montrer les limites d'un système et ses fonctions principales* (pour les utilisateurs) à l'aide des cas d'utilisation et des acteurs
- Illustrer les réalisations de CU à l'aide de diagrammes d'interaction
- *Représenter la structure statique d'un système* à l'aide de diagrammes de classes, associations, contraintes
- *Modéliser la dynamique, le comportement des objets* à l'aide de diagrammes états/transitions
- *Révéler l'implantation physique de l'architecture* avec des diagrammes de composants et de déploiement
- Étendre les fonctionnalités du langage avec des stéréotypes
- Un langage utilisable par l'homme et la machine : permettre la génération automatique de code, et la rétro-ingénierie

UML

- Règles
  - normatives
    - comment il faut faire, pas vraiment d'accord
  - descriptives
    - comment les gens font : conventions
- A ne pas oublier
  - on peut supprimer n'importe quel élément d'un diagramme
  - on est toujours libre de dessiner ce que l'on veut (surtout en mode esquisse)

Remarques

- Présentation d'UML non exhaustive
- Pour la description exacte de toute la syntaxe et sémantique :

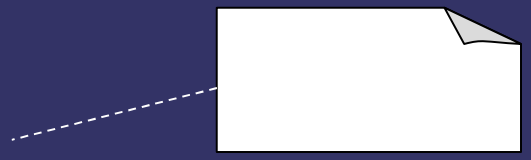
  http://www.omg.org

Mécanismes de base (1)

- **Mots-clés**
  - classer les éléments similaires du modèle en familles << mot-clé >>
    - << interface >>
- **Stéréotypes**
  - Beaucoup plus formalisé
    - Exemple : classes Fenêtre, Icône, Bouton
      → valeurs communes (afficher/masquer),
      → stéréotype << visuel >
- **Valeur marquée**
  - Attacher une information arbitraire à un élément de modélisation,
  - Ajouter une propriété à un élément de diagramme : paire (nom,valeur)
  - notation : { nom = valeur }
  - Exemple : {auteur = YP }, {version = 1.3}

Mécanismes de base (2)

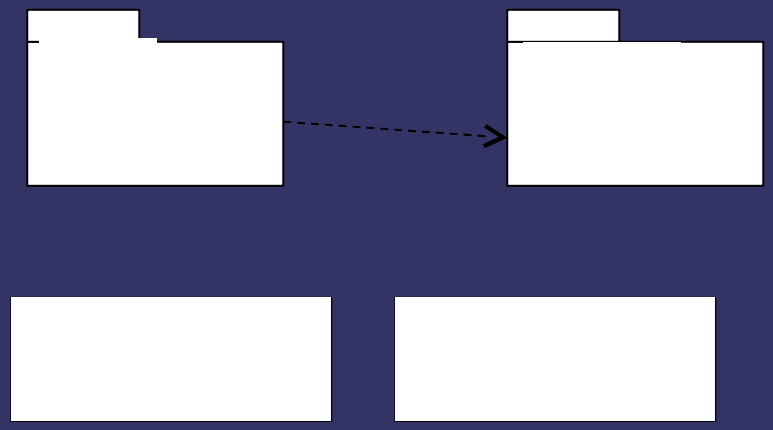
- **Note**
  - commentaire placé sur un diagramme,
  - liens en pointillés
  - pas de sémantique
  - quelques stéréotypes << besoin >>, << responsabilité >>



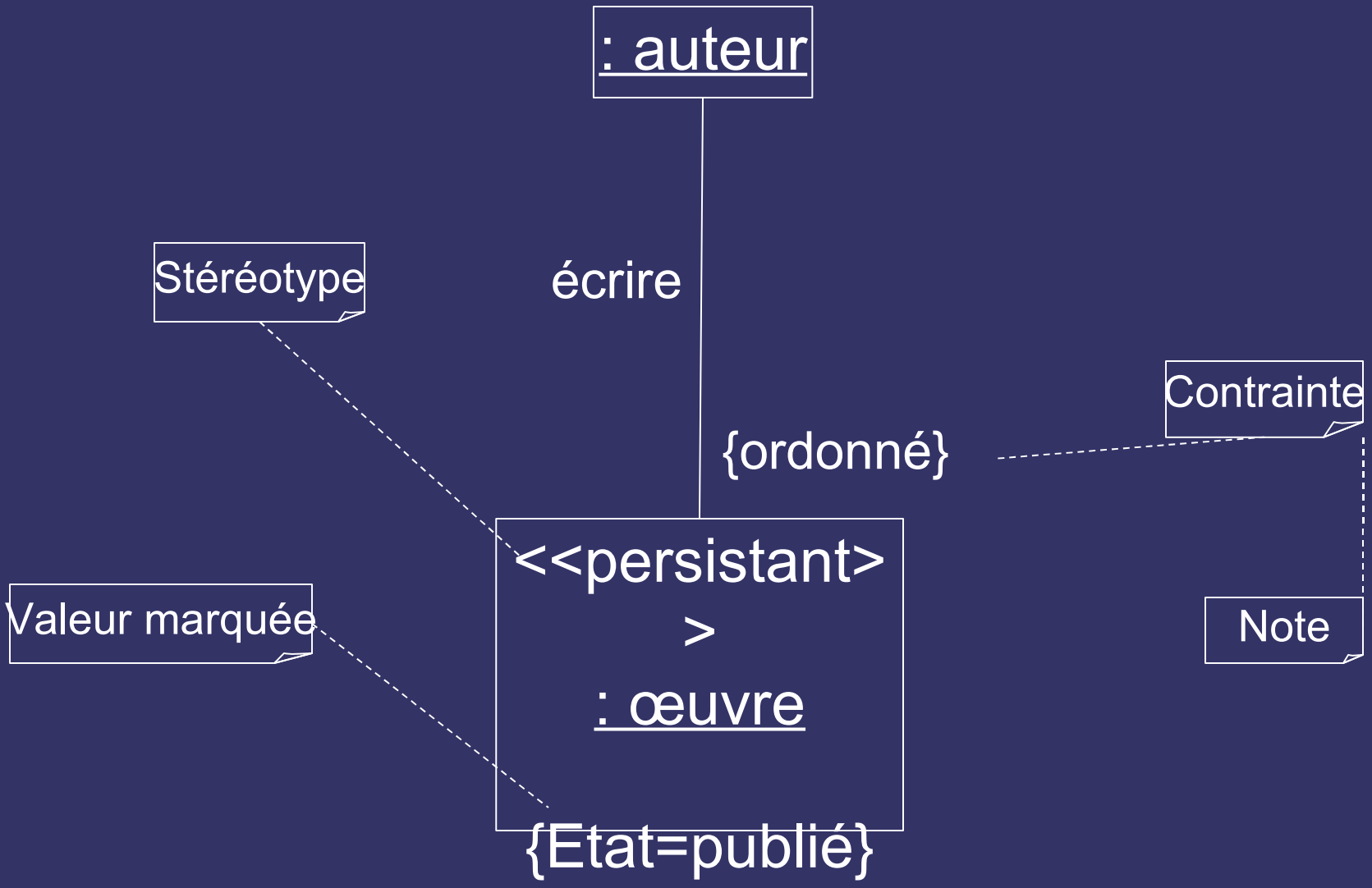
- **Contrainte**
  - relation sémantique quelconque entre 2 ou plusieurs éléments du modèle définissant des propositions devant être maintenues à *Vrai* pour garantir la validité du système modélisé
  - notation : {}, à côté de l'élément concerné.
  - certaines sont prédéfinies (XOR, ordonné).
  - Les autres sont créées par l'utilisateur (langue, pseudo-code, ou OCL, etc.)

Mécanismes de base (3)

- **Dépendance**
  - forme faible de relation sémantique = relation d'utilisation unidirectionnelle entre deux éléments = relation sémantique non structurelle entre client et fournisseur
  - notation : flèche pointillée de l'élément source vers l'élément cible, éventuellement stéréotype
  - pas d'instances (liens)



Mécanismes de base : exemple



III
Diagrammes de classes

Diagrammes de classes

- Vue logique / statique / structurelle d'un système : les classes et leur relations
- Dans UML
  - classes, structures et comportements
  - relations : associations, agrégations, dépendances, généralisation/spécialisation, noms de rôles
  - contraintes
  - indicateurs de multiplicité et de navigation

Classes : généralités

- Collections d'objets avec structure / comportements/ relations/ sémantique communs
- UML
  - rectangle avec trois compartiments :
    - Nom
    - Attributs
    - Opérations
  - représentation plus ou moins détaillée suivant l'étape et l'objectif de l'analyse
- Nom : vocabulaire du domaine, nom au singulier.

  Ex : Fichier, client, compte, chat,…

Attributs

- Constituent la structure de la classe
- Trouvés en examinant les définitions des classes, le problème, et les connaissances du domaine

Visibilité nom [multiplicité] : type = valeur_initiale {propriétés}

public **+**
privé **-**
protégé **#**

Facultatif
Ex:
couleurs [3]
points [2..*]

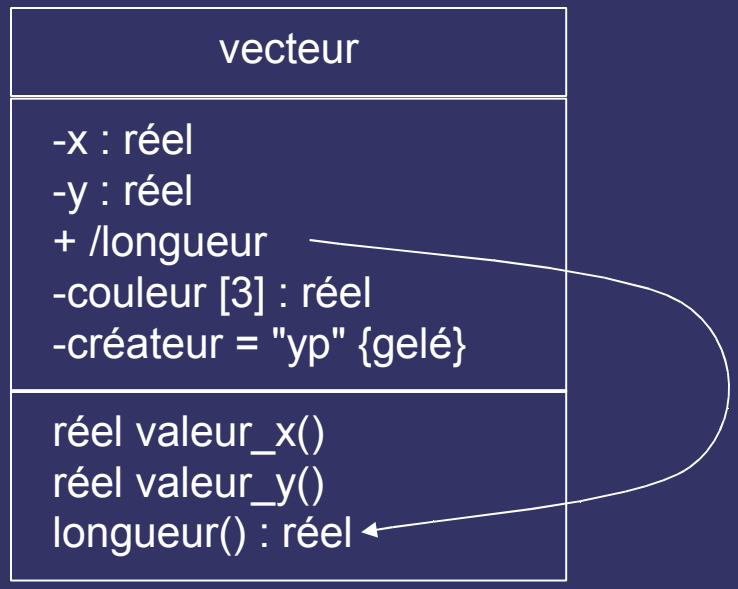
Facultatif
Mais impératif pour l'implantation

Facultatif

Facultatif, Mutabilité
Ex :
{gelé} mise à jour interdite
{obligatoire} valuation oblig.

Attributs

Télévision
-on/off : Bouton -Couleur : enum {gris, noir} - marque : chaine -Télétexte : booléen = vrai - chaines [5…*] : canal -Enceintes[2..6] : haut-parleur -Type : typeTV {gelé} -Volume : parallépipède = (600,650,500)



Attributs dérivés

→ méthodes d'accès

Opérations

- Constituent le comportement d'une classe
- Trouvées en examinant les diagrammes d'interaction
- Rappel : méthode = implantation d'un opération dont elle spécifie l'algorithme ou la procédure associée

visibilité nom (liste_d_arguments) : type_retour {propriétés}

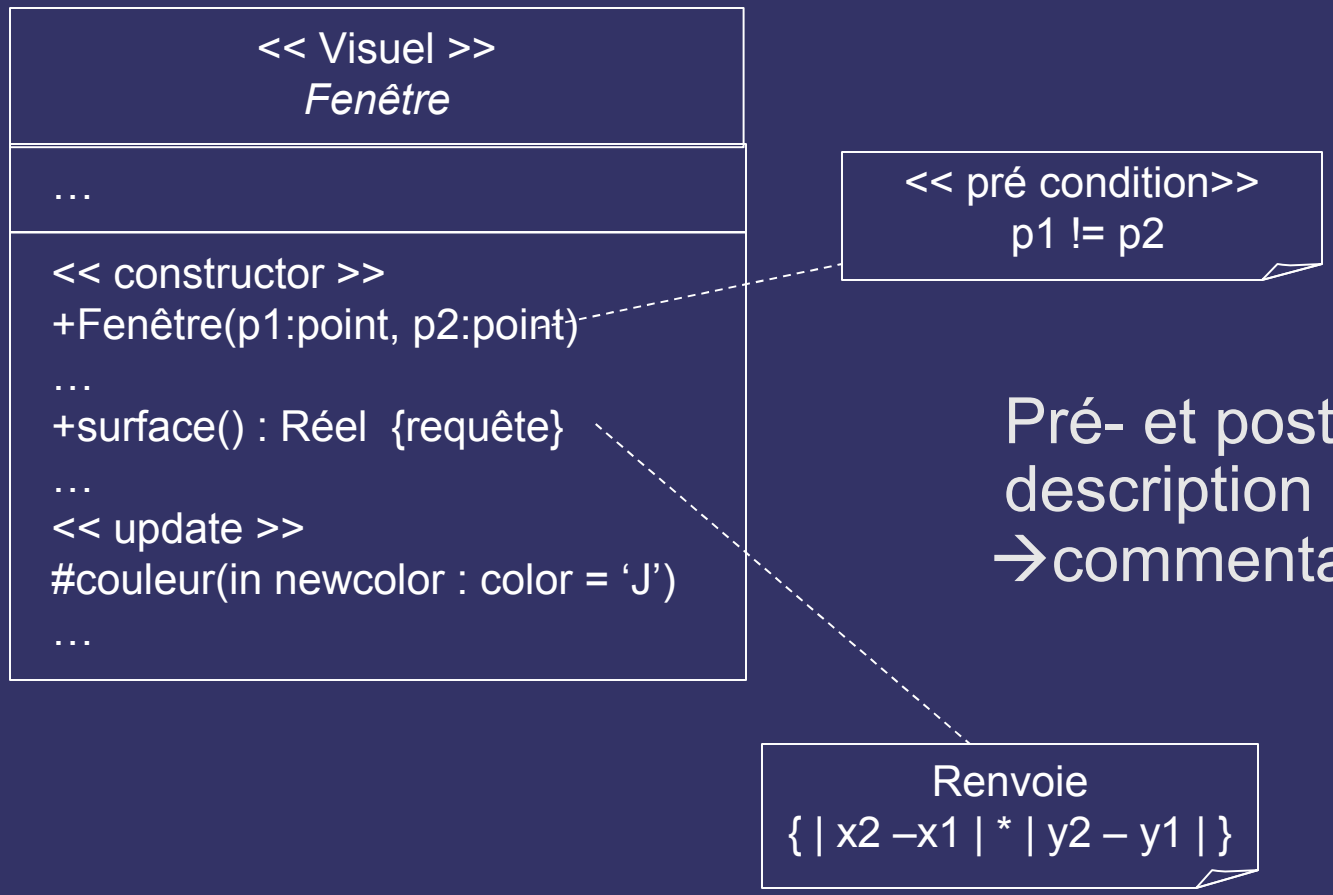
argument ::= genre nom_arg : type = val_défaut

public **+**
privé **-**
protégé **#**

in | out | inout

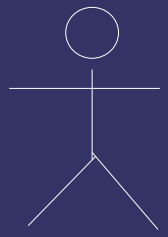
<< abstrait >>
<< requête >>
…

Opérations



Pré- et post-conditions, description du contenu
→commentaires + OCL

Exemples de classes



<< acteur >>
toto

Niveaux
de description

Fenêtre

<< Visuel >>
Fenêtre

forme : zone
visibilité : booléen

afficher()
masquer()

<< Visuel >>
Fenêtre
{ abstract,
auteur = yp,
statut = testé }

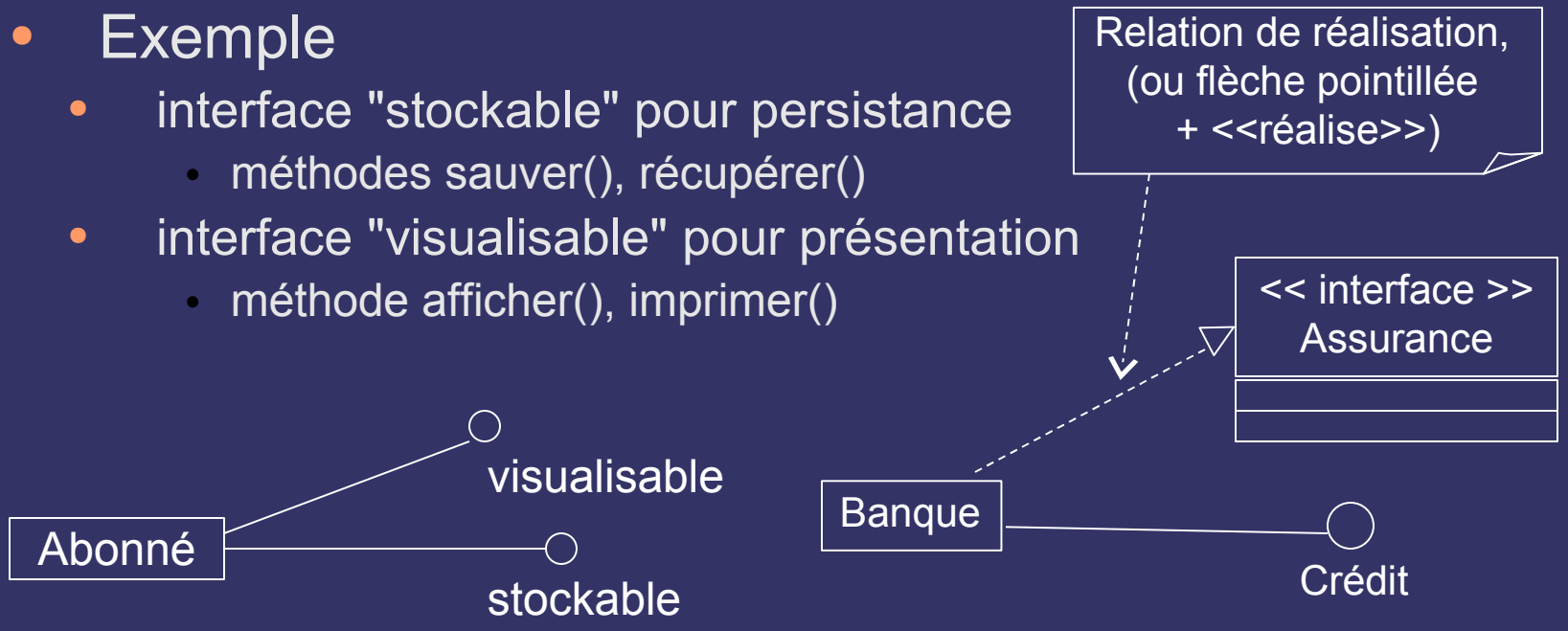
+forme : zone = {100,100}
#visibilité : booléen = faux
+forme_défaut: rectangle
-xptr : Xwindow

+afficher()
+masquer()
+créer
-attachXWindow(xwin : Xwindow)

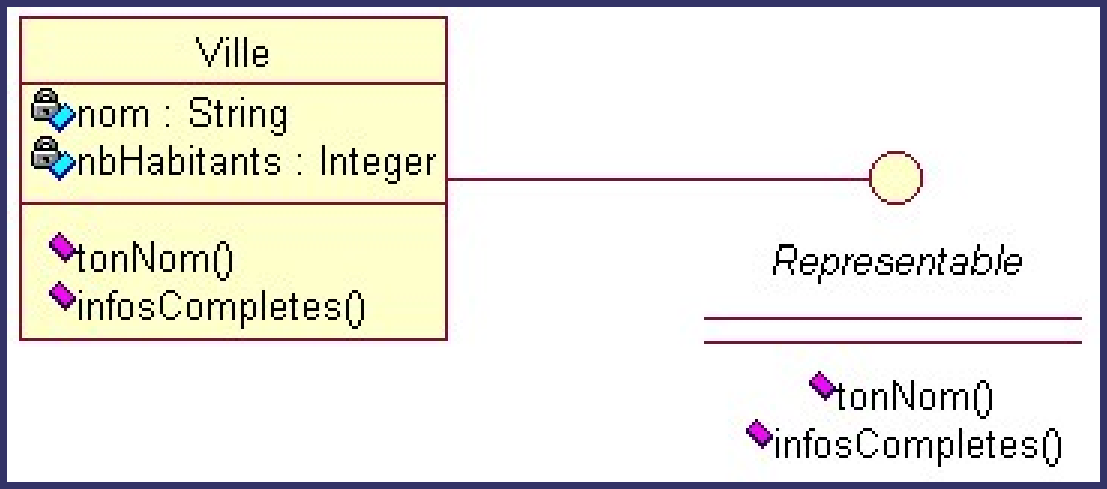
(pour l'implantation)

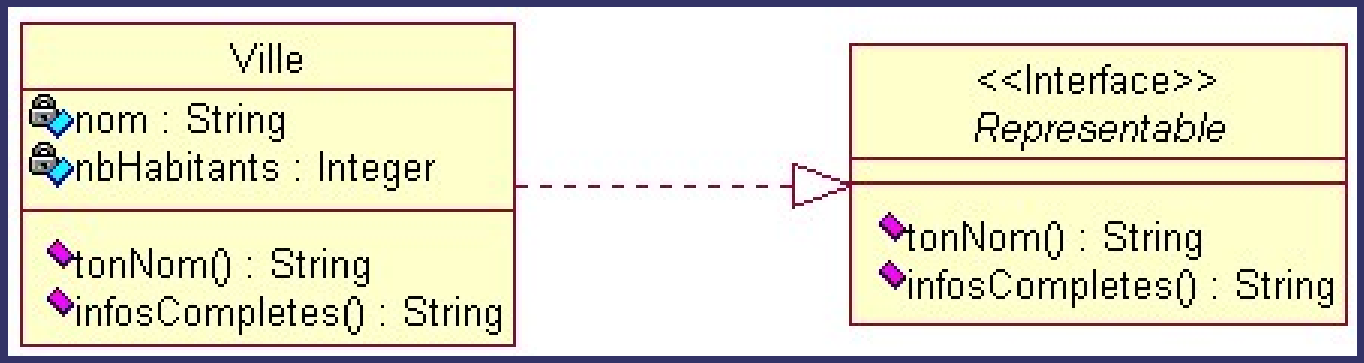
Interfaces de classes

- Facette d'un objet : comportement visible, *service* fourni par une classe
- Liste d'opérations (signatures de méthodes, public)
- Classe : *réalisation* d'une interface
- Exemple
  - interface "stockable" pour persistance
    - méthodes sauver(), récupérer()
  - interface "visualisable" pour présentation
    - méthode afficher(), imprimer()

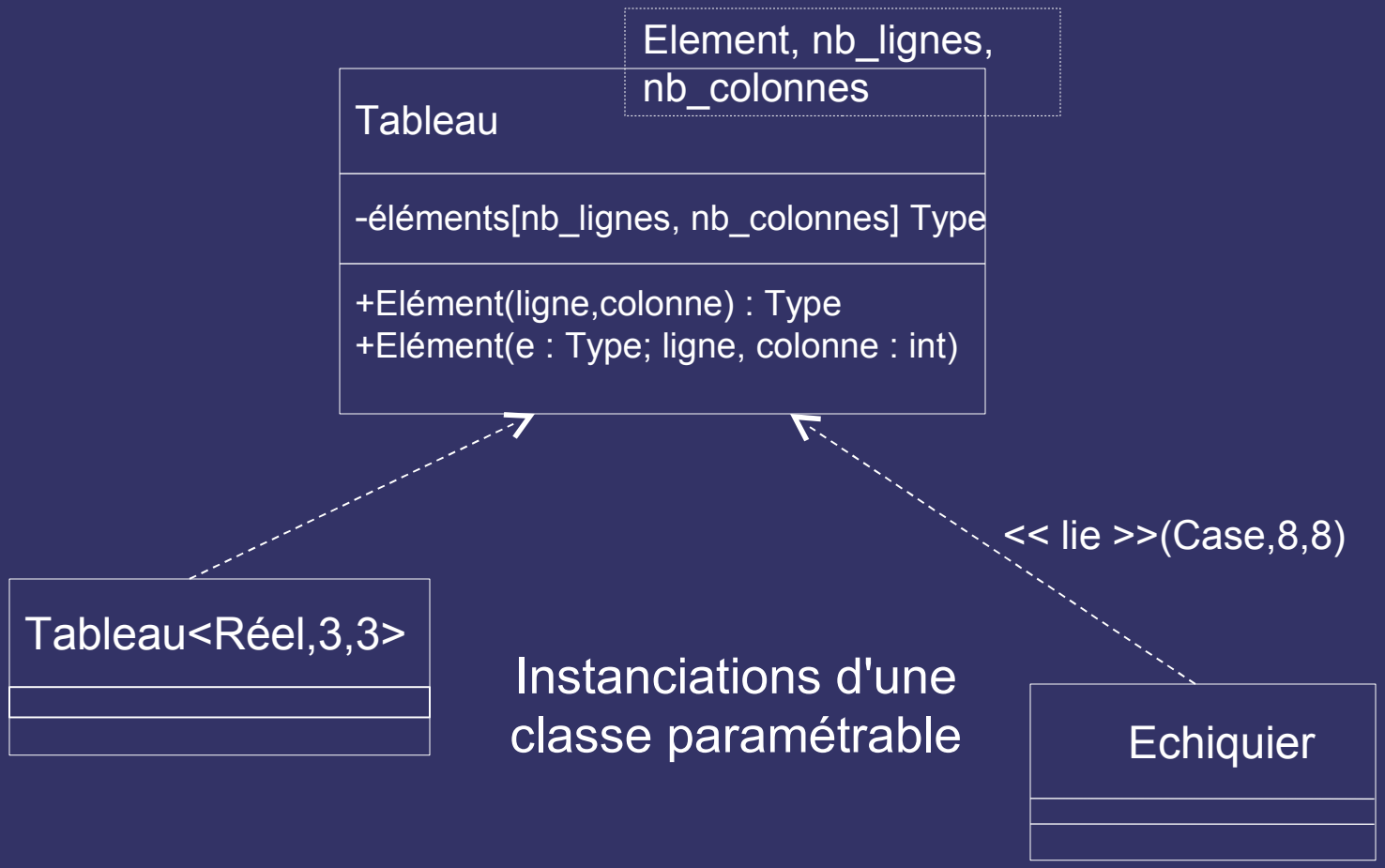


Interface


Ville
nom : String
nbHabitants : Integer
tonNom()
infosCompletes()
Representable
tonNom()
infosCompletes()


Ville
nom : String
nbHabitants : Integer
tonNom() : String
infosCompletes() : String
<<Interface>>
Representable
tonNom() : String
infosCompletes() : String

Classes paramétrables



Liens et relations

- *Liens* entre objets / *relations* entre classes
- Possibilité de communication entre objets :

  interactions entre objets (cf. diagrammes)
  → relation entre classes
- 3 grands types
  - *associations* ("est produit par", "est affilié à", "se trouve à", "est conduit par"…) : dépendance entre classes, communication entre objets

    Ex: « un lecteur lit un livre » (forme verbale)
  - *agrégation/composition* ("fait partie de") : objets composites / composants

    Ex : « un train est composé de wagons »
  - *généralisation* ("est une sorte de") : héritage entre objets

    Ex : « un chat est une sorte d'animal »

Relations : multiplicité et navigation

- Multiplicité : combien d'objets sont impliqués dans une relation (= contrainte)
  - nombre d'instances d'une classe en relation avec une instance d'une autre classe
  - pour chaque association/agrégation il y deux décisions à prendre : une pour chaque extrémité
- Bi-directionnalité par défaut, mais souvent nécessité de restreindre la navigation à une direction

  → on rajoute une flèche indiquant la direction

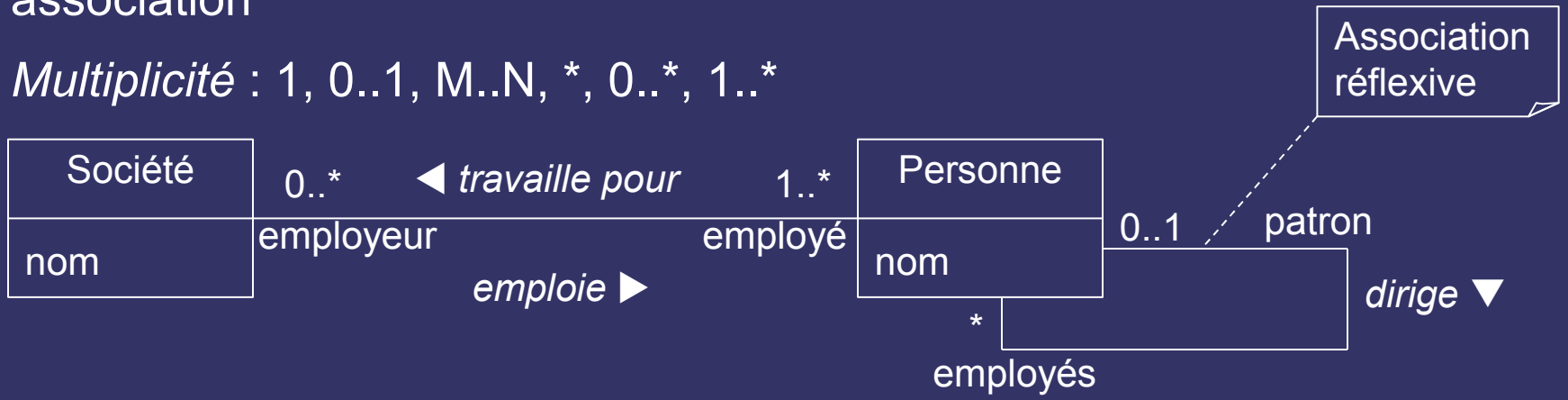
Associations

- Connexion bidirectionnelle entre classes
- Notation générale :

Nom : forme verbale, sens de lecture avec flèche

Rôles : forme nominale, identification extrémité association

Multiplicité : 1, 0..1, M..N, *, 0..*, 1..*

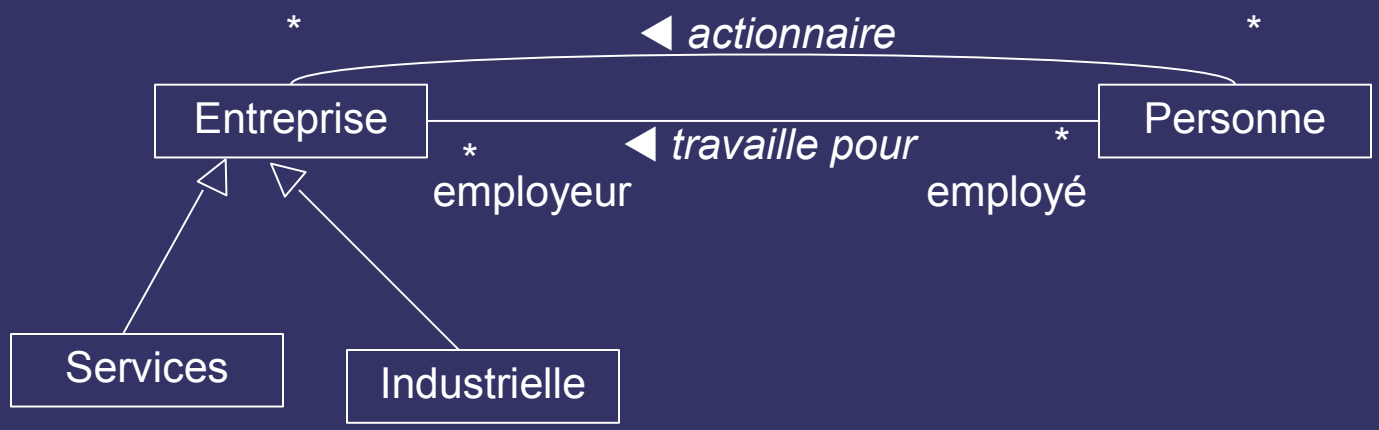


Associations : remarques

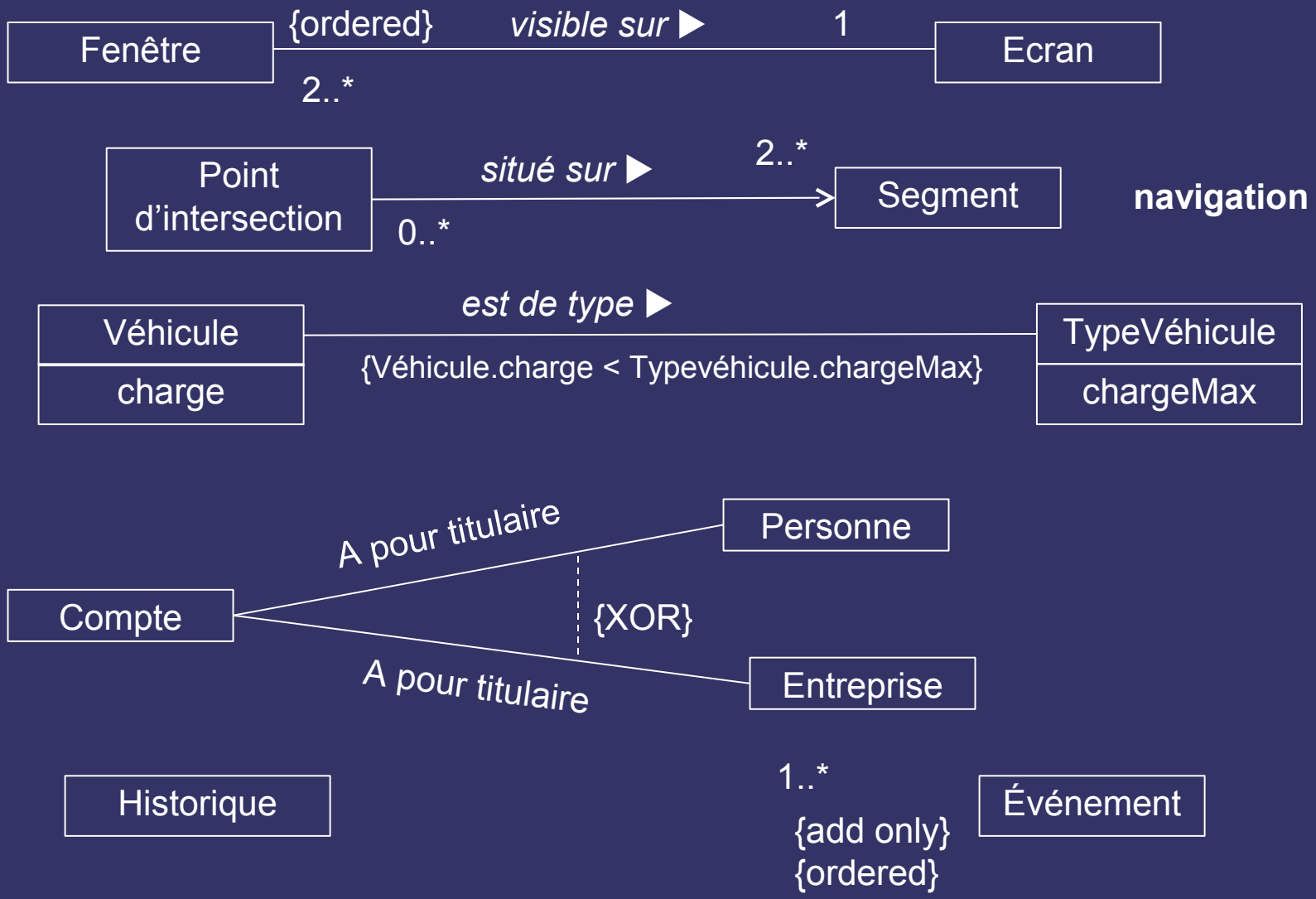
- Multiplicité et implantation :

  la multiplicité influe sur le choix des structures de données :
  0..n → test vide, 0..* → ensemble, …

- Tout objet doit être accessible *via* une association !
- Les associations ont une durée de vie, sont indépendantes, sont héritées.

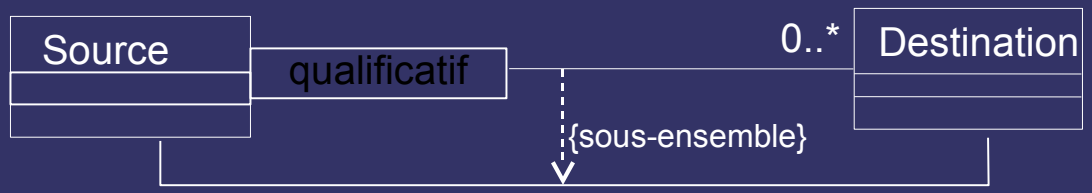


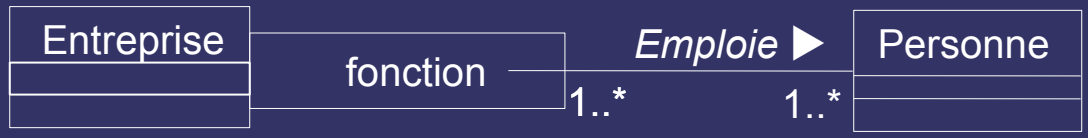
Différentes contraintes



Associations qualifiées

- Restrictions : sélection d'un sous-ensemble des objets qui participent à l'association à l'aide d'une clé. Cet attribut est propriété de l'association



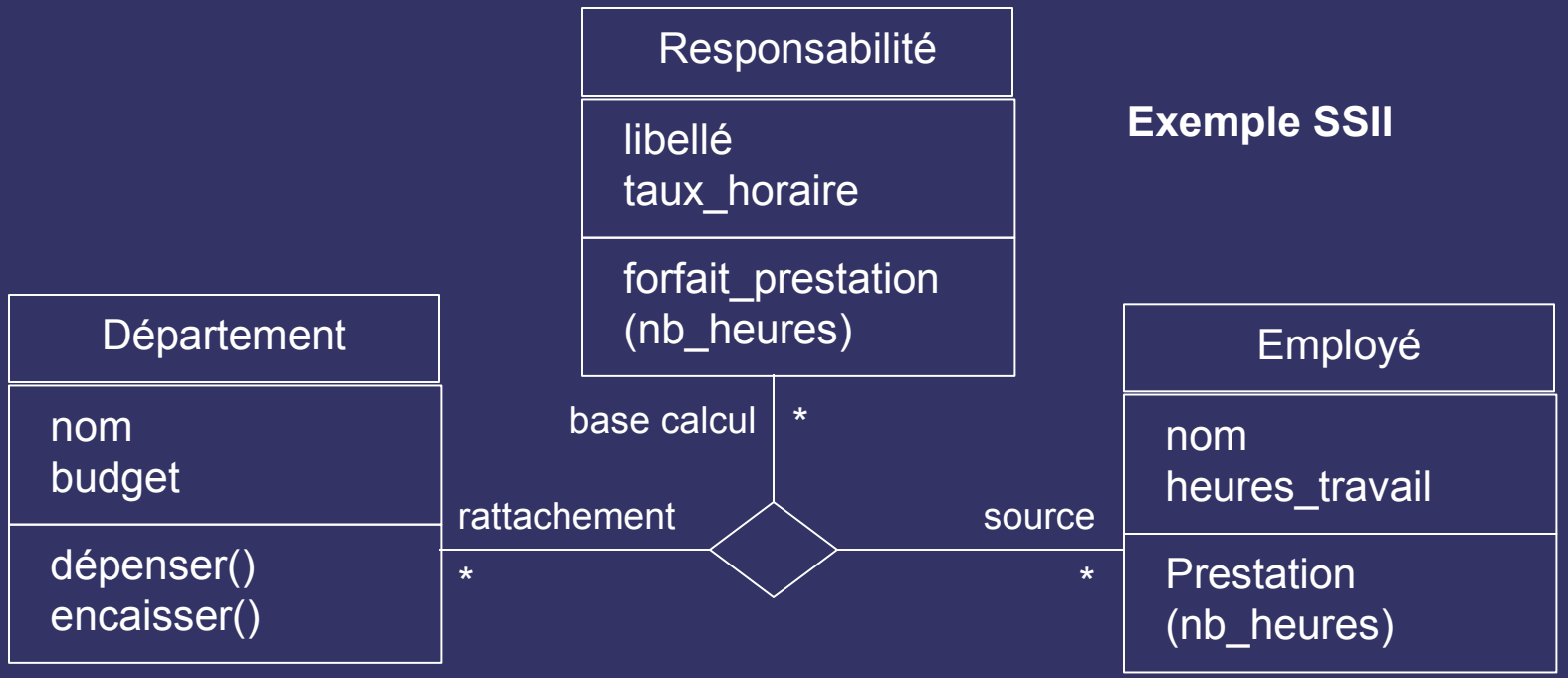


Remarque :

cardinalité

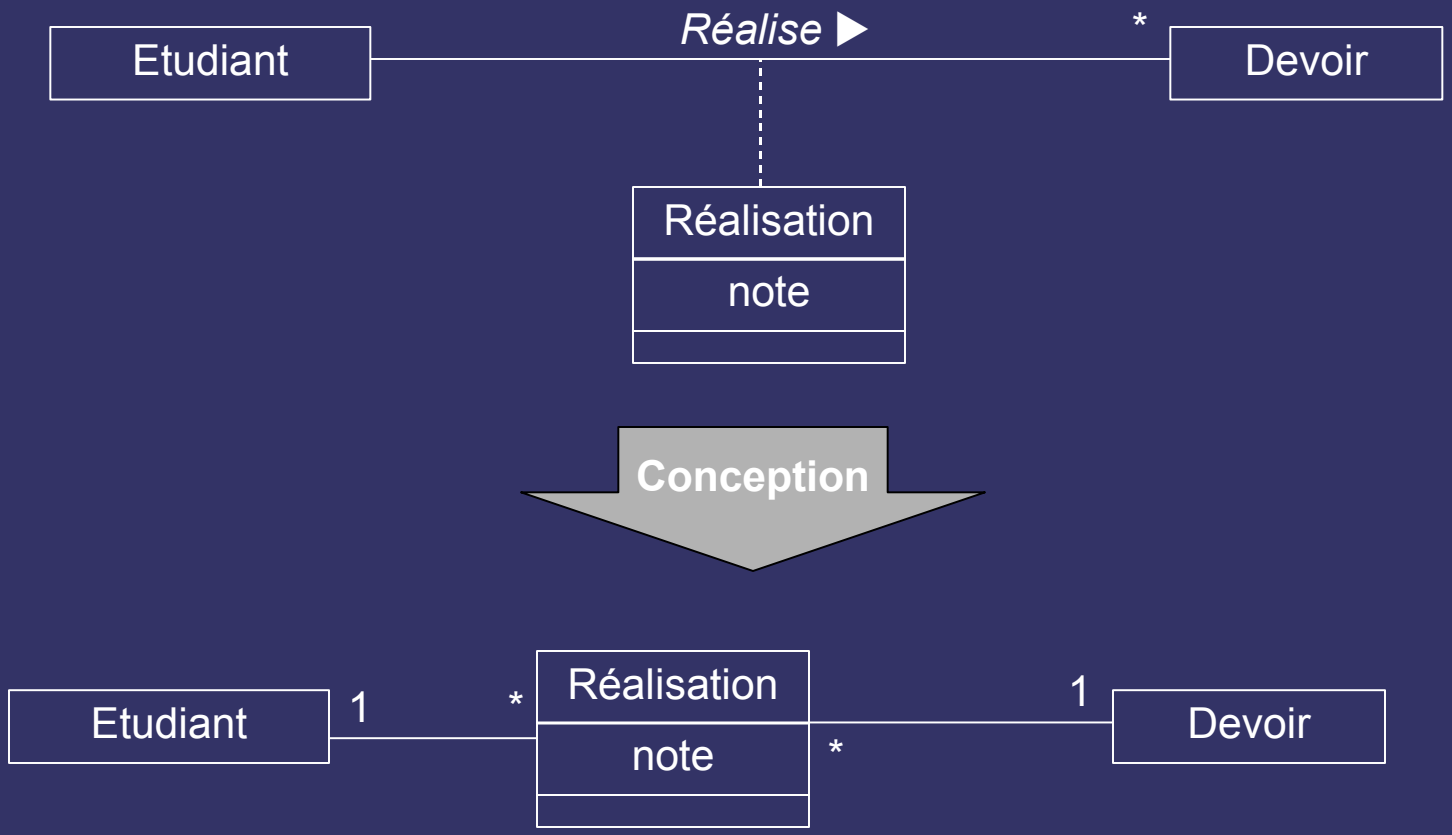
Association n-aire

- Groupe de liens entre au moins trois instances
- Instance de l'association = n-uplet des attributs des instances impliquées



Classe association

- Association avec des attributs et des opérations

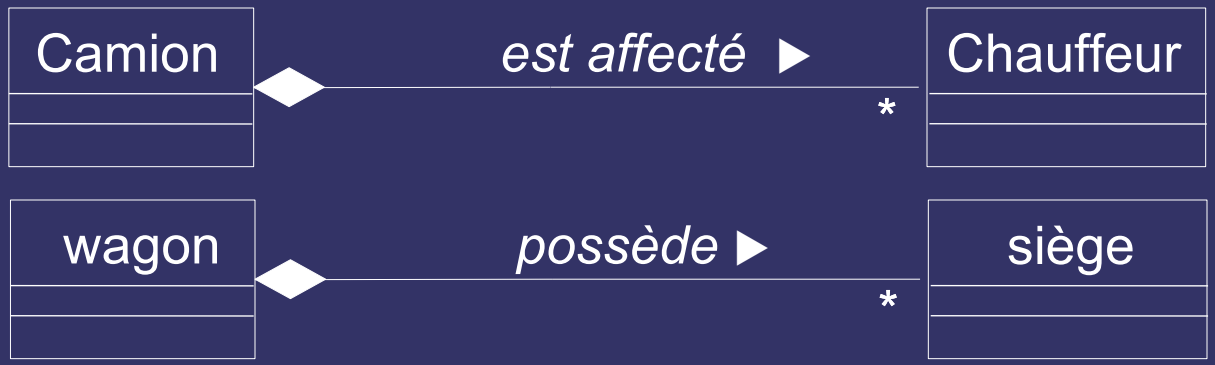


Agrégation

- Forme d'association plus forte, antisymétrique

→ rôle prépondérant d'une extrémité par rapport à l'autre

tout/partie - composant/composé - maître/esclave



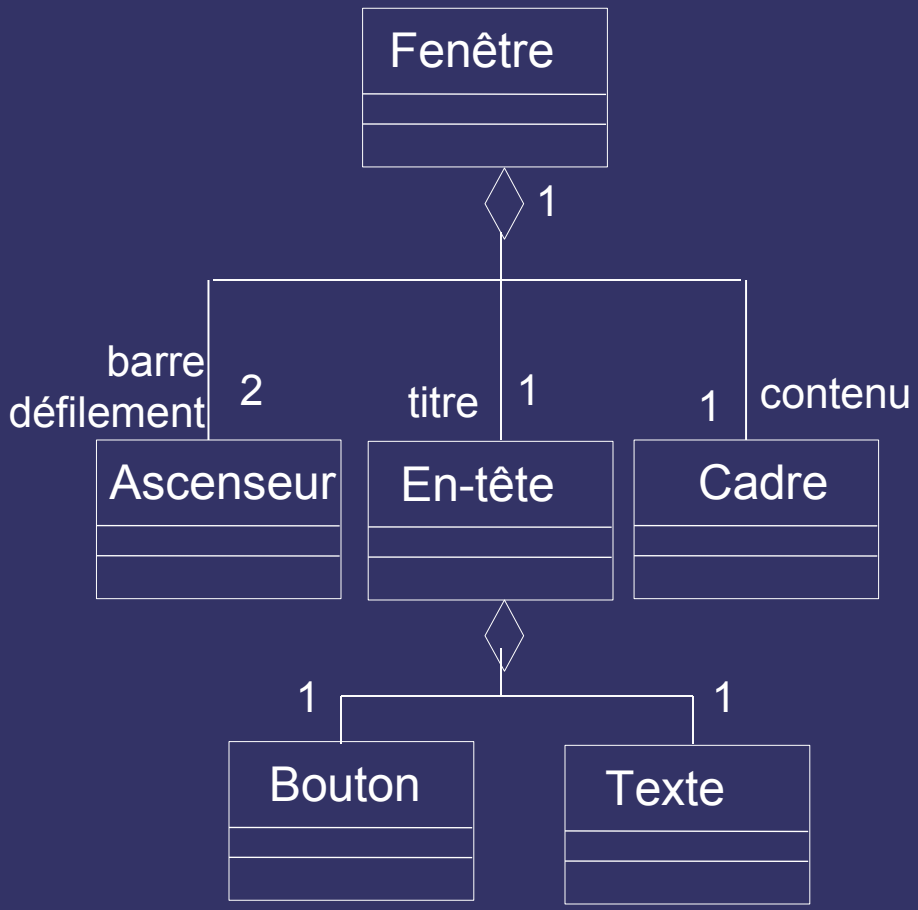
- Utilisation de la structure d'arbre sous-jacente
  - propagation de valeurs (ex. document / page : format portrait)
  - délégation d'opération (ex. *dessine* appliqué aux sous-composants d'une forme graphique)

Choisir entre agrégation et association

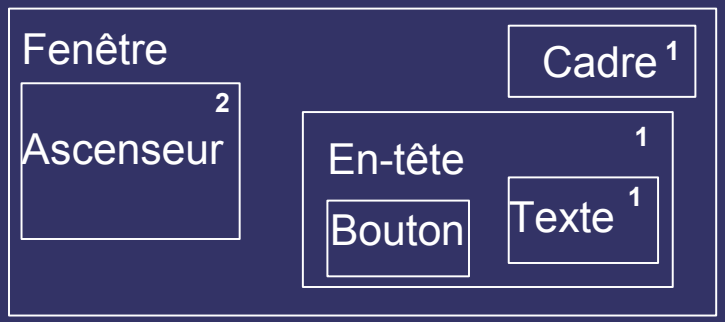
- Y a t'il asymétrie intrinsèque, avec lien de subordination entre instances des deux classes ? (agrégation)
- Ou au contraire, indépendance des objets ? (association)
- Existe-t-il des opérations du tout qui s'appliquent aux parties, des valeurs d'attribut du tout qui se propagent vers les parties
- Peut-on dire « X est partie de Y » ?
- Dans le doute : association (moins contraint)

Composition d'objets

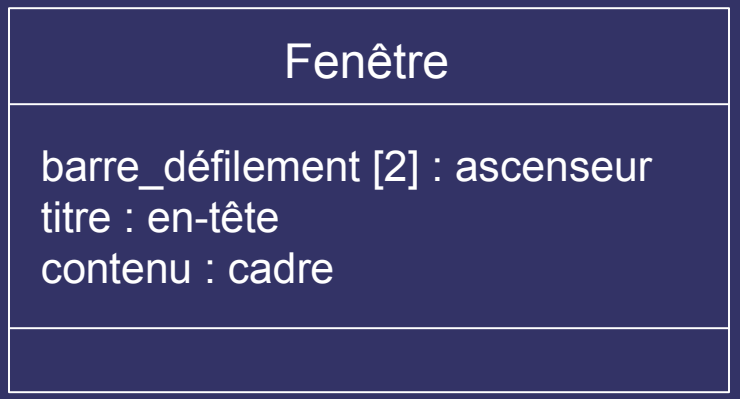
- Cas particulier d'agrégation = contenance physique
- Si destruction objet composé, destruction des composant



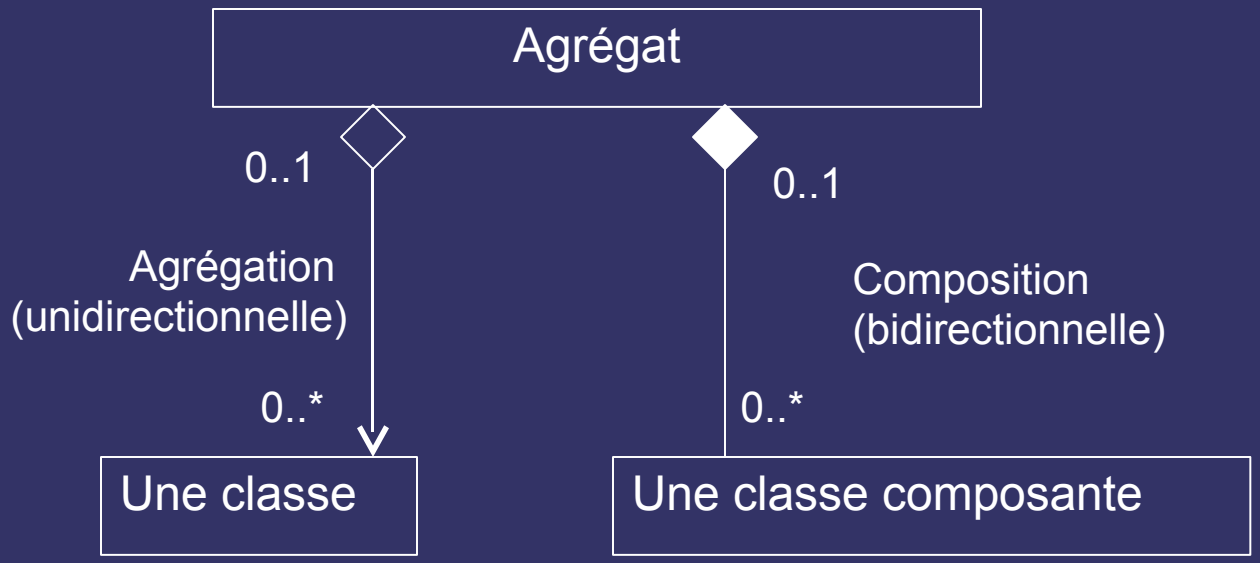
- Représentation emboîtée



- Identité sémantique entre composition et attributs

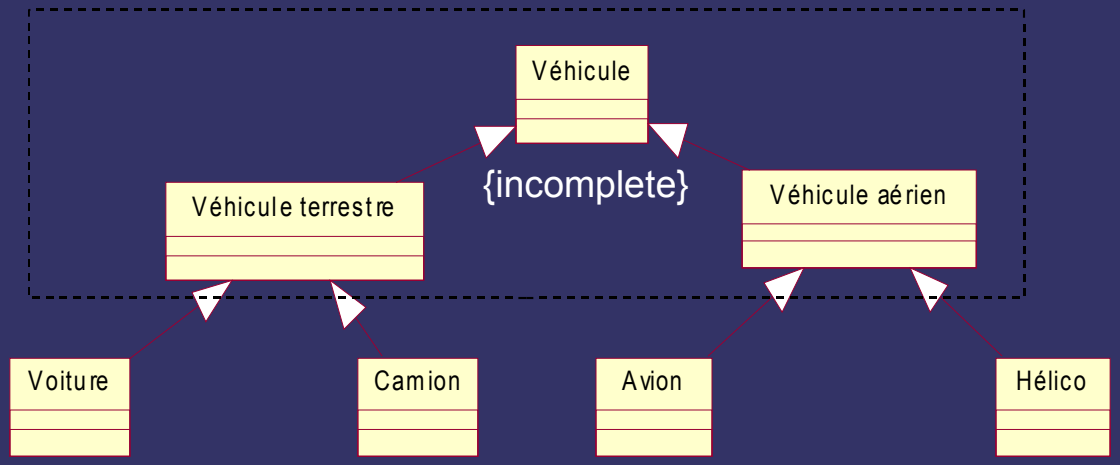


En résumé


Agrégat
0..1
Agrégation
(unidirectionnelle)
0..*
Une classe
0..1
Composition
(bidirectionnelle)
0..*
Une classe composante

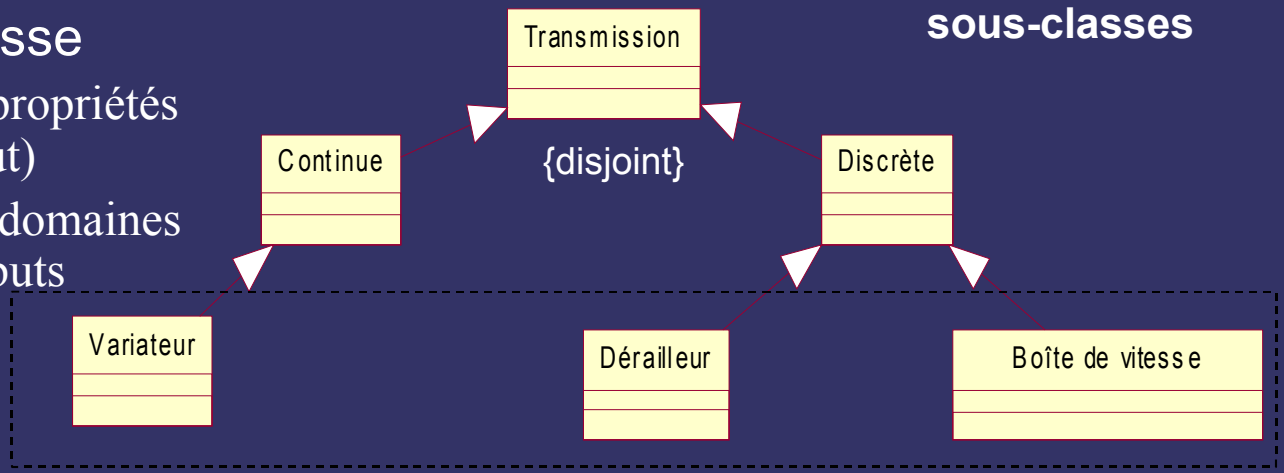
Généralisation et spécialisation

- *Généralisation* : regroupement, « est-un », « sorte-de »



- *Spécialisation* (symétrique généralisation): raffinement de classe
  - par extension de propriétés (ajouter un attribut)
  - par restriction de domaines de valeurs d'attributs

super-classes
sous-classes

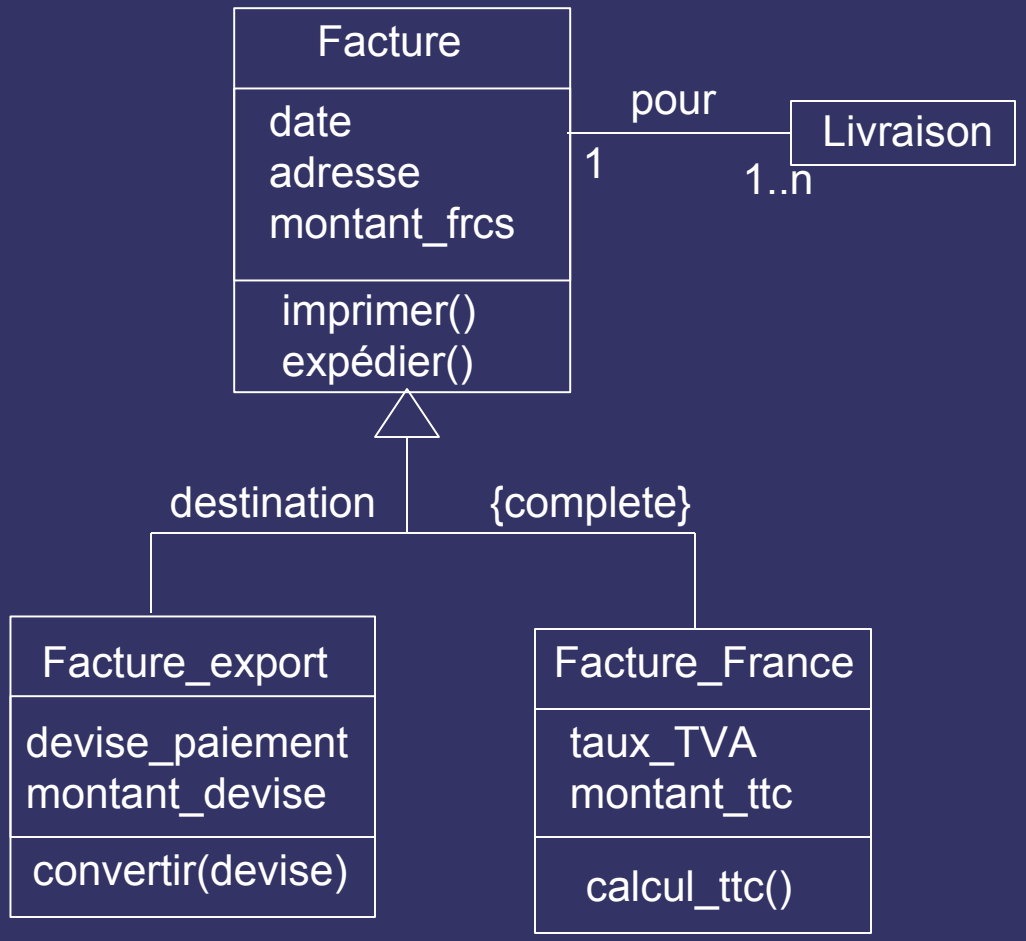


L'héritage

- Moyen de réaliser la classification ou l'organisation en classes → vocabulaire à réserver à l'implantation
- Principe de substitution : toutes les propriétés de la classe parent doivent être valables pour les classes enfant → véritable classification
- Possibilité d'héritage d'implantation (<<implantation>>)

Hiérarchies de classes

- Contraintes : {complete}, {incomplete}, {disjoint}, {overlapping}
- Discriminant
- Attributs / opérations communes, relations communes sont montrées au niveau le plus haut.



Exemple de structure récursive

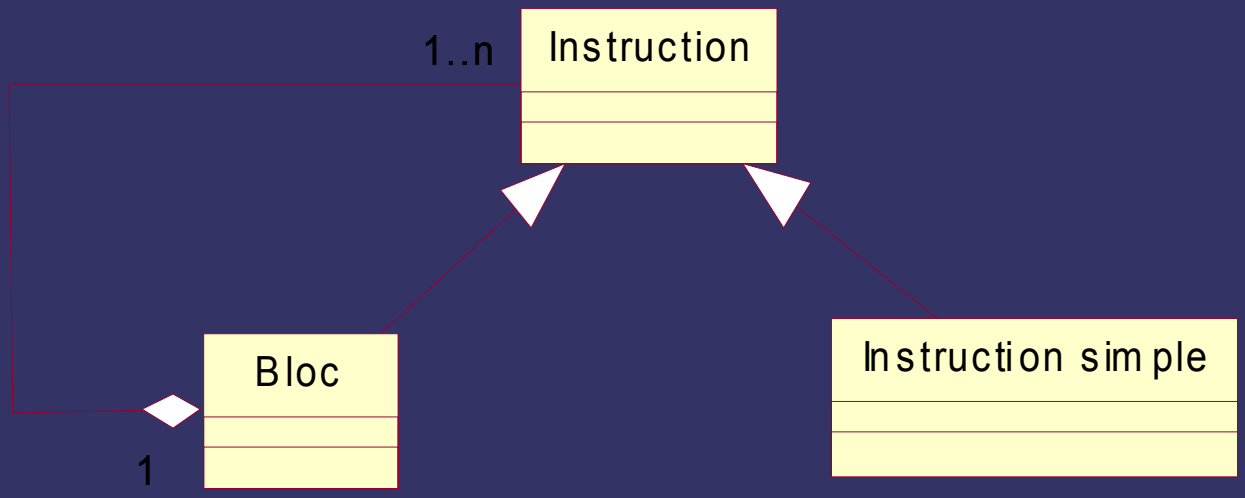
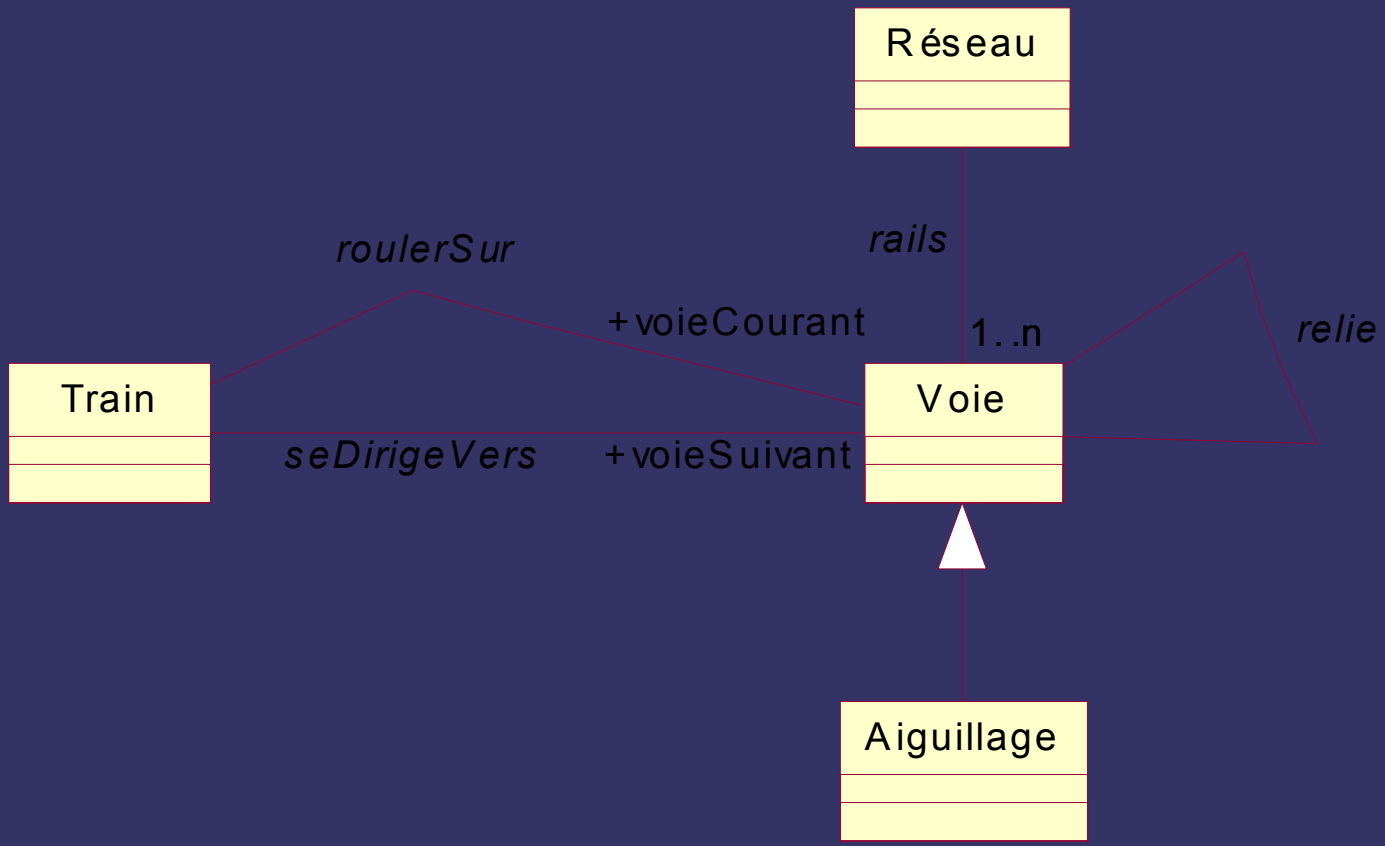


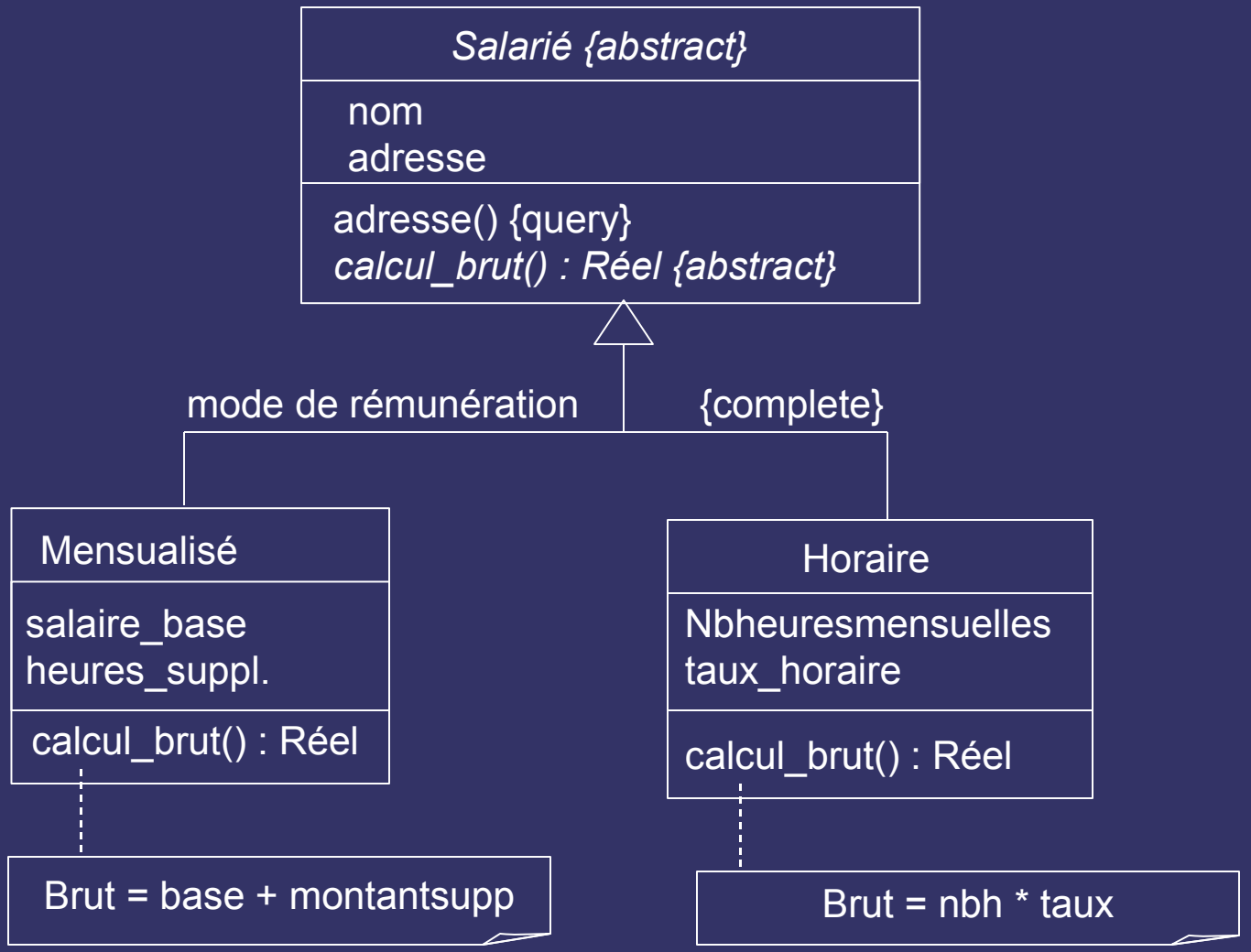
Diagramme de classe du train



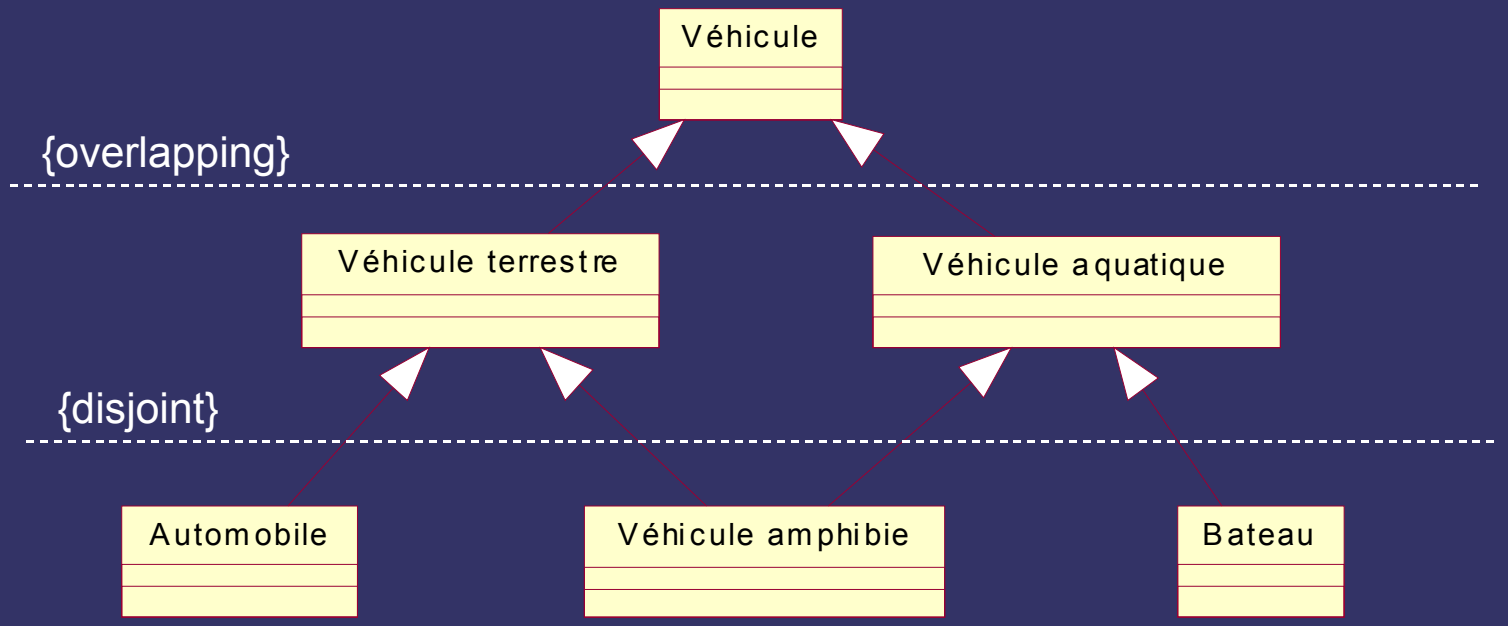
Classes abstraites

- Classes concrète = classe instanciable
  → toutes les feuilles d'un arbre de généralisation sont concrètes.
- Classe abstraite = classe non instanciable
  - regroupement de propriétés (attributs, opérations, associations)
  - spécification pour les sous-classes
  - italiques dans le nom + evt. { abstrait }

Classes abstraites



Généralisation multiple (1)



- Accumuler les caractéristiques des super-classes

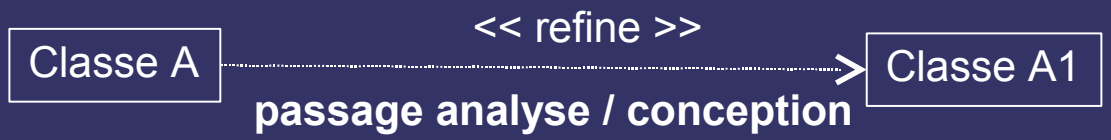
Généralisation multiple (2)

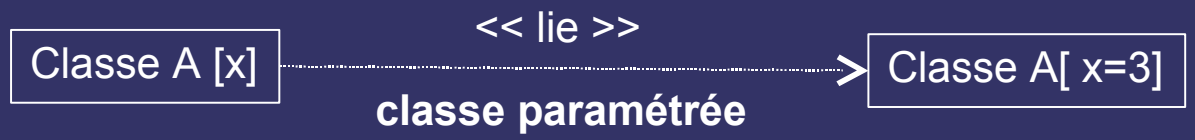
- Problème : conflits d'héritage (un attribut vitesse_max en km/h, en Nœuds → qu'hériter ?)
- En général, très difficile à utiliser : certains langages négligent l'héritage multiple
  - apporte plus de problèmes qu'il n'en résout
  - mieux vaut catégoriser sans recouvrement possible

    → utiliser la *délégation*
- Remarque : possibilité d'héritages d'analyse et d'implantation

Relations de dépendance

- 4 types
  - Abstraction : différents niveaux d'abstraction (ex. raffine, trace)
  - Liaison : classes paramétrées (ex. lie)
  - Permission : accès (ex. ami)
  - Utilisation : nécessité d'un élément cible (ex. utilise, appelle, crée)





Stéréotypes de classes

- Représenter des décisions générales d'implantation

<< type >> : spécification abstraite des opérations applicables à un domaine d'objets.

Ex. << type >> Ensemble avec ajouter_elt(), enlever_elt()

<< implementation class >> : mode de réalisation

- de la structure physique des données : attributs / associations
- des méthodes implantées dans un langage

Ex. << implementation class >> EnsembleTableau avec ajouter_elt(), enlever_elt(), taille(),

Une classe d'implantation *réalise* un type

Stéréotypes de classes

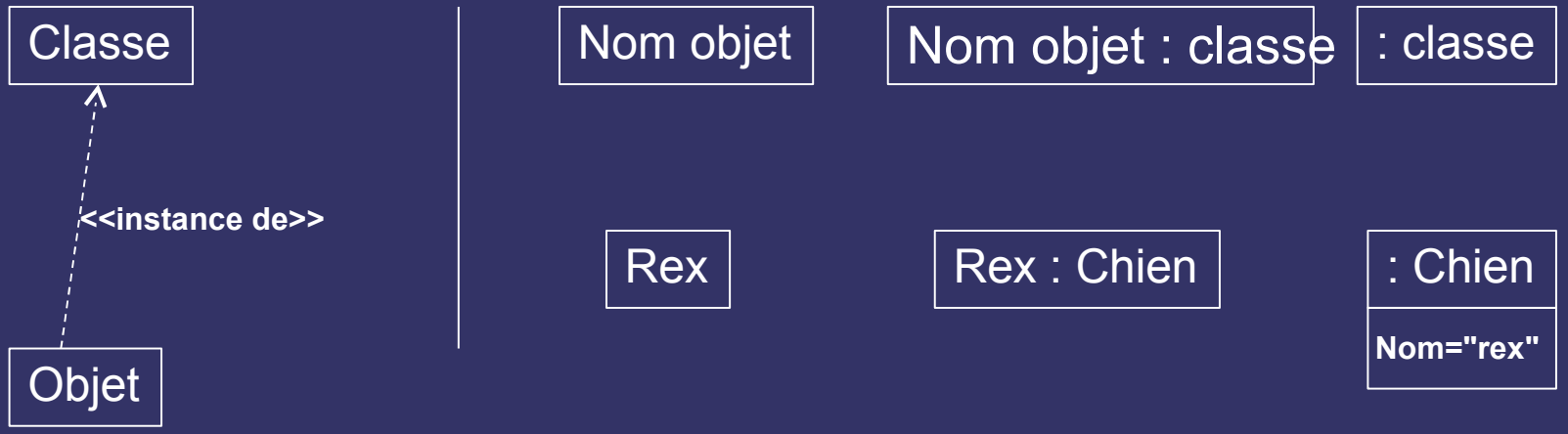
<< interface >> : spécification des opérations visibles par l'environnement d'une classe : une partie seulement des opérations.

Ex. << interface >> Puit avec ajouter_elt() en relation de dépendance << realize >> avec Ensemble

<< utility >> : bibliothèque de variables globales et de fonctions utiles

Ex. << utility >> maths avec sin(), cos(), random()

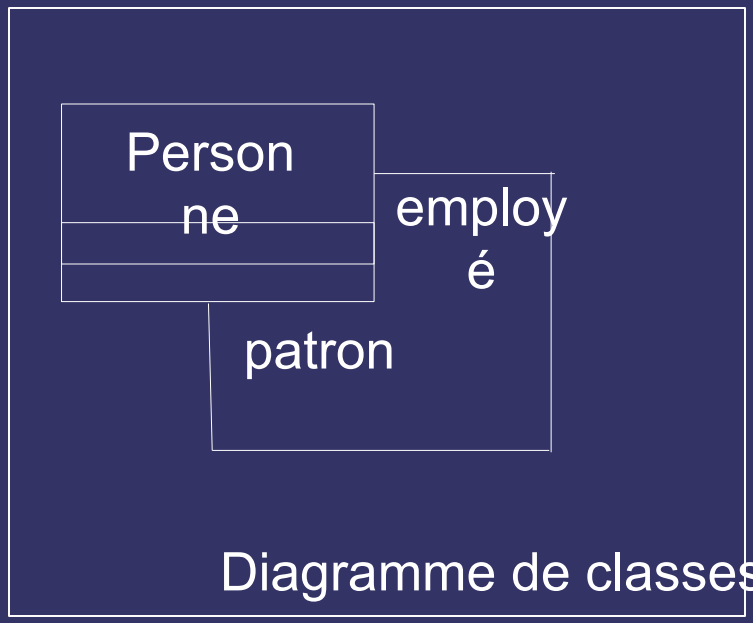
Objets

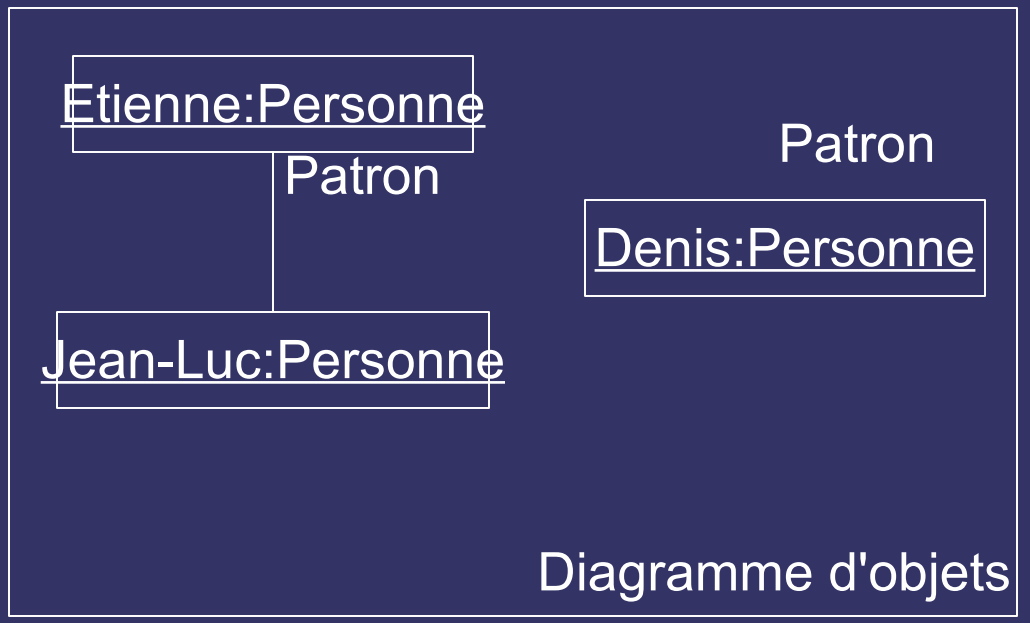


Groupe d'objets

Diagramme d'objets

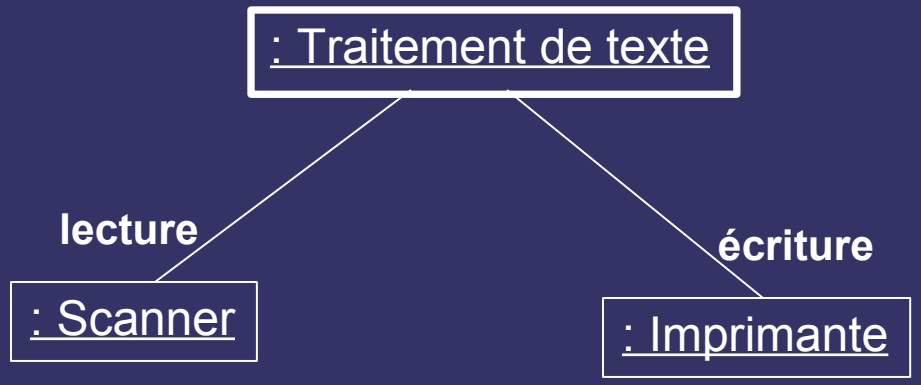
- dérivé du diagramme de classes
  - montrer un contexte
  - faciliter la compréhension des structures complexes (récursivité, associations ternaires…)





Objets actifs

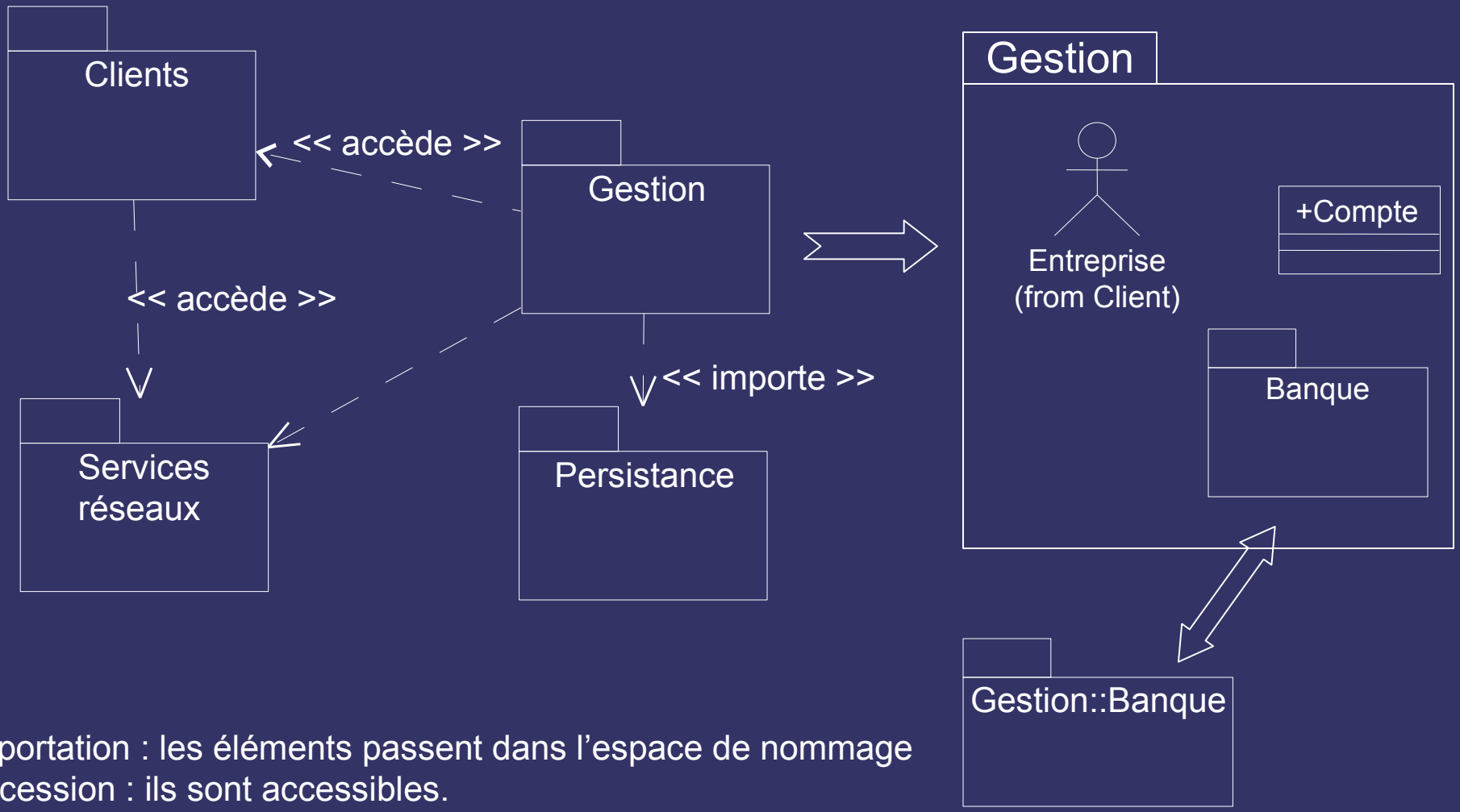
- Liés à des *classes actives*
- Possèdent un flot de contrôle, activent d'autres objets et récupèrent le contrôle
- Environnement multitâche : plusieurs objets actifs (ex. client / serveur)



Paquetage

- Objectif
  - organisation générale (partition/hiérarchie) des éléments de modélisation
  - *par* regroupement logique d'éléments de diagramme qui entretiennent entre eux des relations étroites
  - *pour* clarté / partage du travail dans une équipe
- Un paquetage
  - contient des éléments de modélisation (notamment des classes) avec des liaisons fortes entres eux
  - peut en importer (ex. Véhicule::Voiture # classe voiture dans le paquetage Véhicule)
  - peut avoir des interfaces
- Forme générale du système
  - paquetages # dossiers
  - hiérarchie de paquetage
  - relations de dépendances entre paquetages (<< importe >>, << accède >>)

Exemple

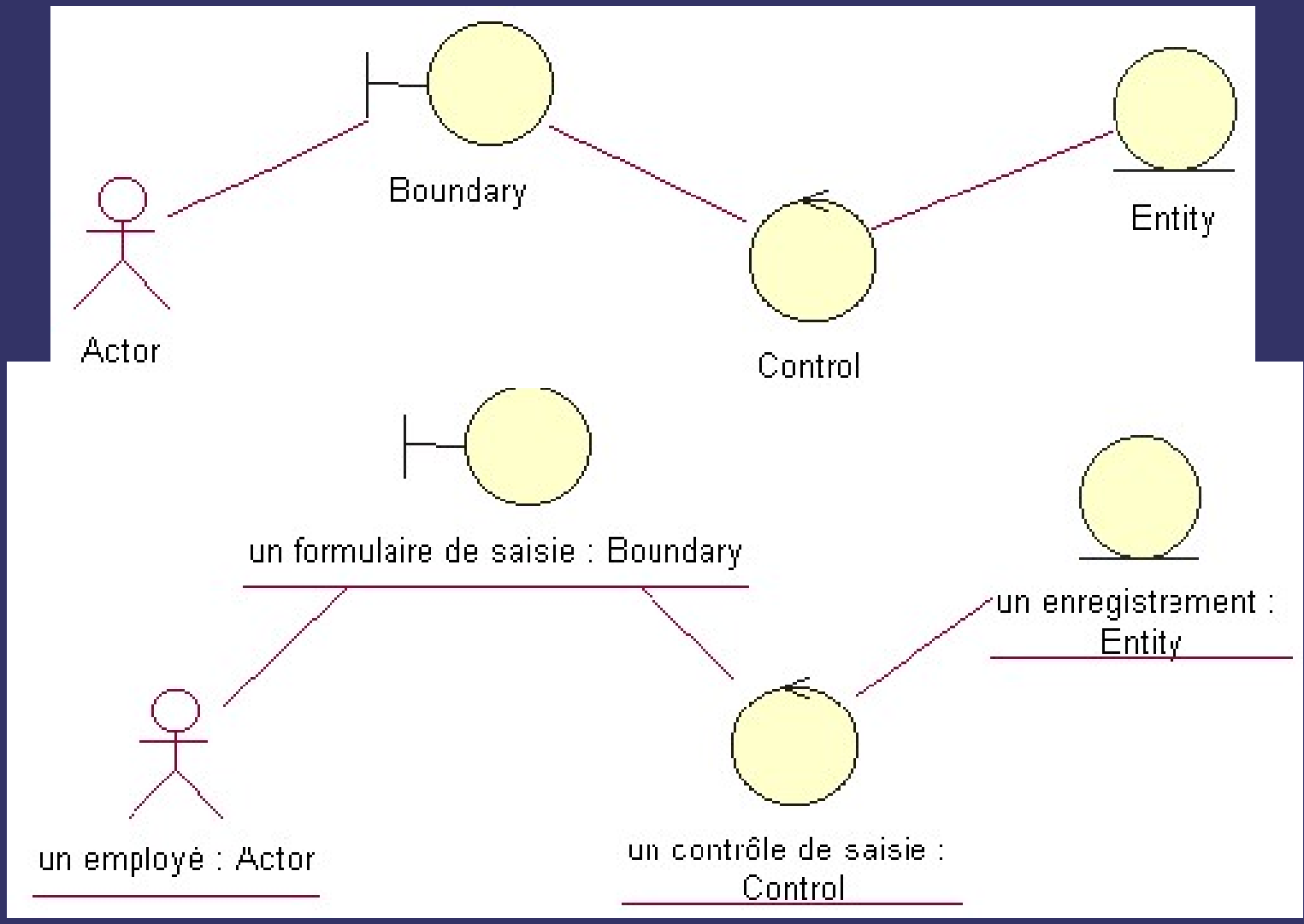


Importation : les éléments passent dans l'espace de nommage
Accession : ils sont accessibles.

Dépendances et paquetages

- Quelques règles
  - Pas de cycle de dépendances
  - Minimiser les dépendances
  - Plus de dépendances entrantes dans un paquetage, plus celui-ci doit être stable
  - Paquetages utilisés partout : mot-clé << global >>

Analyse frontière / contrôle / entité



IV
Cas d'utilisation

Pourquoi les cas d'utilisation

- Passer du flou du cahier des charges à des fonctionnalités exprimées dans un langage du domaine
- Dialogue entre concepteurs et utilisateurs pour exprimer les besoins *réels* du système (phase d'analyse des besoins) en modélisant ce qu'il se passe *du point de vue de l'utilisateur*
- Fonctionnalités du système déclenchées par acteur externe
- Exemple : accès au courriel UFR par le web

A l'extérieur du système : les acteurs

- Acteur : entité située *hors* du système et qui interagit avec lui en jouant un *rôle*, qui déclenche toujours un stimulus initial permettant d'avoir une réaction du système
- Déterminer les acteurs permet de préciser les limites du SI
- 4 catégories d'acteurs
  - acteurs principaux ou utilisateurs des fonctions principales du système. Ex. Distributeur Bancaire : *client*
  - acteurs secondaires qui effectuent des tâches administratives ou de maintenance. Ex. DB : *technicien maintenance*
  - matériel externe ou dispositifs incontournables faisant partie du domaine de l'application. Ex. DB : *imprimante*
  - autres systèmes avec lesquels le système doit interagir. Ex. DB : *Ordinateur central banque*

Acteur/rôles en UML

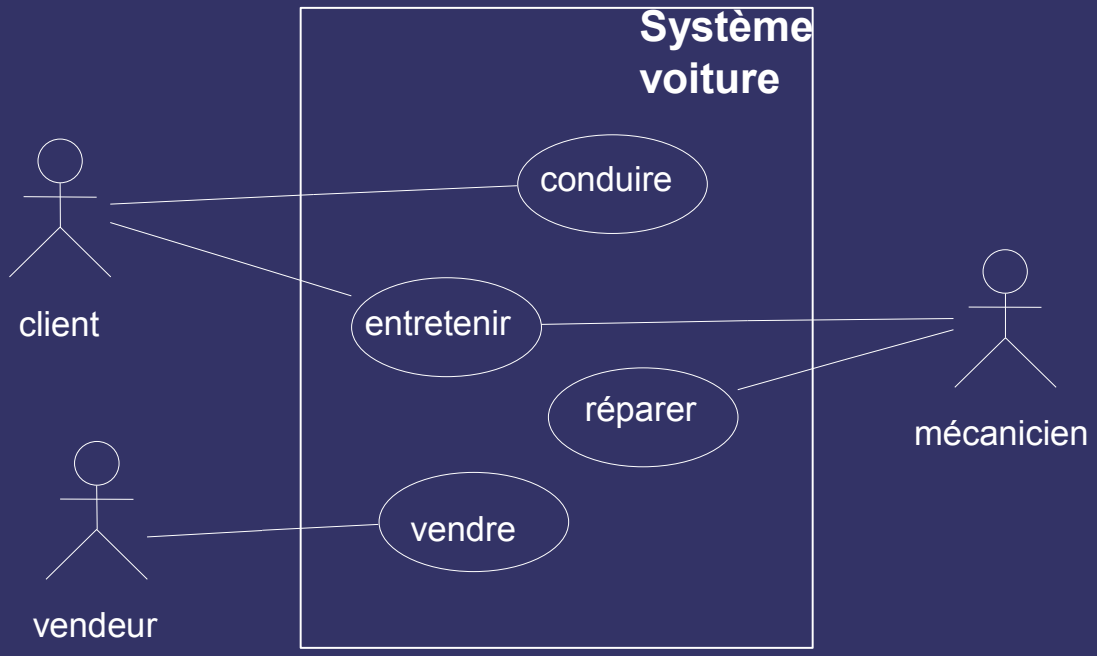
- Une classe avec stéréotype

- Un acteur doit être défini précisément, en quelques lignes.

Client : personne qui se connecte au distributeur bancaire à l'aide de sa carte. Peut avoir ou non un compte dans la banque qui possède le distributeur.

Cas d'utilisation

- Description générique d'une transaction complète entre acteur et système

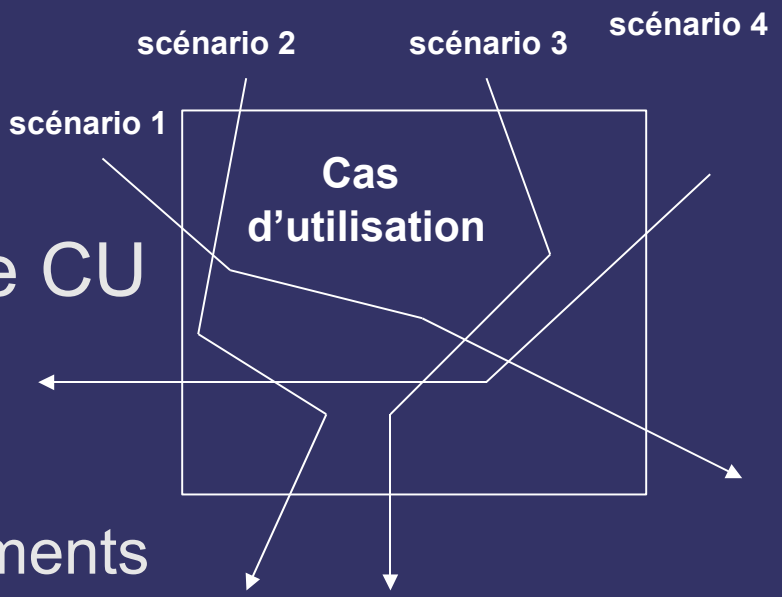


- Un CU doit être simple, clair, et précis
- Souvent un seul acteur humain par CU
- UML : un CU est en relation d'association avec un ou des acteurs

Cas d'utilisation et scénarios

- *Cas d'utilisation* = séquence de transactions entre le système et un ou plusieurs acteurs au cours d'un dialogue (interaction)
  - détermination en terme d'informations échangées et d'étapes dans la manière d'utiliser le système
  - un CU définit une famille de scénarios d'exécution (instances de CU) incluant les cas d'erreurs
  - Ex : CU = identification
- *Scénario* = séquence particulière de messages dans le CU pendant une interaction particulière
  - message : communication entre objets, porteuse d'information, et dont l'effet attendu est une action
  - tous les scénarios d'un CU sont issus du même acteur et ont le même objectif
  - les scénarios servent de base pour les jeux d'essais
  - Ex : scénario = Hector se connecte, tape son login « xxx » suivi de <entrée>, tape son mot de passe « toto » suivi de <entrée>, le système affiche un fenêtre marquant « Bienvenue Hector, cela fait 10 jours que vous ne vous êtes pas connecté, etc.

Cas d'utilisation et scénarios



- Scénario = chemin dans le CU
- Description du CU
  - ensemble des scénarios
  - document avec flot d'événements
    - du point de vue de l'acteur
    - détaille ce que le système doit fournir à l'utilisateur quand le CU est exécuté
      - flot normal des événements (80 %)
      - flots d'événements alternatifs
      - flots d'exceptions (quand le CU ne ne termine pas correctement)

Documentation des CU / scénarios associés

- Identification : titre, but du CU, résumé, acteurs, date
- Description des enchaînement
  - Préconditions
  - Enchaînements
    - début (« qd X se produit ») / fin (« qd Y se produit »)
    - interaction CU/acteur (dedans/dehors du système)
    - échanges d'information (« l'utilisateur se connecte au système, donne son nom, … »), chronologie et origine des infos.
    - répétitions de comportement

```
pendant que,
-- ce qu'il se passe
fin pendant
```

```
boucle
-- ce qu'il se passe
fin boucle
```

    - situations optionnelles (« l'acteur choisit un des éléments : a/ choix X ; b/ choix Y ; c/ choix Z. Puis l'acteur continue…»)
  - Exceptions
  - Postconditions

Documentation des CU

- Éventuellement en plus
  - besoins d'IHM / maquettes
  - contraintes non fonctionnelles
  - diagrammes de séquences au niveau besoins
    - acteurs + système seulement, boîte noire (voir plus loin)
  - diagrammes dynamiques simples
    - activité/état (voir plus loin)
- Règle générale : écrire le minimum pour que **tout** soit décrit sans ambiguité et de façon claire
  - n scénarios tous différents
  - m scénarios avec des alternatives
  - etc.

Exemple scénarios pour CU

Identification

Titre : opération sur compte

But : les opérations que le client peut réaliser sur son compte sur un distributeur bancaire.

Résumé : retrait, dépôt de chèque

Acteurs : Client

Auteur/Date : Toto / 20-09-2000

…

Exemple scénarios pour CU

…

Description des enchaînements :

Préconditions : l'utilisateur est identifié, il possède un compte

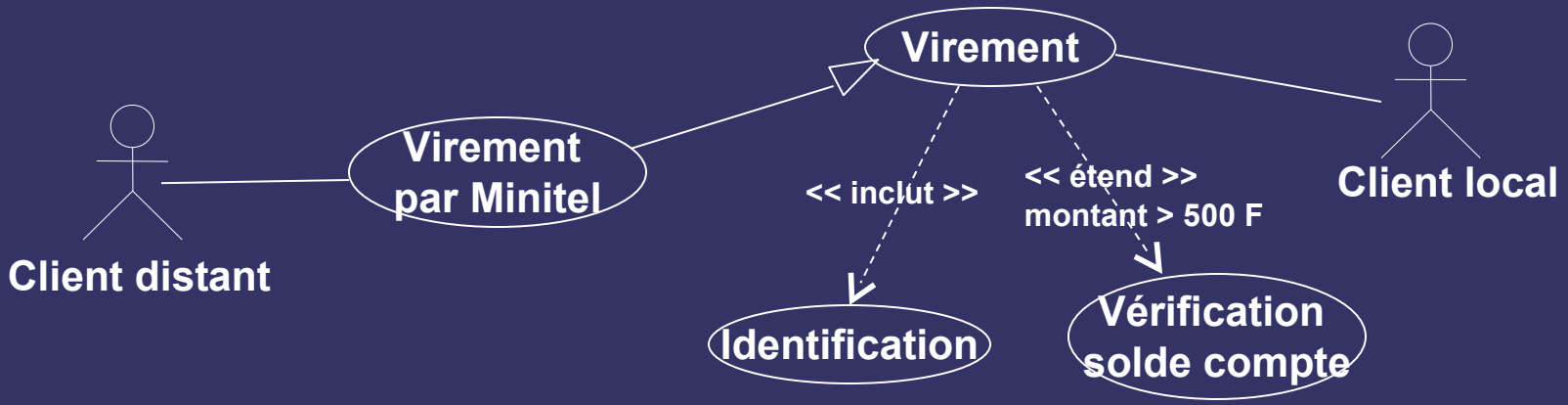
Enchaînements : le cas d'utilisation commence quand l'utilisateur se connecte. Le système lui propose : a/ de retirer du liquide ou b/ de déposer un chèque

Si le client choisit a/, le système lui propose divers montants. L'utilisateur choisit une somme. Le système vérifie que le compte est suffisamment approvisionné (sinon : Exception 1), et délivre l'argent et un reçu. … le cas d'utilisation se termine quand l'utilisateur retire sa carte bancaire.

Exceptions : Exception 1: le système informe l'utilisateur de l'échec de l'opération.

Postconditions : si il y a eu retrait, le solde du compte a baissé.

Diagrammes de CU



- 3 stéréotypes de liens entre CU
  - << include >> : la réalisation d'un CU nécessite la réalisation d'un autre, sans condition, à un point d'extension **(le plus important)**
  - << extend >> : entre deux instance de CU : le comportement de CU1 peut être complété par le comportement de CU2 : option avec condition et point d'extension *(certains auteurs conseillent de ne pas utiliser)*
  - << generalize >> : héritage, sorte_de. Relations entre CU et pas entre instances de CU
- Pas d'interactions entre acteurs
- Texte (scénarios) plus important que diagramme de CU
- **Attention : pas de décomposition fonctionnelle !**

CU : questions à se poser

- Pour fixer
  - quelles sont les intentions de l'acteur
  - quelles actions est-il susceptible de faire ?
- Plus précis
  - quelles sont les tâches de l'acteur ? quelles info l'acteur doit-il créer, sauvegarder, modifier, détruire, ou simplement lire ?
  - l'acteur doit-il informer le systèmes de changements externes ?
  - le système doit-il informer l'acteur de changements internes ?
- Attention
  - c'est bien le domaine du quoi, et du quoi faire, pas du comment (sinon on rentre en phase de conception)
  - on doit rester au niveau de l'interaction acteur/système
  - le CU doit rester simple
  - le niveau de détail nécessaire dépend du risque associé au CU

CU : questions supplémentaires (Müller)

- Quelques questions de vérification
  - Existe-t-il un brève description qui donne une vraie image du cas d'utilisation ?
  - Les conditions de démarrage et d'arrêt du CU sont-elles biens cernées ?
  - Les utilisateurs sont-ils satisfaits de la séquence d'interactions entre acteur et CU ?
  - Existe-t-il des étapes communes à d'autre CU ?
  - Les mêmes termes employés dans des CU différents ont-ils même signification ?
  - Est-ce que toutes les variantes sont prise en compte par le CU ?
- Documentation informelle
  - Est-ce que tous les besoins identifiés de manière informelle sont pris en compte ?
  - Est-ce qu'un même besoin a été alloué à plusieurs CU ?
  - Existe-t-il des CU qui ne sont pas référencés par des besoins informels ?

V
Diagrammes d'interaction

Cas d'utilisation et interactions

- Le diagramme des CU présente une vue du système *de l'extérieur*
- Une interaction décrit comment les cas d'utilisation (classes, opérations) sont réalisés comme interactions dans une « société » d'objets (communication entre objets)
  → virage vers l'objet
- Vue dynamique du système : deux types de diagrammes d'interaction
  - diagrammes de communication (UML1 : collaboration)
  - diagrammes de séquences

Messages entre objets

- Matérialisation d'une communication avec transmission d'information entre objet émetteur (source) et récepteur (destination)
- Un message déclenche une opération, l'émission d'un signal, ou la création/destruction d'un objet
- Deux types principaux
  - appel de procédure ou flot de contrôle emboîté (retour implicite) → déplacer()
  - flot de contrôle asynchrone → démarrer()
  - Autres : à plat, dérobant (réception si attente), minuté (message actif pendant Δt)

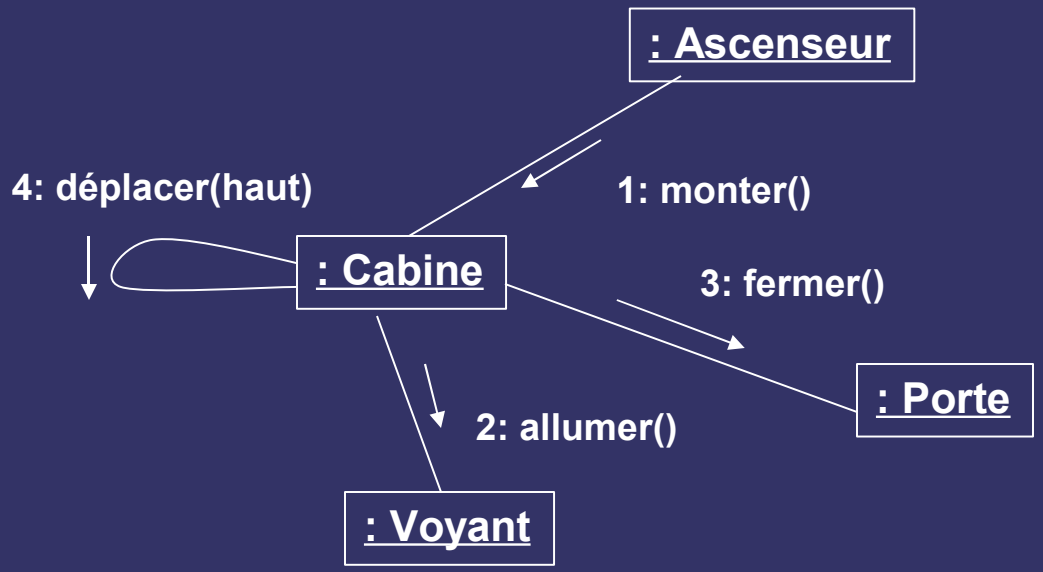
Interactions

- Communication entre instances des éléments d'une collaboration (contexte) = ensemble partiellement ordonné de messages
- Plusieurs interactions possibles pour une même collaboration
- Éléments d'une interaction
  - instances
  - liens (= supports messages)
  - messages déclenchant des opérations
  - rôles joués par les extrémités de liens

- Plusieurs vues possibles : espace et temps

Diagramme de communication

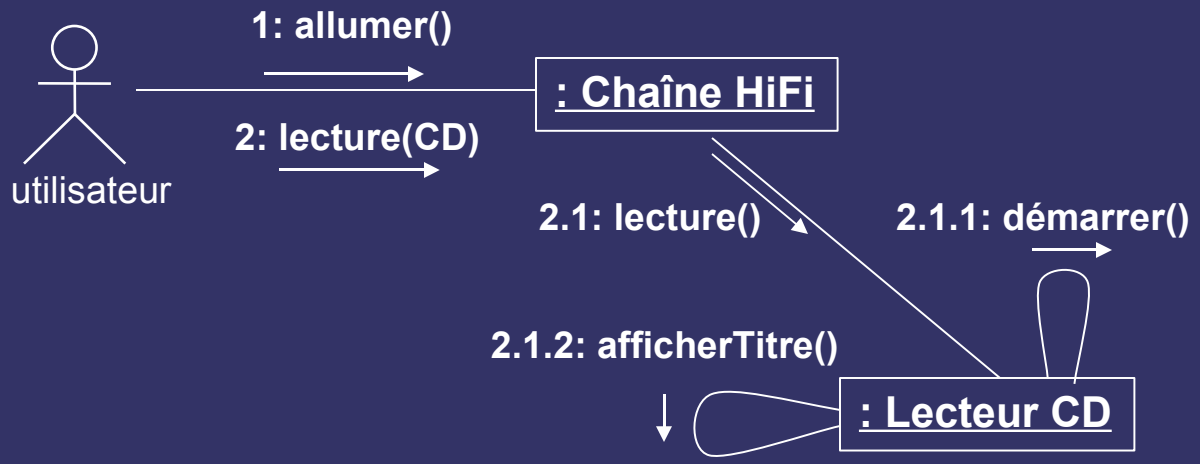
- Diagramme d'objets rendant compte de la dynamique
  - structure spatiale permet la collaboration d'objets
  - dimension temporelle : ordre des messages (numérotation)



Message ::= synchronisation séquence : résultat := nom (arguments)

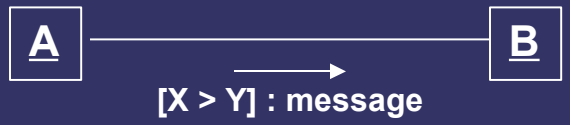
- Synchronisation
= prédecesseur / [condition]

- Séquence :
  - numérotation
  - récurrence



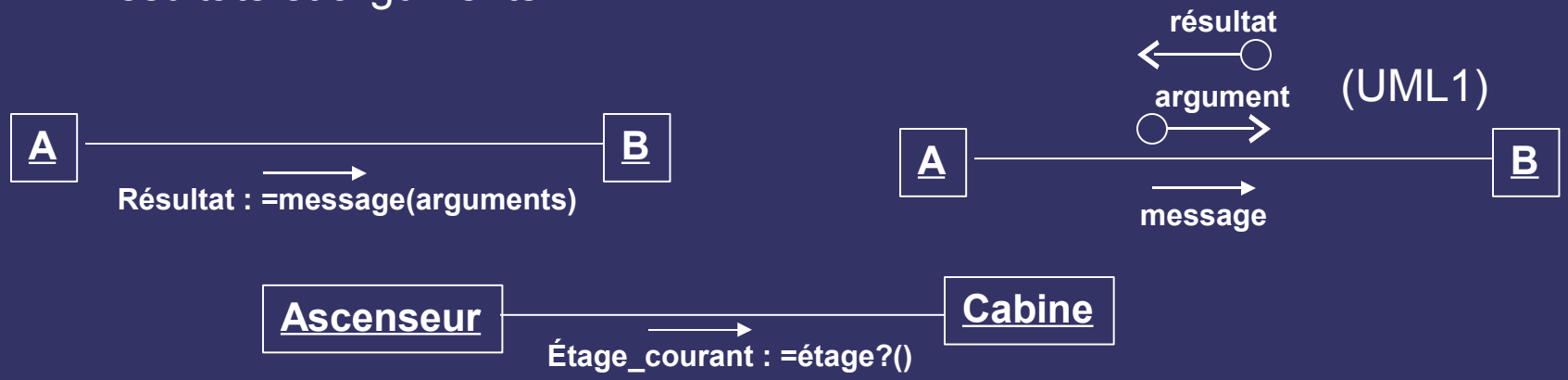
itération

condition



Message ::= synchronisation séquence : résultat := nom (arguments)

- Résultats et arguments



- Sélection

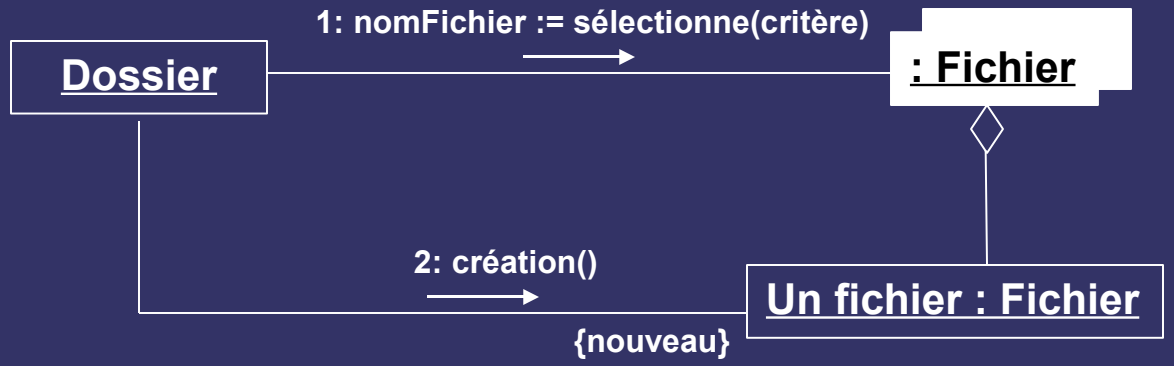
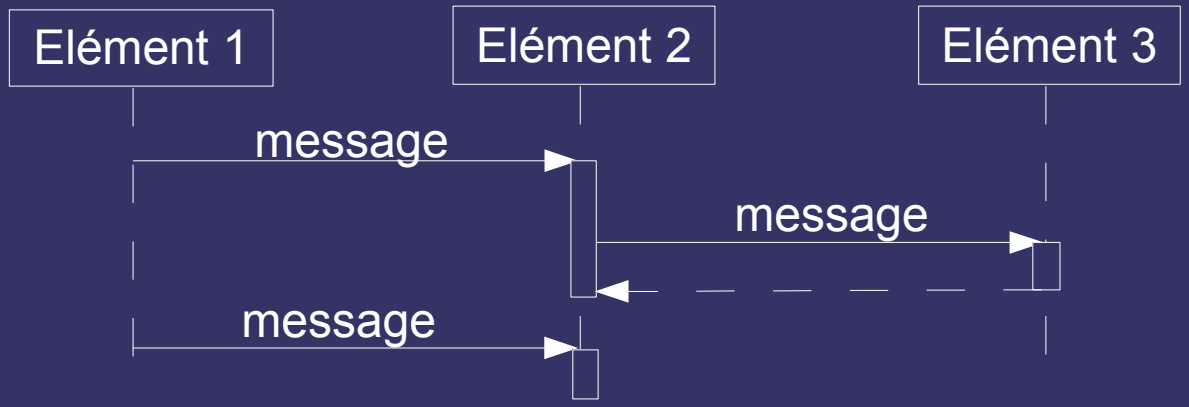


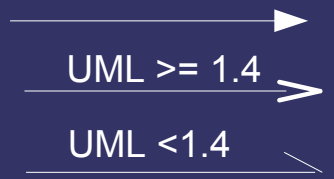
Diagramme de séquences

- Interactions entre objets dans une séquence *temporelle*
  - aspect chronologique ne rendant pas compte explicitement du contexte
  - permet de bien montrer qui fait quoi dans une interaction
- Description de scénarios typiques et des exceptions
- Participants de l'interaction
  - éléments (UML1 : objets). Dans les fait, souvent objets.
- Lignes de vie verticales
  - barres d'activation = activité de l'élément



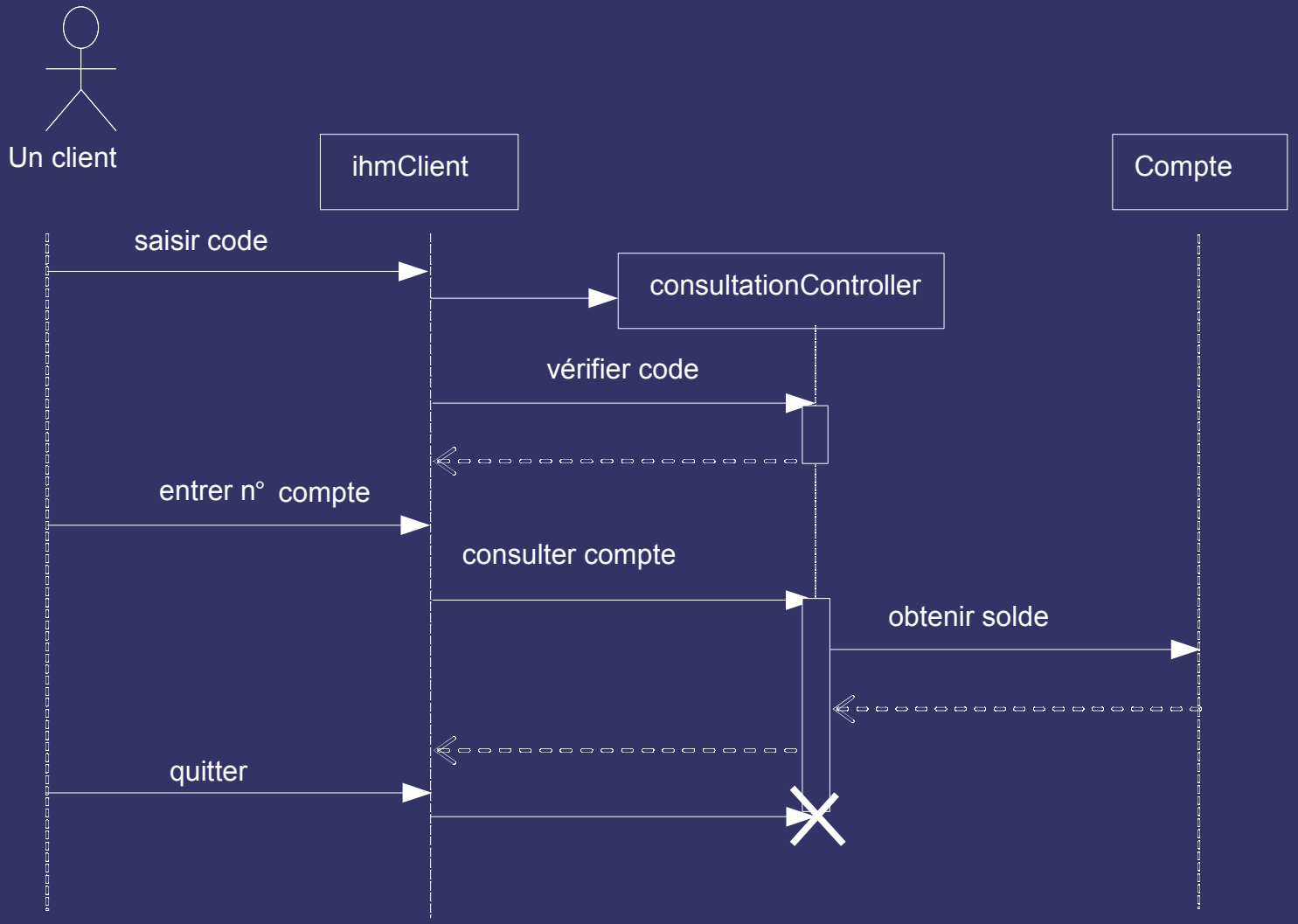
Messages dans les diagrammes de séquences

- Echange de messages
  - flèches d'appel standard (passer le flot de contrôle)
  - flèche d'appel asynchrone (ne pas attendre le retour)
  - Retour
  - Message de création
  - Message de destruction

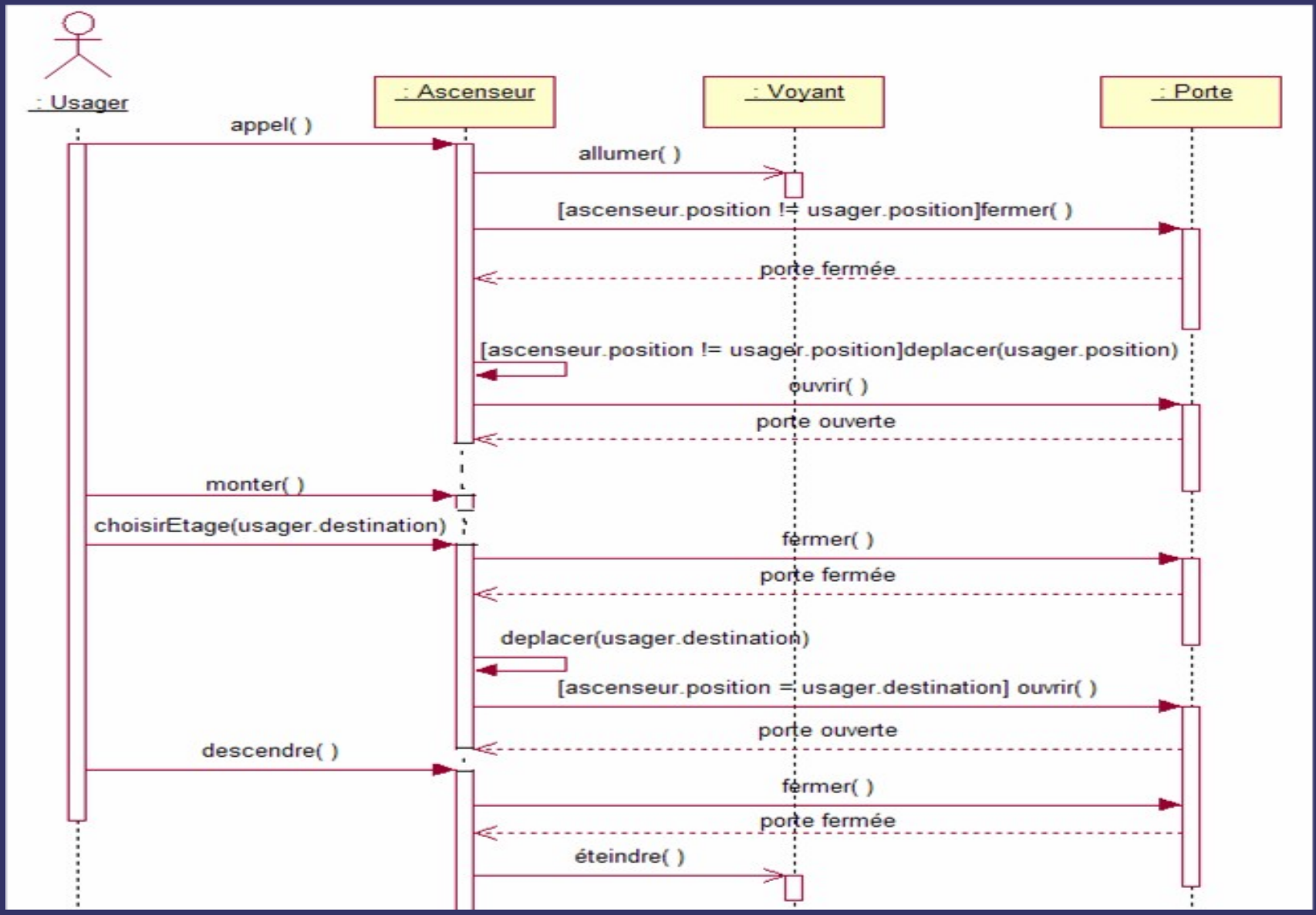


- Lancement de l'interaction provient de l'extérieur
  - 1er message = « message trouvé »

Exemple


Un client
ihmClient
Compte
saisir code
consultationController
vérifier code
entrer n° compte
consulter compte
obtenir solde
quitter

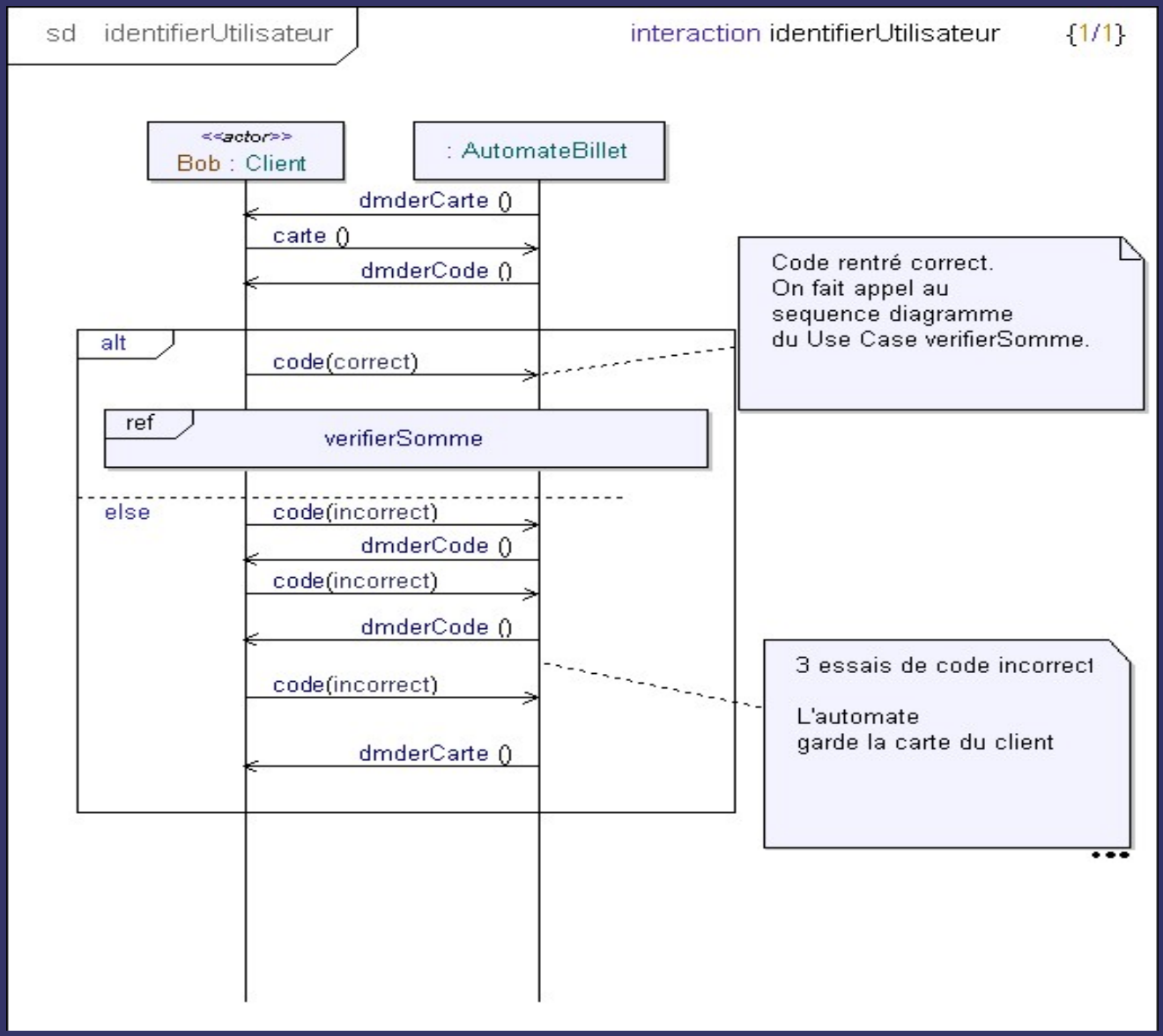
Exemple


: Usager
: Ascenseur
: Voyant
: Porte
appel( )
allumer( )
[ascenseur.position != usager.position]fermer( )
porte fermée
[ascenseur.position != usager.position]deplacer(usager.position)
ouvrir( )
porte ouverte
monter( )
choisirEtage(usager.destination)
fermer( )
porte fermée
deplacer(usager.destination)
[ascenseur.position = usager.destination] ouvrir( )
porte ouverte
descendre( )
fermer( )
porte fermée
éteindre( )

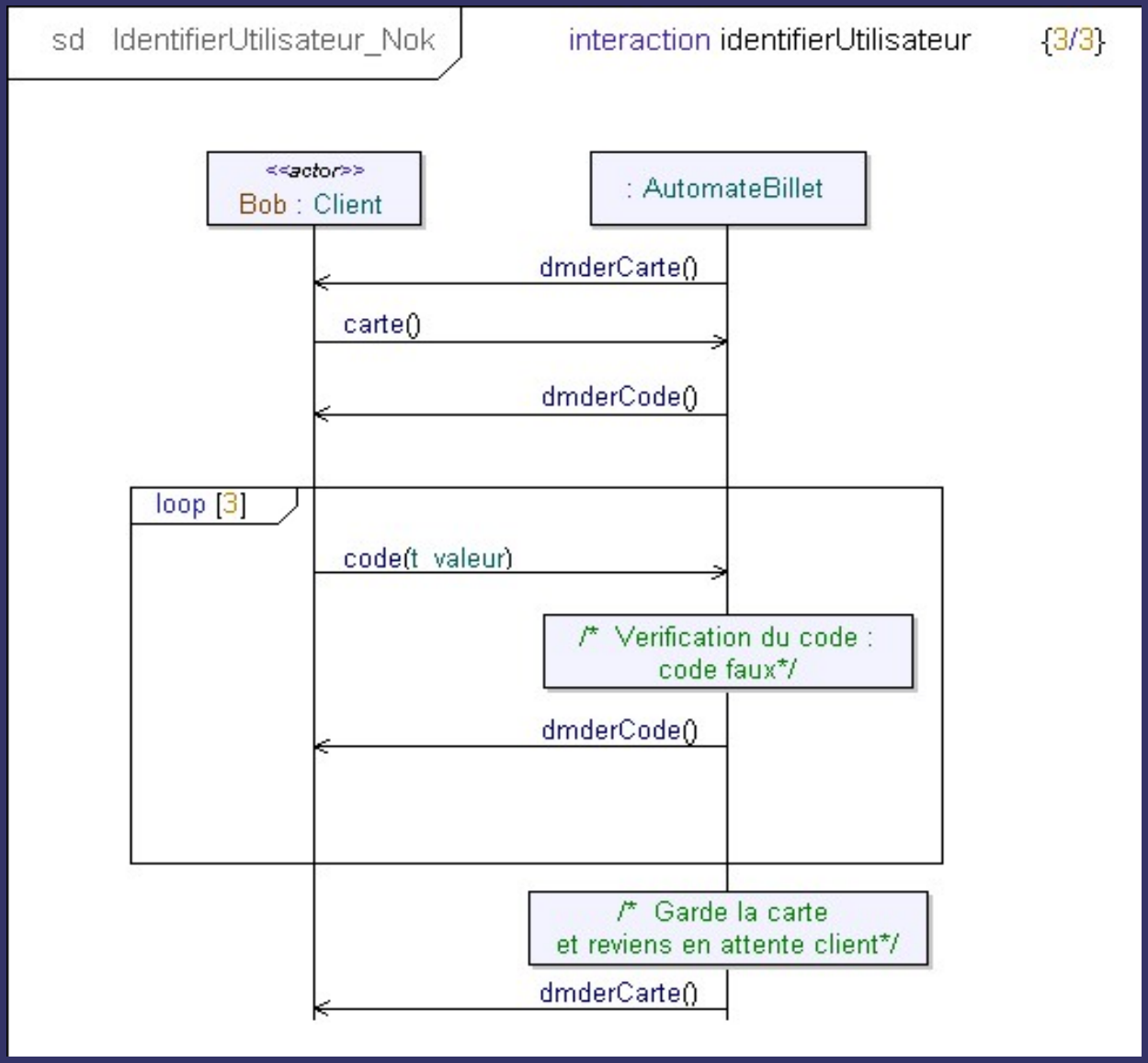
Opérateurs (UML2.0)

- Cadres nommés qui entourent la partie critique
- Pour exprimer
  - conditionnelles : Alt
  - itérations : Loop
  - interactions optionnelles : Opt
  - diagramme de séquence : sd (pour tout un DS)
  - parallèlisme : par
  - Etc.
- A suivre
  - Quelques exemples (devellopez.com)

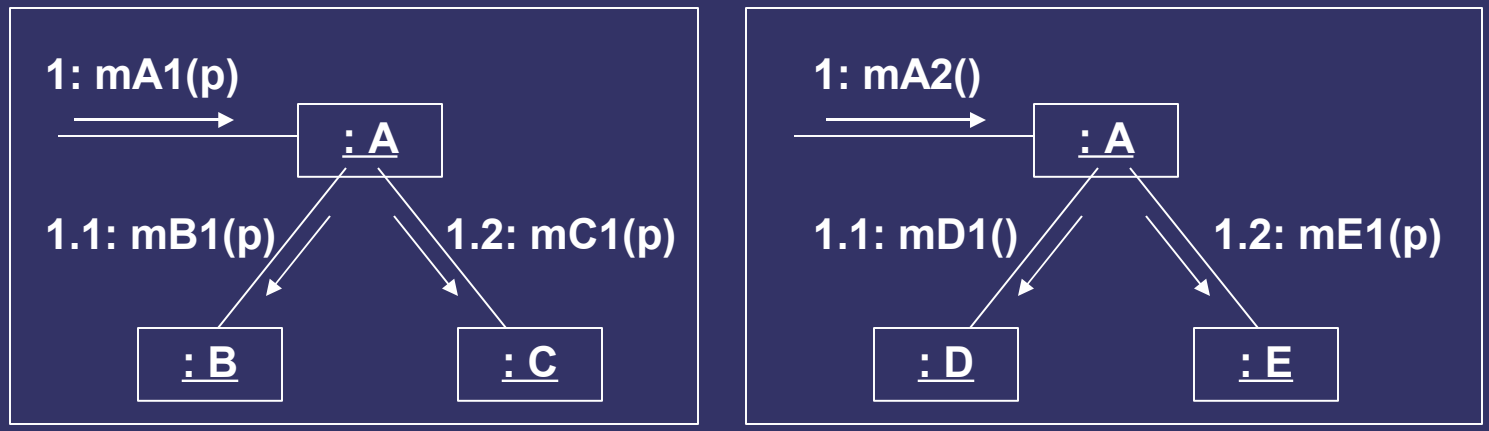
alt

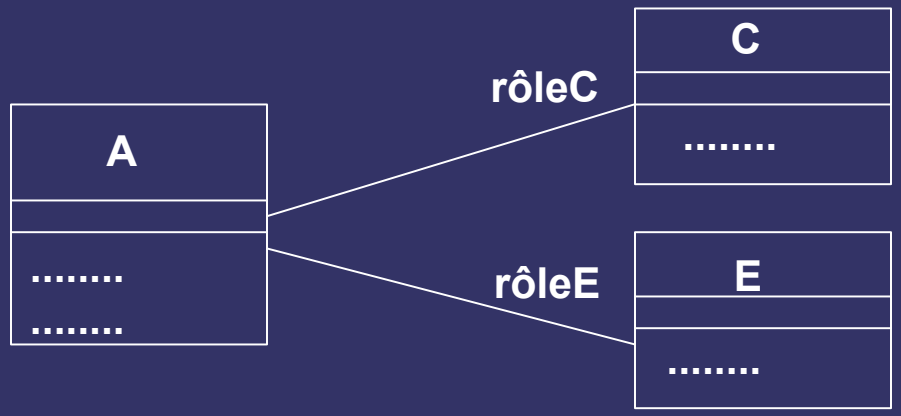

sd identifierUtilisateur
interaction identifierUtilisateur {1/1}
<<actor>>
Bob : Client
: AutomateBillet
dmderCarte ()
carte ()
dmderCode ()
Code rentré correct.
On fait appel au
sequence diagramme
du Use Case verifierSomme.
alt
code(correct)
ref
verifierSomme
else
code(incorrect)
dmderCode ()
code(incorrect)
dmderCode ()
code(incorrect)
dmderCarte ()
3 essais de code incorrect
L'automate
garde la carte du client

loop


sd IdentifierUtilisateur_Nok
interaction identifierUtilisateur {3/3}
<<actor>>
Bob : Client
: AutomateBillet
dmderCarte()
carte()
dmderCode()
loop [3]
code(t valeur)
/* Verification du code :
code faux*/
dmderCode()
/* Garde la carte
et reviens en attente client*/
dmderCarte()

Union de diagrammes de communication/séquence





→ Diagramme structurel + responsabilités (base des opérations)

VI
Diagrammes dynamiques

Diagrammes dynamiques

- Si objets *déjà* déterminés dans un diagramme statique
  - *décrire* le comportement concret de la vie d'un objet (lié aux scénarios) en termes d'états

    → ensemble de chemins uniques (d'un état à un autre)
  - *fixer* le comportement attendu d'un objet au long de sa vie (spécification)

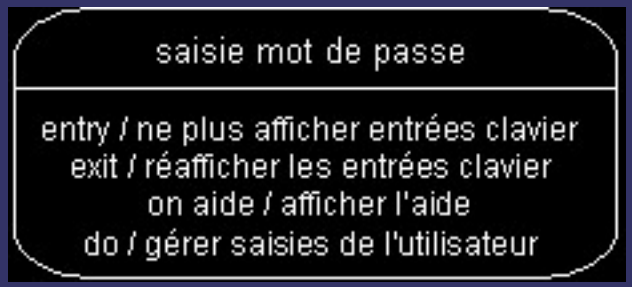
    → contraintes limitant les scénarios possibles (2^n si *n* attributs)

    → un graphe résumant tous les scénarios (passer d'un état à d'autre*s*).

    Une exécution = un scénario

Diagramme d'états

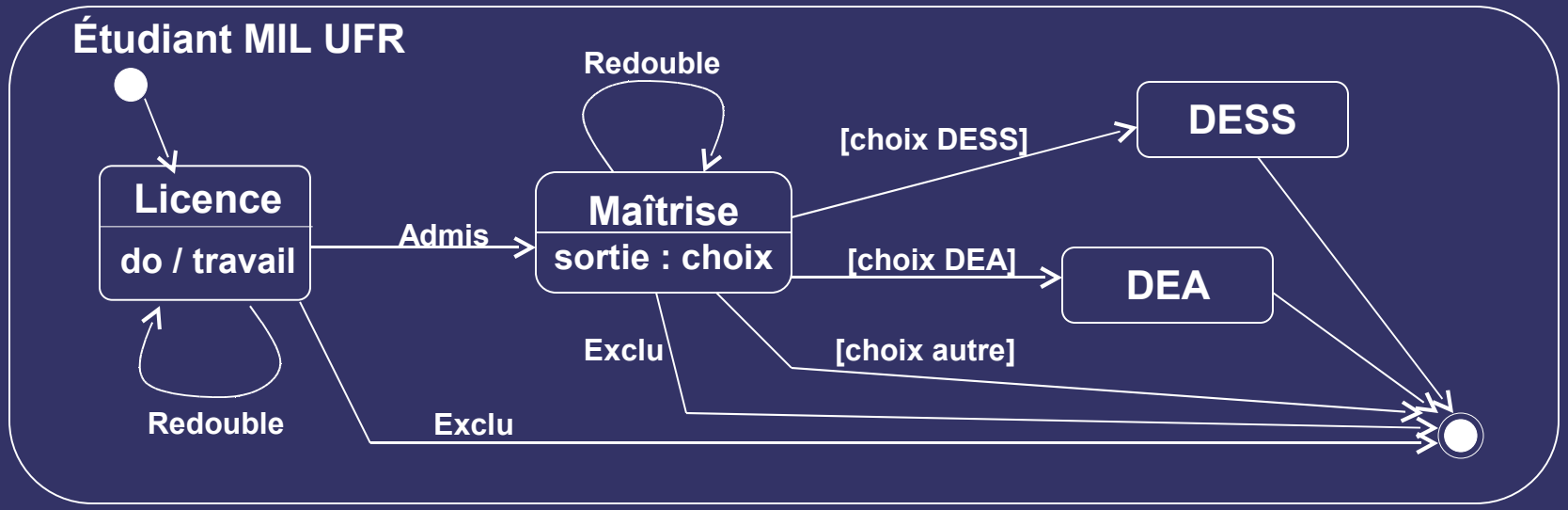
- Automates à états fini
- Abstraction des comportements possibles pour une classe
- État
  - valeurs des attributs, moment dans la vie d'un objet (durée/stabilité)
- Transition entre états
  - sur événement + condition respectée,
  - action à exécuter
- Dans un état
  - activité : continue (sonnerie), tâche de fond (pagination), attente, suite d'actions…

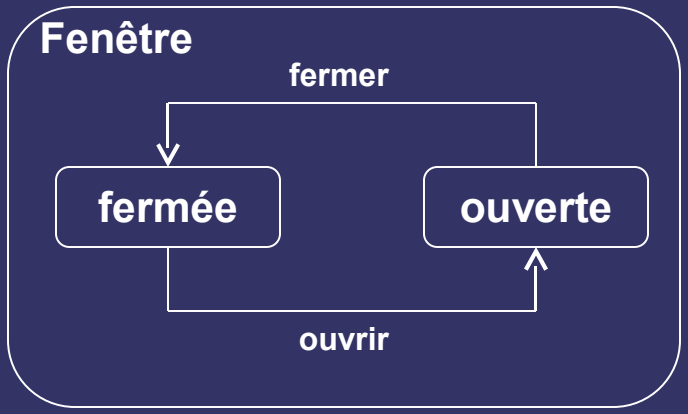


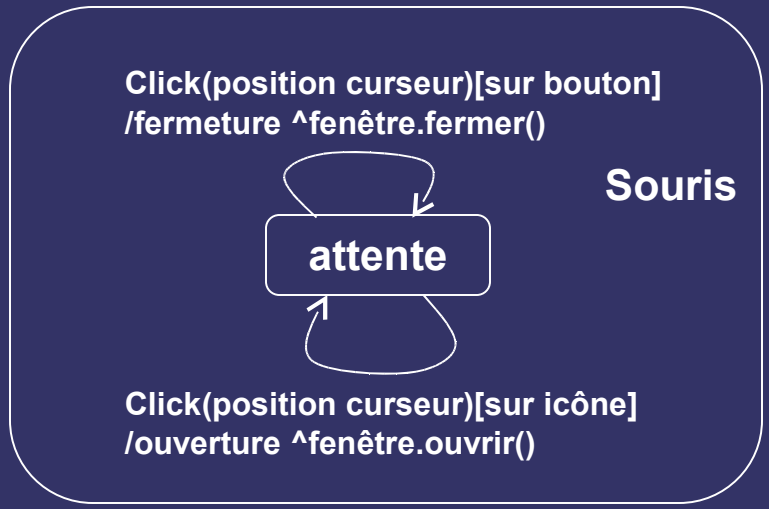
Syntaxe générale

- Transition
  - tout est facultatif
- Etat
  - activité temporaire (ordinaire)
    - notion d'autotransition sur événement extérieur
    - deux activités spéciales : sur entrée et sortie
  - activité continue
    - do / activité
    - peut être interrompue

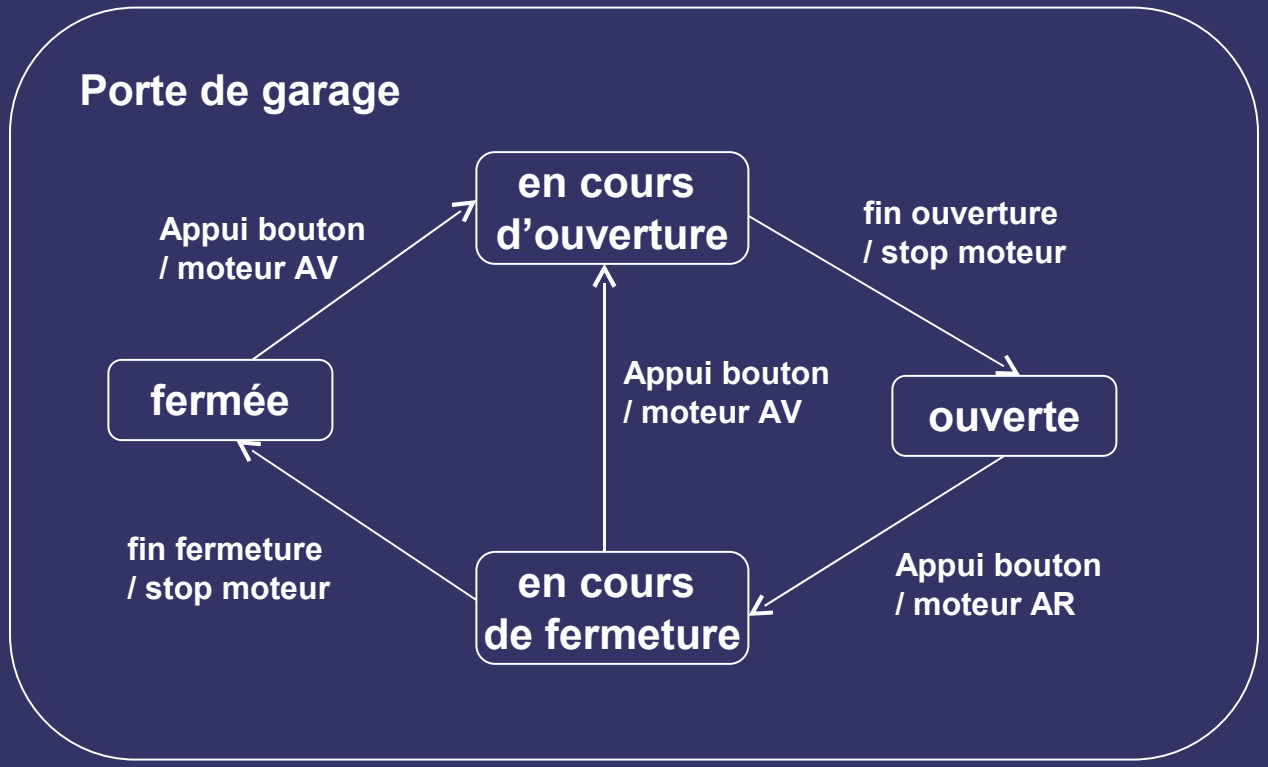
Exemples de diagrammes d'états





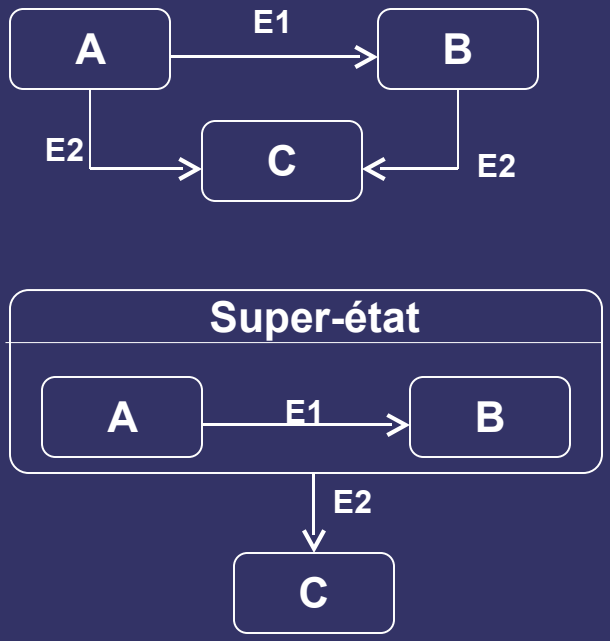


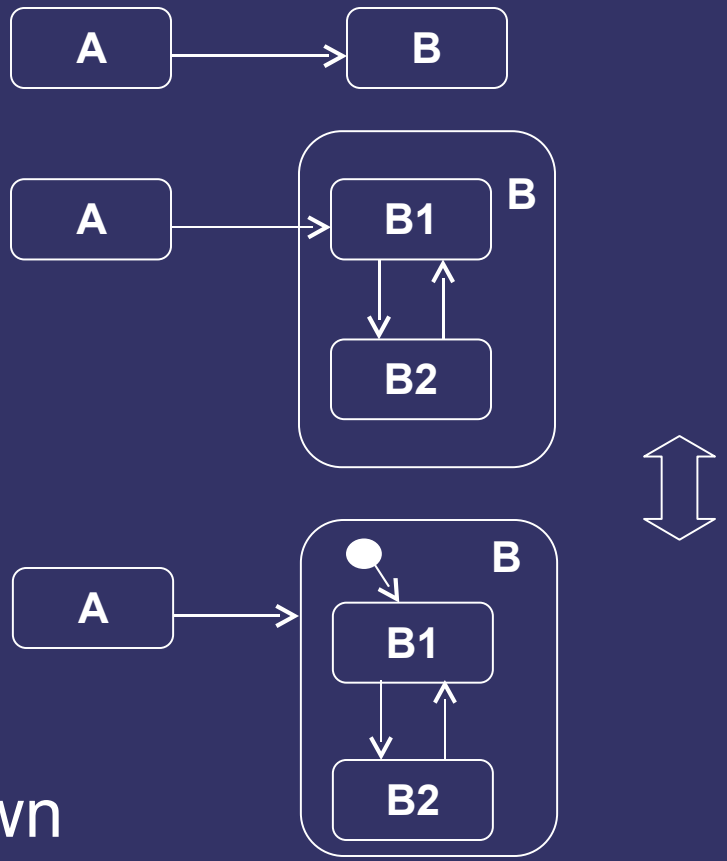
Exemple de diagrammes d'états



États composites

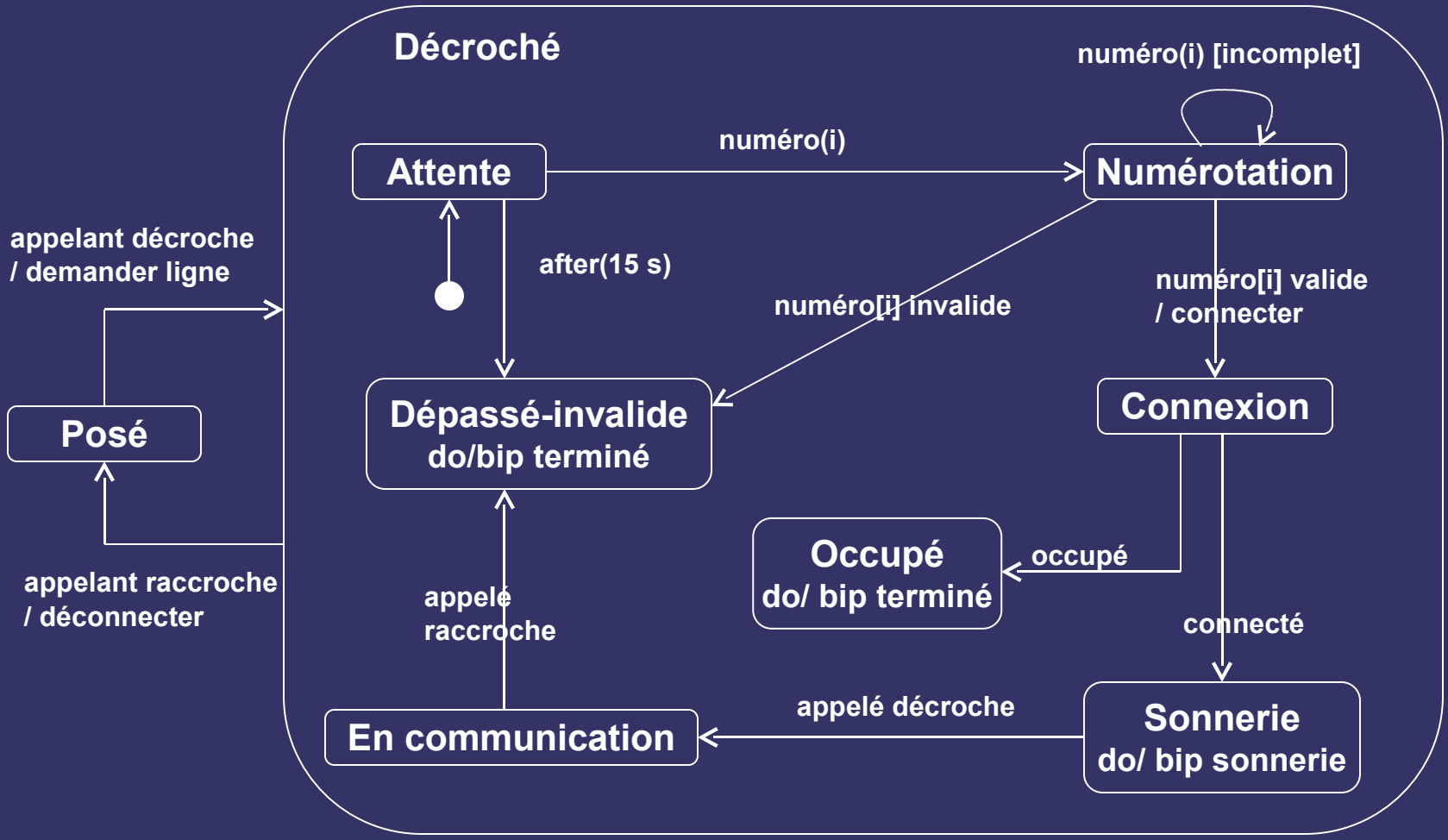
Les sous-états héritent des transitions externes



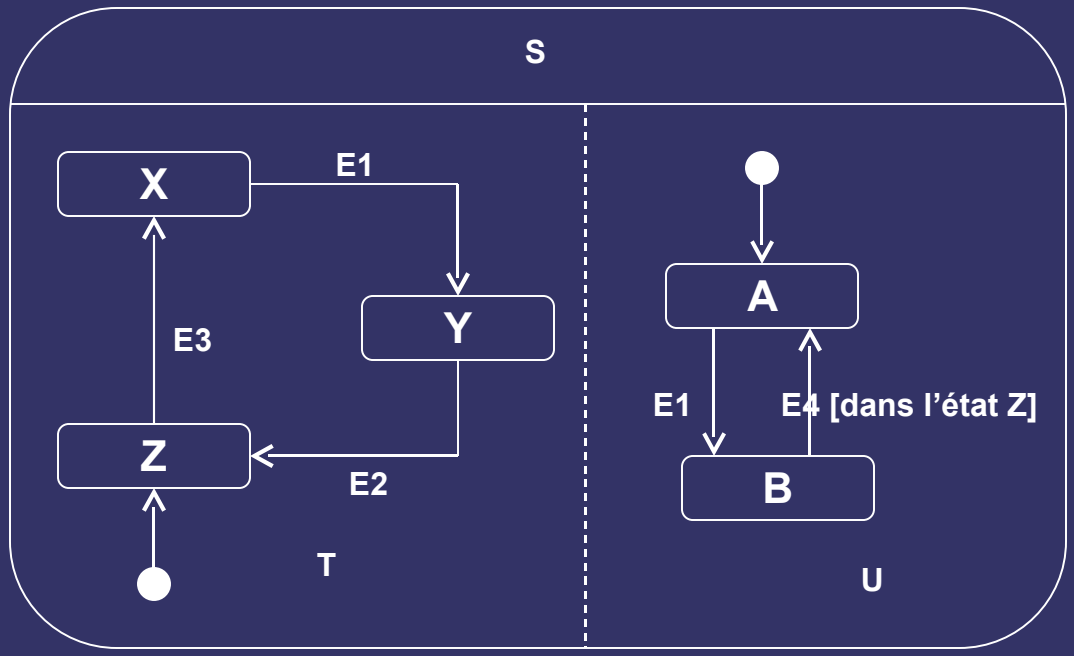


- Conception : topdown

Diagramme états/transition



États concurrents



- Exercice : trouver le diagramme d'état « à plat » équivalent

Diagramme d'activité

- Graphe d'enchâinement d'activités
  - logique procédurale
  - workflows
- A utiliser
  - en cas d'opération complexe, parallèle
  - pour décrire des processus métier / workflows

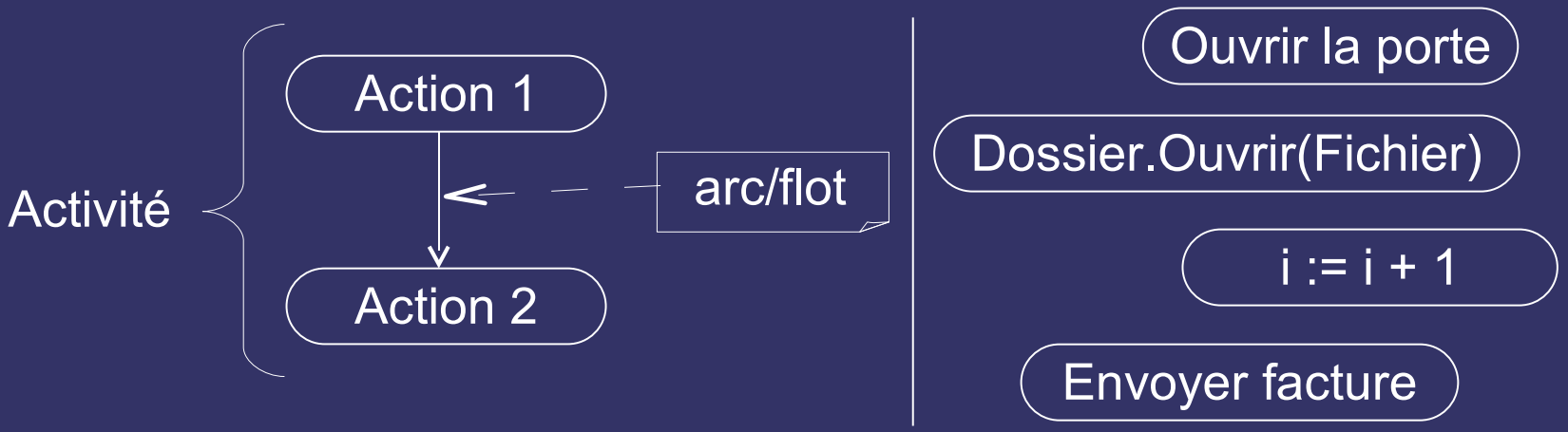
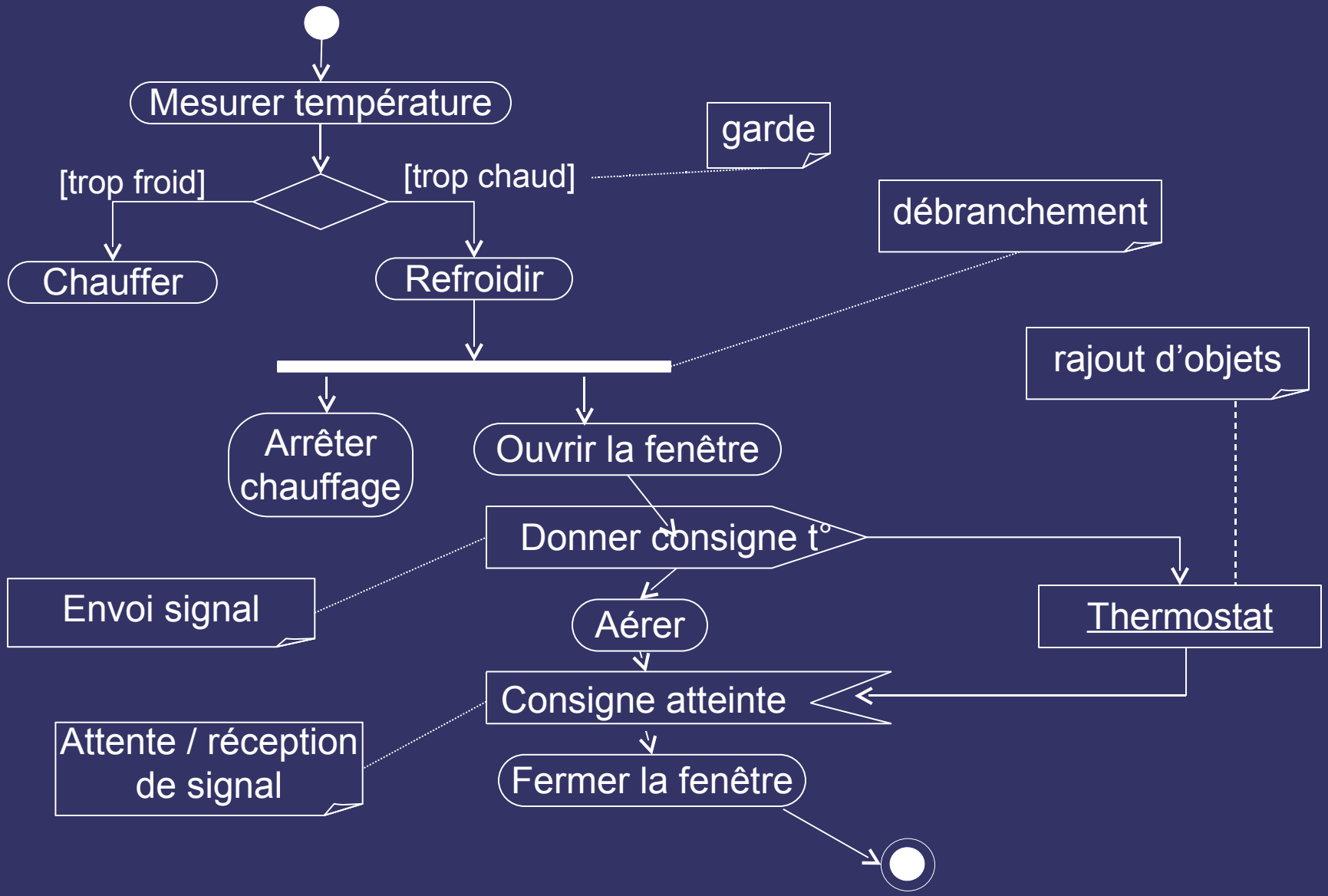
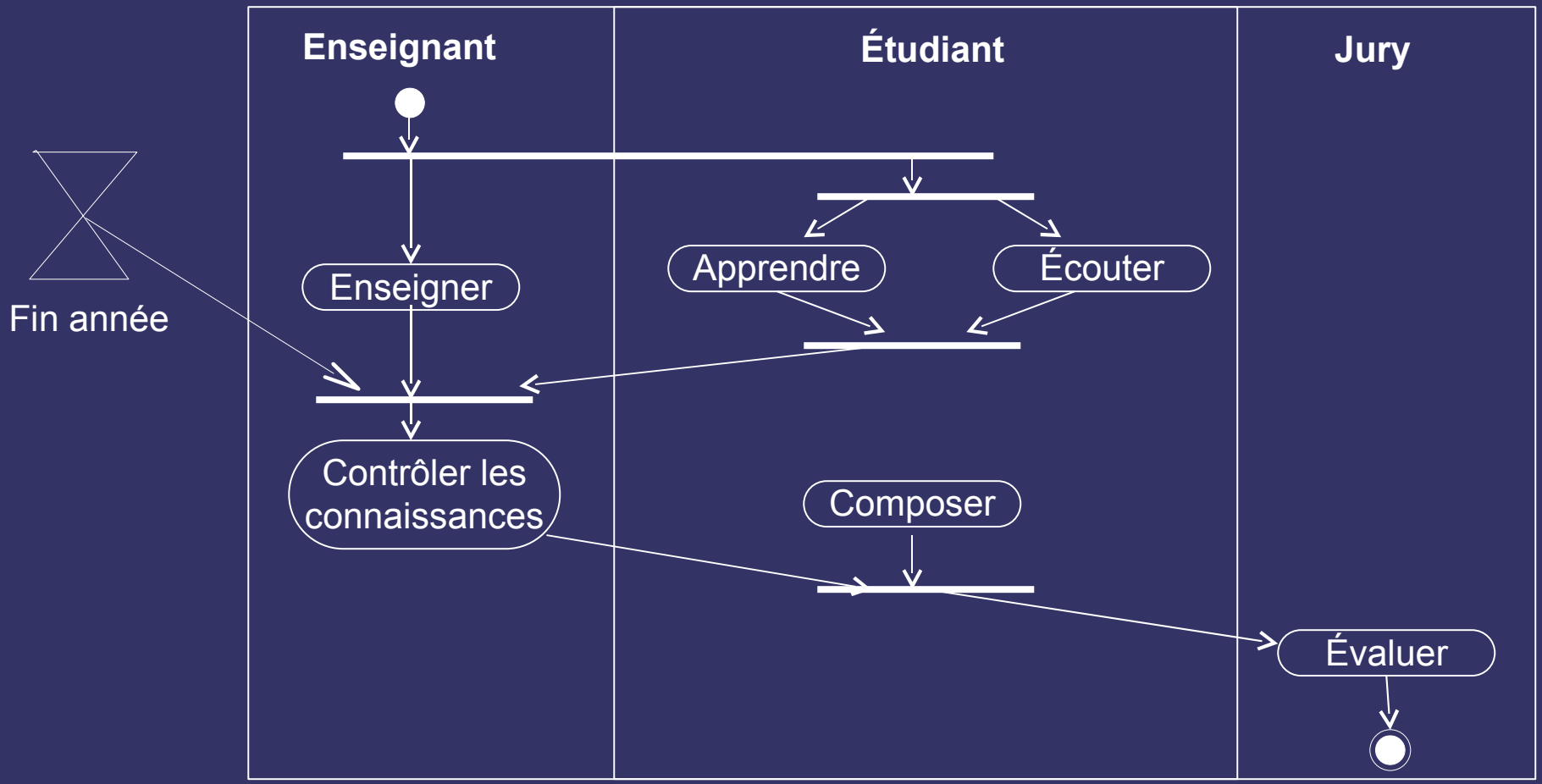


Diagramme d'activité : exemple



Travées

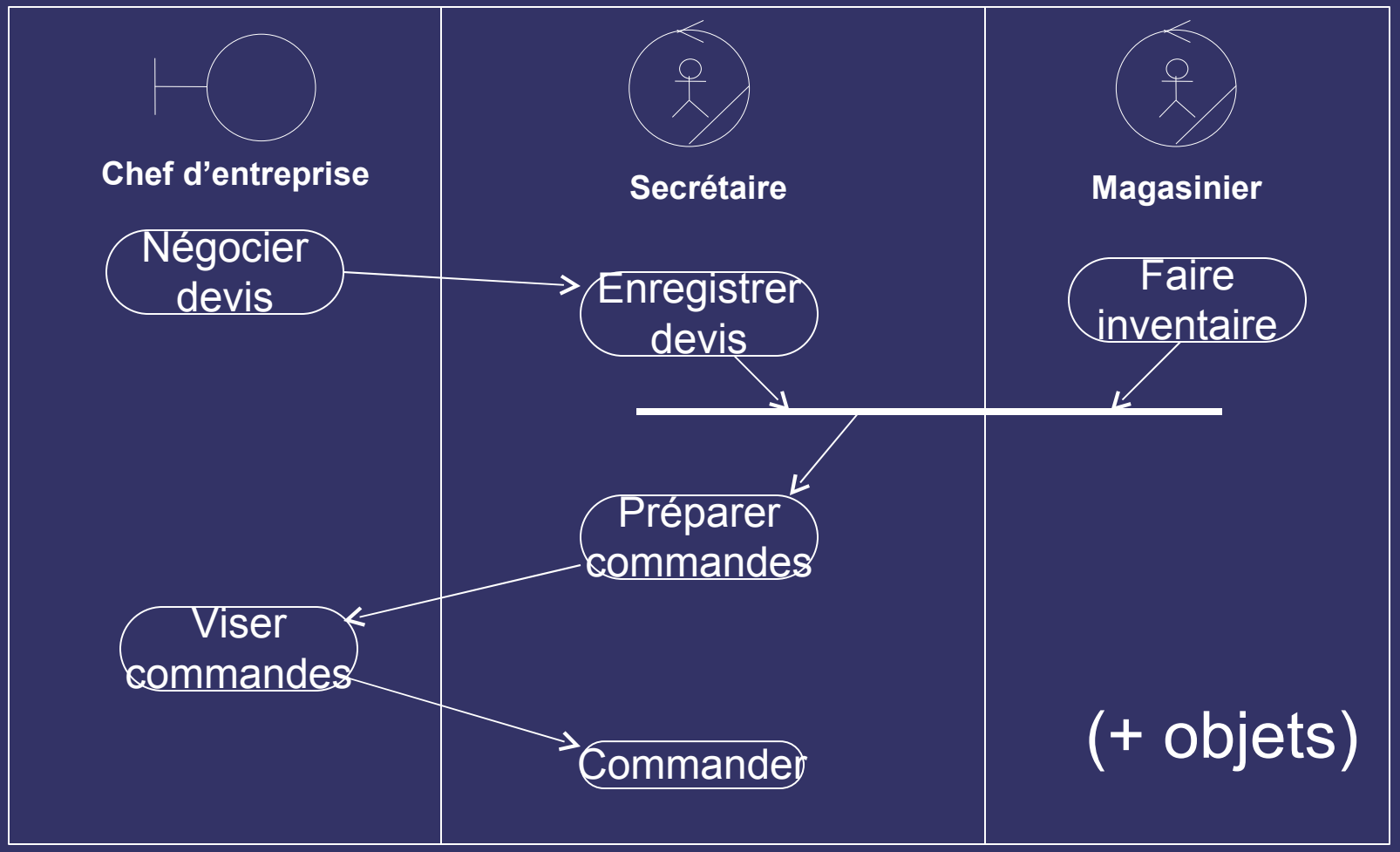
- Montrer les responsabilités au sein du mécanisme ou d'une organisation



« Business modeling »

- Un peu plus loin que les activités d'un objet : l'entreprise
  - questions QUI et OU
- Objets responsables
  - Case worker (interaction avec l'ext. de l'entreprise)
  - Internal worker
  - Entity : objet passif
- Couloirs d'activités

« Business modeling » : exemple



Diagrammes d'activité UML2

- Possibilité de sous-activités (cadre)
- Possibilité d'utiliser des jetons
  - cf. réseaux de Pétri
- Connecteurs
- Régions d'expansion
  - Pour représenter des actions qui se passent pour plusieurs éléments de même type (itératif ou concurrent)

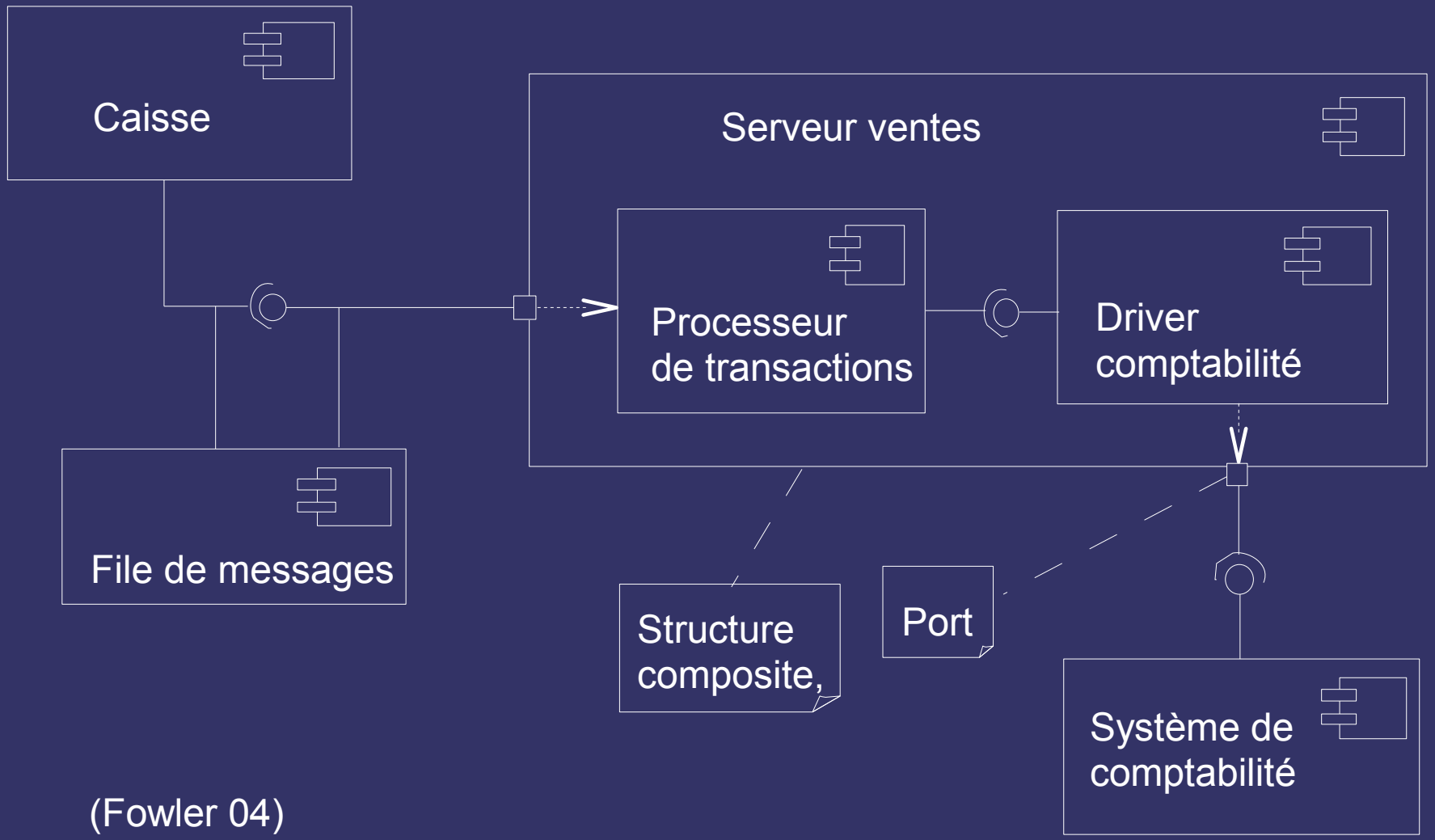
VII
Diagrammes de composants et de déploiement

Diagramme de composants

- Organisation et dépendances entre composants logiciels.
  - description des composants et de leurs relations dans le système en construction.
  - un composant doit être compris comme un élément du système qu'on peut acheter, mettre avec d'autres composants, etc.
  - division en composants = décision technique *et* commerciale (Fowler)
  - (avant UML2.0 : composant = n'importe quel élément, y compris fichiers. Maintenant, utiliser les *artefacts*).

Réalisation

Diagramme de composants



(Fowler 04)

Diagramme de déploiement

- Disposition physique des différents matériels qui entrent dans la composition d'un système, ainsi que disposition des programmes exécutables sur ces matériels.
  - visualiser la distribution des composants dans l'entreprise
  - Unités = nœuds
    - équipements = matériel
    - environnement d'exécution = logiciel
  - Un noeud contient des artefact : classes, ...
- Relations entre éléments : supports de communication

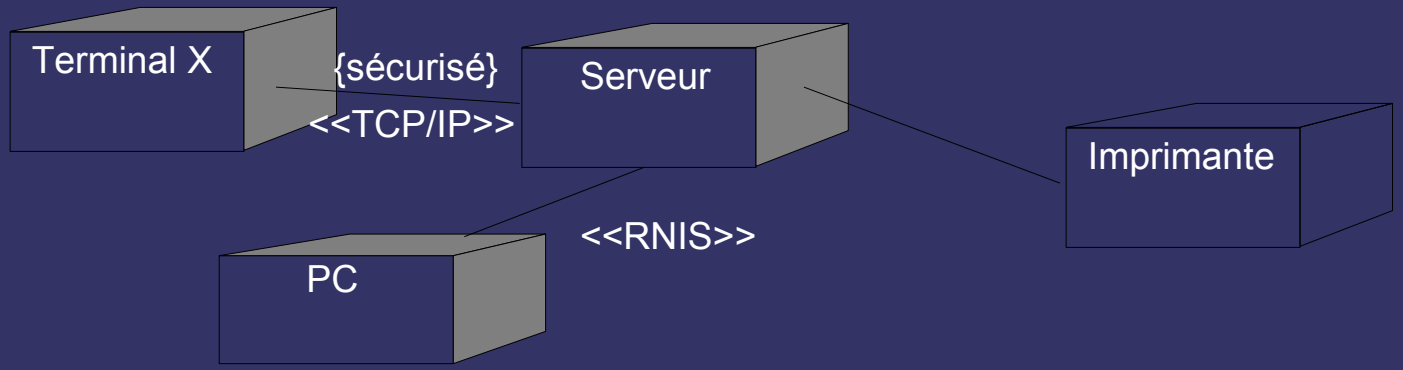
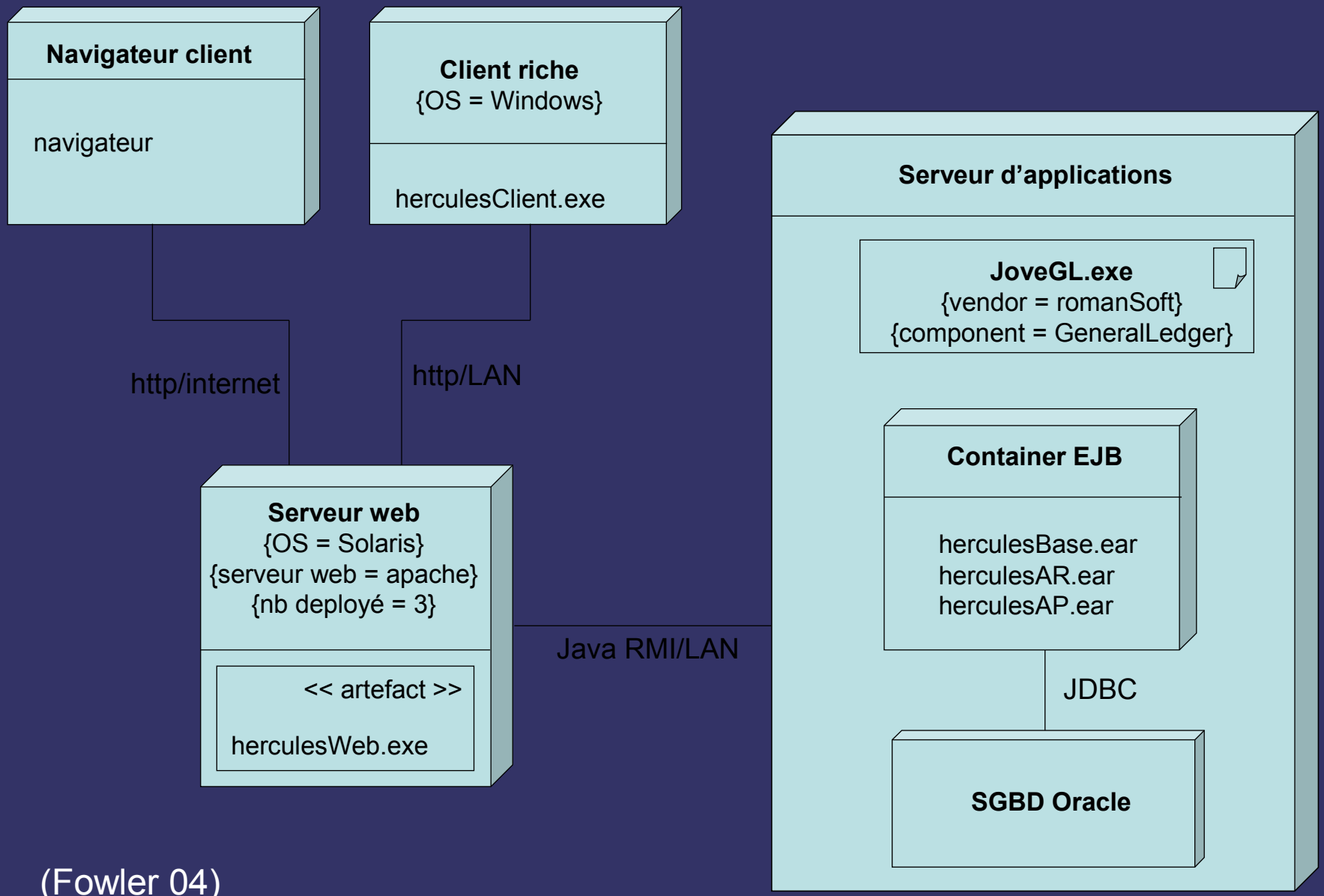


Diagramme de déploiement



(Fowler 04)

VIII
Divers

Autres diagrammes

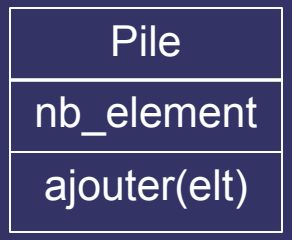
- Collaborations (non officiel)
  - Présenter les éléments impliqués dans une colloboration
- Vue d'ensemble des interactions
  - Mixte diagramme activité / diagrammes de séquences
- Timing
  - Interactions avec focus sur les contraintes temporelles
- Structures composites
  - Permettent de décomposer une classe

Object Constraint Language

- Standardisé par l'OMG
- Permet d'exprimer des contraintes de façon formelle
- Expression
  - d'invariants au sein d'un classe ou d'un type : bon fonctionnement des instances
  - contraintes au sein d'une opération : bon fonctionnement de l'opération
  - pré- et post- conditions d'opérations : avant et après l'exécution
  - gardes : sur la modification de l'état d'un objet
  - expressions de navigation : chemins

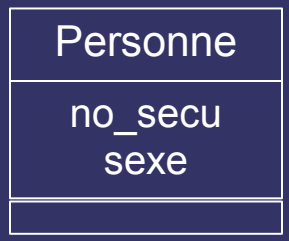
OCL : exemples

context nom_élément [**inv**|**pre**|**post**] : expression de la contrainte

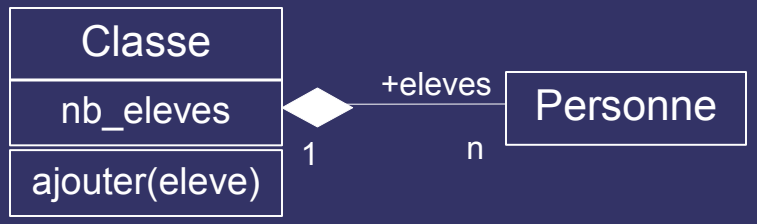


context Pile **inv** :
    self.nb_elements >= 0 -- nb_element = attribut de Pile

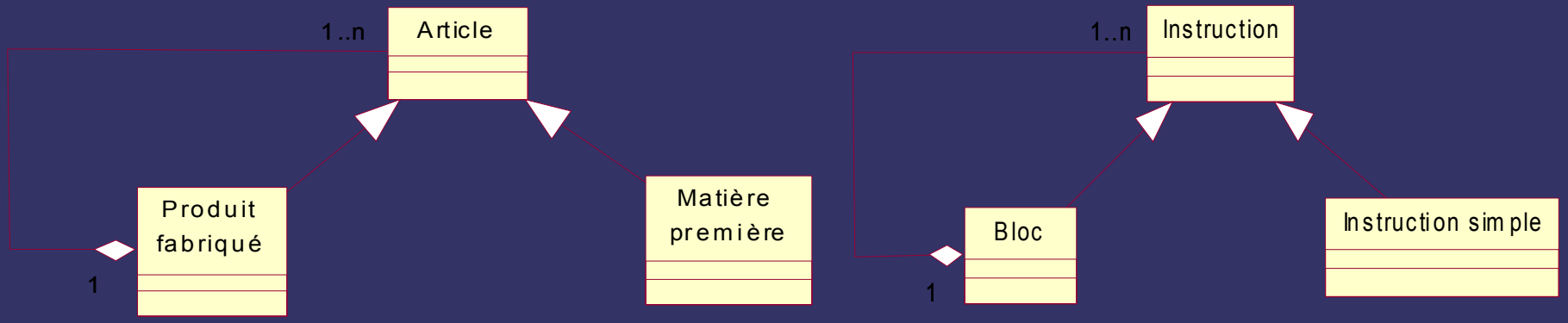
context Personne **inv** -- intégrité de l'objet personne
                / attributs no_secu et sexe
    if sexe = "F" **then** no_secu.commence_par() = 2
    else no_secu.commence_par() = 1
    endif



context Classe::ajouter(un_eleve : eleve)
    pre classe_non_surchargée : nb_eleves <= 25
    post : eleves→exists(un_eleve)



Design pattern / framework



- design pattern → solution de conception réutilisable (cf. couture)
  - indépendance / langages de programmation
  - solution générique éprouvée et documentée
  - nom, contexte, description, pièges, etc.
  - http://www.hillside.net/patterns
- framework → plus haut niveau
  - infrastructure réutilisable
  - beaucoup de classes abstraites et une architecture à étendre
  - difficile à maîtriser

Traduction de UML en code

- Différents types
  - objet (bien sûr)
  - impératif (utile ?)
  - bases de données (de plus en plus utilisé)
- Automatique ?
  - information dans les modèles insuffisante
  - paramétrage des outils : traduction des associations, méthodes engendrées automatiquement
  - degré de traduction dépend du langage cible (ex. Java et héritage multiple)
  - il existe des outils de vérification
- Modèle structurel largement traduit
  - squelette des classes
  - héritage, associations,
  - modules
  - …

Traduction en objet (C++)


```
#ifndef COURS
#define COURS
class Eleve
class Cours {
      public:
            void associer(un_module : string) ;
            void set_titre(const string value) ;
            const string get_titre() const ;
            void set_module(const string value) ;
            const string get_module() const ;
      protected:
            Eleve* Eleves ;
      private:
            string titre ;
            string module ;
}
#endif
```

```
#include "Eleve.h"
#include "Cours.h"

void Cours::associer(un_module : string)
{
      /* A ECRIRE */
}

void Cours::set_titre(const string value)
      { titre = value ; }
const string Cours::get_titre()
      { return titre ; }
void Cours::set_module(const string value)
      { module = value ; }
const string Cours::get_module()
      { return module ; }
```

Traduction en objet (C++)


```
public class Cours {

   private string titre ;
   private string module ;

   protected eleves = new Vector() ;
      /* par exemple */

   public set_titre(String value)
      { titre = value ; }
   public get_titre()
      { return titre ; }
   public set_module(String value)
      { module = value ; }
   public get_module()
      { return module ; }
```

```
public Cours ()
  {

  }
   public associer(String m) {

         /* remplir ici */
    }

}
```

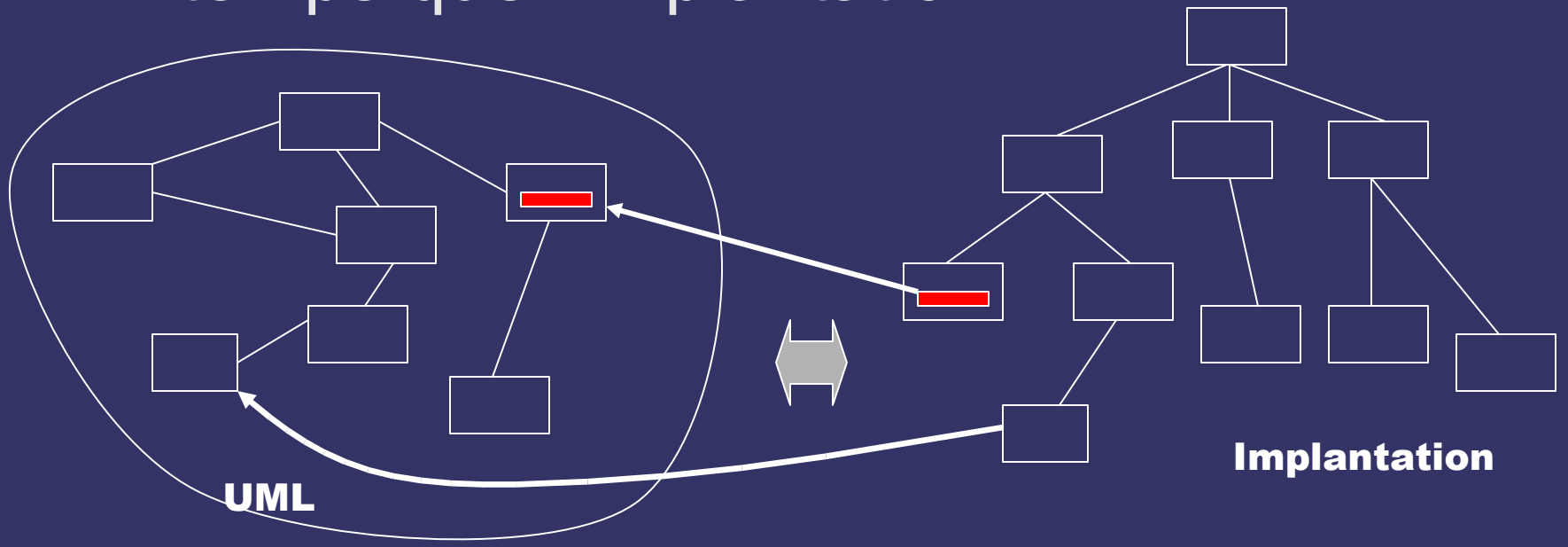
Traduction en relationnel


```
CREATE TABLE eleve (
      eleve_id NUMBER (5) ,
      annee DATE,
      PRIMARY KEY (eleve_id)
) ;

CREATE TABLE cours (
      eleve_id NUMBER (5)  REFERENCES eleve(eleve_id) ,
      cours_id NUMBER (5) ,
      titre CHAR (128) ,
      module CHAR(48) ,
      PRIMARY KEY (cours_id)
) ;
```

Nécessité de la rétro-ingénierie (reverse engineering)

- Faire évoluer le modèle en même temps que l'implantation



- Seuls les outils automatiques peuvent le faire

IX
Conclusions

Conclusions sur UML

- Propriétés d'UML
  - Unification des concepts de modélisation
  - Puissance d'expression
    - Nombreux formalismes (issus de méthodes existantes)
    - Mécanismes d'extension inclus
  - Description par un méta-modèle
    - Syntaxe et sémantique des modèles
  - Compromis formalisation / niveau d'abstraction
- Langage universel
  - Domaines d'application
    - Noyau + profils de spécialisation : ex. temps réel, modélisation activité, ...
  - Compromis acceptation / homogénéité-redondance
    - Forces importantes pour l'adoption d'un standard

Conclusions sur UML

- UML est un standard international : adopté un peu partout
  - les modèles sont simples, faciles à lire et à communiquer
  - mais il y en beaucoup, et beaucoup de variantes
  - difficulté pour des systèmes très complexes, d'où la nécessité d'outils
- Des outils puissants,
  - et l'impression qu'UML est une méthode
- Or UML est un ensemble de modèles, un langage
  - dénominateur commun : très (trop) riche
  - il faut adapter la méthode et l'utilisation d'UML au système à construire

Conclusion sur UML

- Nous avons vu le langage de description, qui encapsule une partie de la sémantique de description
- Il faut passer au méthodes, c'est à dire avoir une démarche de concption et d'utilisation des diagrammes

➔ Cours suivant

Bibliographie

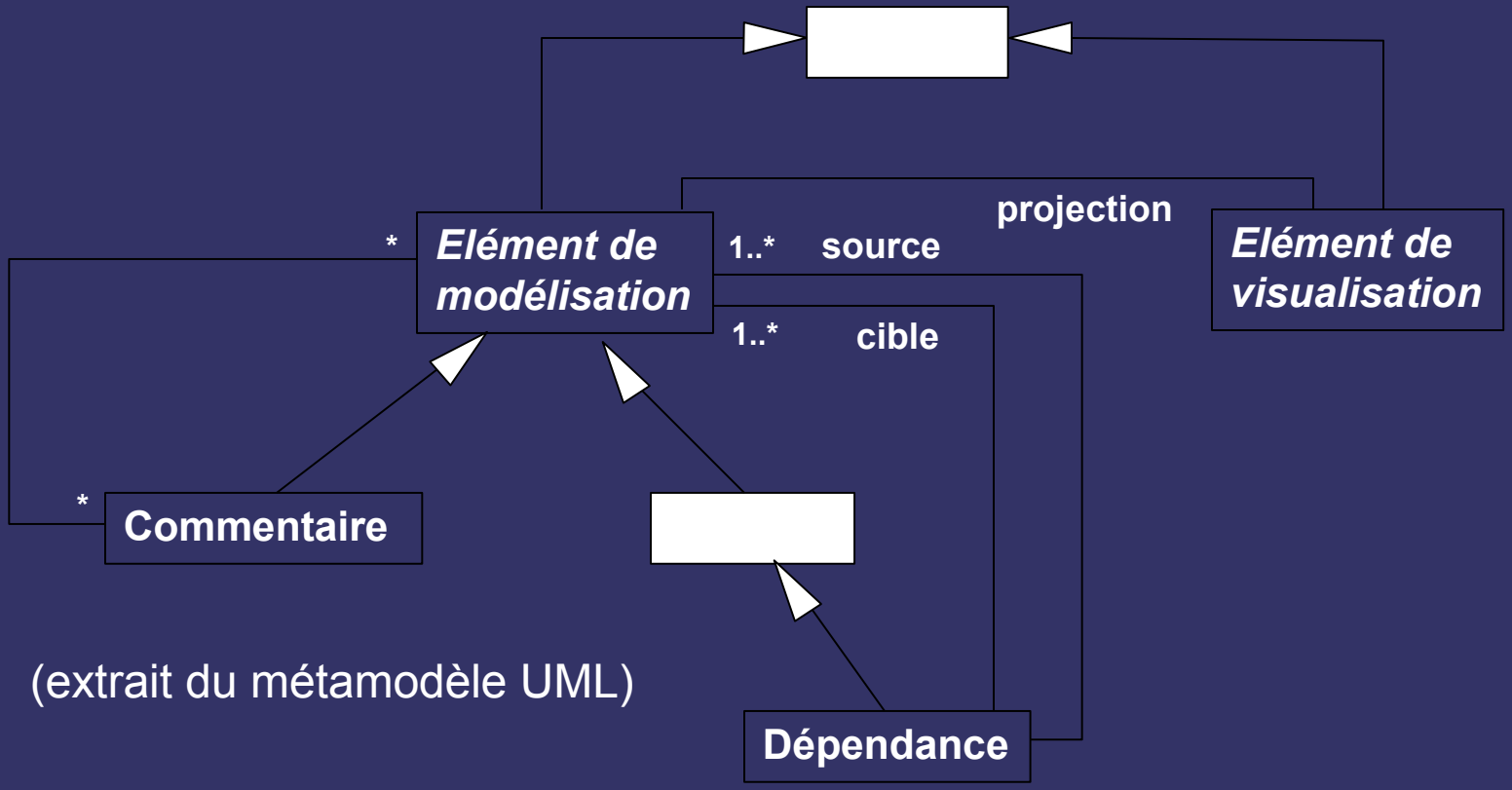
- Muller/Gaertner : « Modélisation objet avec UML », Eyrolles, 2000
- Fowler. UML 2.0
- Nombreux livres en français et anglais
- Web

Crédits – Remerciements

- P.-A. Muller
- F. Laforest (INSA)
- B. Morand (Caen)
- Documents Rational
- J.-L. Sourrouille (INSA)
- M. Fowler

Méta-modélisation

- Faciliter le travail de modélisation : décrire le modèle avec lui-même



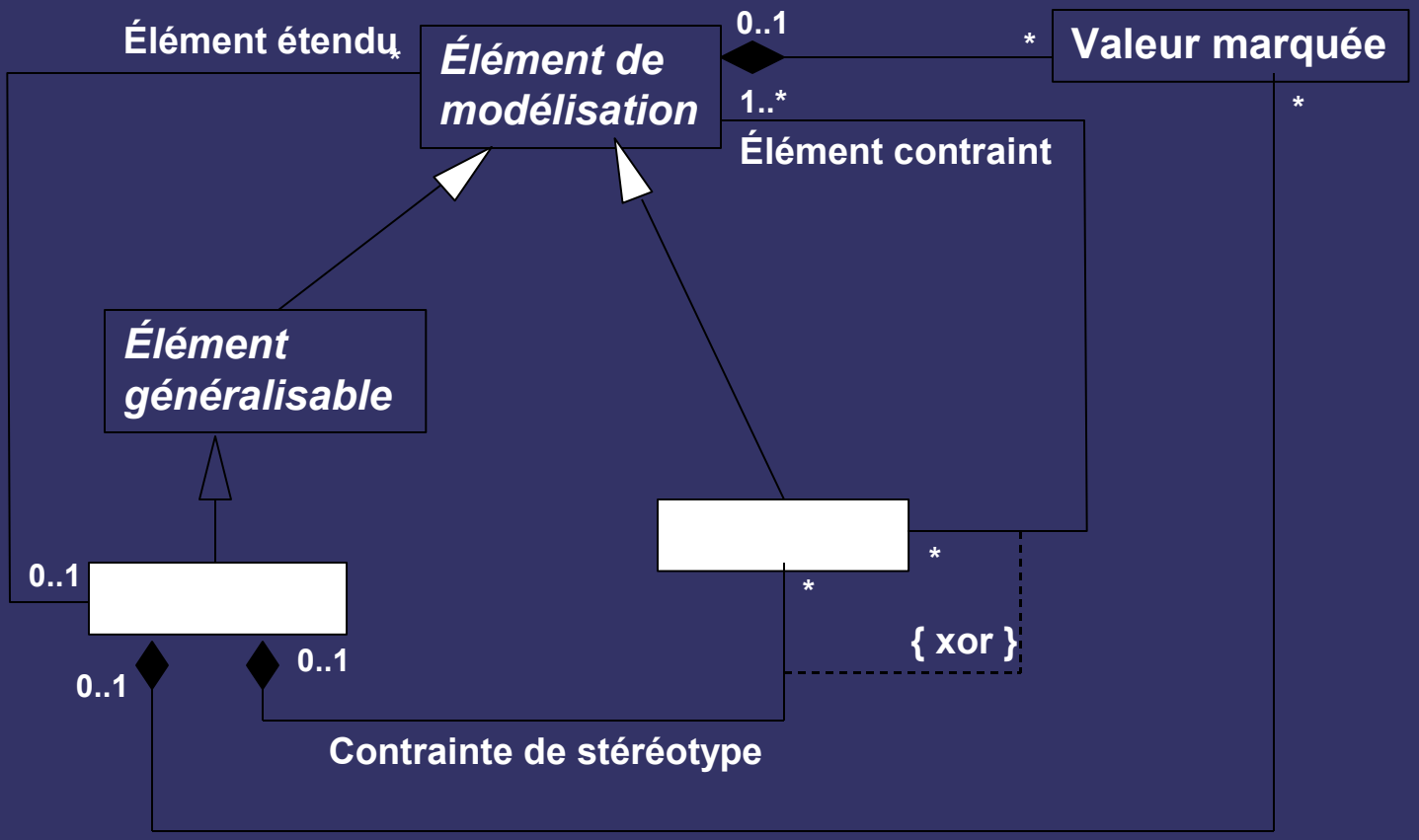
(extrait du métamodèle UML)

Étendre UML

- En utilisant les stéréotypes, contraintes, valeurs marquées
- Exemples de stéréotypes
  - stéréotypes de classes : boundary, control, entity, utility, exception
  - stéréotypes d'héritage : uses, extends
  - stéréotypes de composants : subsystem
- Chacun peut définir un nouveau stéréotype héritant des propriétés d'un stéréotype de base, complété par des contraintes et des valeurs marquées

  ➔le modèle est étendu, mais on reste dans le même cadre

Etendre UML


Élément étendu
*
Élément de modélisation
0..1
*
Valeur marquée
1..*
Élément contraint
*
Élément généralisable
0..1
*
*
{ xor }
0..1
0..1
Contrainte de stéréotype

Notion de collaboration

- Regroupement d'éléments mis en œuvre pour exprimer une réalisation d'un cas d'utilisation, dans un contexte donné (lien <<realize>>)
- Collaboration au niveau spécification : fournir un contexte
  - classes + associations, rôles joués (= restriction de classes à la collaboration)
    - classes : Ex. /Etudiant : Personne (restriction aux étudiants)
    - associations : noms de rôles
- Au niveau instance : une collaboration donne lieu à une ou plusieurs *interactions*
  - objets + liens (conformes aux rôles du niveau spécification)
  - + messages échangés entre les objets