Cours Python

—

Unified Modeling Language (UML)

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Télécommunications & Réseaux

Références

[1] P.-A. Muller and N. Gaertner, *Modélisation objet avec UML*.
Eyrolles, 2è ed., 2003.

[2] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*.
Addison-Wesley, 1999.

[3] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*.
Addison-Wesley, 1999.

[4] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*.
Addison-Wesley, 1999.

[5] M. Fowler, *UML 2.0*.
CampusPress Référence, 2004.

[6] OMG, “UML Resource Page.” http ://www.omg.org/uml/.

[7] OMG, “OMG Unified Modeling Language TM (OMG UML), version 2.5.”
http ://www.omg.org/spec/UML/2.5/.

[8] J. Rumbaugh *et al.*, *Modélisation et conception orientée objet*.
Masson, 1994.

[9] I. Jacobson *et al.*, *Object-Oriented Software Engineering : A Use Case Driven Approach*.
Addison-Wesley, 1992.

[10] G. Booch, *Object-Oriented Analysis and Design with Applications*.
Benjamin/Cummings, 2*nd* ed., 1994.

Historique

- 1970 : Premières méthodes de développement (Analyse Structurée de Yourdon 1979)
- 1967 : Premier langage orienté-objet : Simula-67 ;
- 1980 : Smalltalk, Objective C, C++, Eiffel, CLOS...
- 1988 : 1reméthode orientée-objet : Schlaer/Mellor (OOSA)
- 1991 : Coad/Yourdon (OOA), Booch (OOD), Rumbaugh (OMT), etc.
- 1992 : Jacobson (Objectory), et de nombreuses autres (plus de 50)
- 1994 : Tentative d'unification : Coleman (Fusion)
- 1994 : Booch + Rumbaugh, puis Jacobson (1995) : UML 0.8 (1995)
- 1996 : Appel d'offre de OMG (RFP) pour une approche normalisée pour les développements objets
- 1997 : Choix et normalisation de UML 1.0 (actuellement 2.0).

Principales versions d'UML

- UML 1.0 (01/1997) : soumission initiale de UML à l'OMG
- UML 1.1 (11/1997) : normalisation de UML
- UML 1.2 (06/1998) : changements cosmétiques
- UML 1.3 (03/2000) : modifications dans les cas d'utilisation et les diagrammes d'activités
- UML 1.4 (09/2001) : composants et profils
- UML 1.5 (04/2004) : langage d'action (*action semantics*)
- UML 2.0 (07/2005) : de nouveaux diagrammes, meilleure prise en compte de la dynamique
- UML 2.1.2 (11/2007), 2.2 (02/2009), 2.3 (05/2010), 2.4 (03/2011), 2.4.1 (08/2011)
- UML 2.5 (06/2015) : simplification (et fusion de infrastructure et superstructure en un seul document, environ 800 pages)

Qu'est-ce qu'un modèle ?

- Un modèle est une simplification de la réalité.
- On construit des modèles pour mieux comprendre le système en cours de développement.

  Pour un observateur A, M est un modèle de l'objet O, si M aide A à répondre aux questions qu'il se pose sur O. (Minsky)

- On construit des modèles pour les systèmes complexes car on ne peut pas comprendre de tels systèmes dans leur ensemble.
- Chaque modèle peut être décrit à différents niveaux de précision.
- Aucun modèle seul n'est suffisant. Tout système non élémentaire est mieux décrit par un petit ensemble de modèles presque indépendants.

Qu'est-ce qu'UML ?

- Un language de modélisation visuel ;
- Un langage pour spécifier, visualiser, construire et documenter les produits d'un développement logiciel ;
- Une notation et une sémantique ;
- Des diagrammes pour modéliser les aspects statiques, dynamiques et organisationnels ;
- Une unification des connaissances acquises dans le domaine de l'orienté-objet ;
- Compatible avec toutes les méthodes / procédés de développement ;
- Doit être outillé (outils support) ;
- UML n'est **pas** un langage de programmation.

Pourquoi le terme « unifié » ?

UML se veut une notation unifiée par rapport :

- aux méthodes et notations déjà proposées ;
- aux différentes phases du cycle de développement d'un logiciel ;
- aux domaines d'application ;
- aux langages et plateformes d'implantation ;
- aux différents procédés de développements ;
- aux concepts internes (explicitation du méta-modèle).

Les constituants d'UML

- **Vue statique du système (architecture du système) : 6 diagrammes structurels**
  - Diagramme de classe
  - Diagramme d'objet
  - Diagramme de paquetage (UML 2)
  - Diagramme de structure composite (UML 2)
  - Diagramme de composant (Vue organisationnelle)
  - Diagramme de déploiement (Vue de déploiement)
- **Vue dynamique (comportement) : 7 diagrammes comportementaux**
  - Diagramme de vue d'ensemble des interactions (UML 2)
  - Diagramme de séquence (fortement changé en UML 2)
  - Diagramme de communication (anciennement collaboration)
  - Diagramme de temps (UML 2)
  - Diagramme de machine à états
  - Diagramme d'activité
- **Autre diagramme :**
  - Diagramme de cas d'utilisation (vue fonctionnelle, interaction utilisateurs/système)
- **Mécanismes d'extension**

Utilisations possibles d'UML

Principalement trois utilisations possibles [5] :

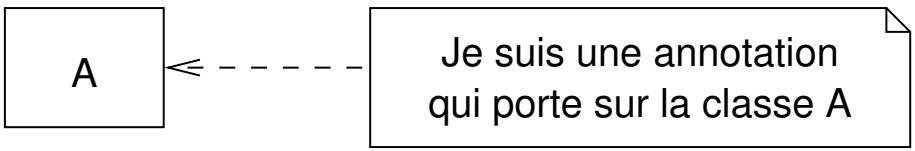
- **esquisse (*sketch*) :** communiquer avec les autres sur certains aspects
  - simplification du modèle : seuls les aspects importants sont présentés
  - échanger des idées, évaluer des alternatives (avant de coder), expliquer (après)
  - n'a pas pour but d'être complet
- **plan (*blueprint*) :** le modèle sert de base pour le programmeur
  - le modèle a pour but d'être complet
  - les choix de conception sont explicités
  - seuls les détails manquent (codage)
- **langage de programmation :** *le modèle est le code !*
  - pousser à l'extrême l'approche « plan »
    $\Longrightarrow$ mettre tous les détails dans le modèle
  - engendrer automatiquement le système à partir du modèle.

  Remarque : UML < 1.5 n'avait pas pour but d'être un langage de programmation.
  Possible depuis avec la notion de **langage d'action**.

Quelques conventions

La notation UML définit pour tous les diagrammes :

- des *commentaires* (ou *annotations*) qui permettent d'ajouter des informations sur un modèle.



  L'annotation est reliée à l'élément décrit par un trait interrompu.

- Les *stéréotypes* sont des mots entre chevrons qui permettent de compléter le vocabulaire UML.

  `«exception», «interface», «instanceOf»...`

- Les *contraintes* : les accolades sont utilisées pour ajouter une contrainte sur un modèle (modification de la sémantique du modèle).

  `{abstract}, {ordered}, {readOnly},` `{ x =` $\rho \times cos(\theta)$ `}`

Remarque : Voir les mécanismes d'extension (T. 36).

La vue statique

Objectifs : Décrire la structure statique du système constituée de :

- classes avec attributs (état des objets), opérations (comportement)... ;
- relations entre classes ;
- objets et liens (pour représenter un état particulier d'un système).

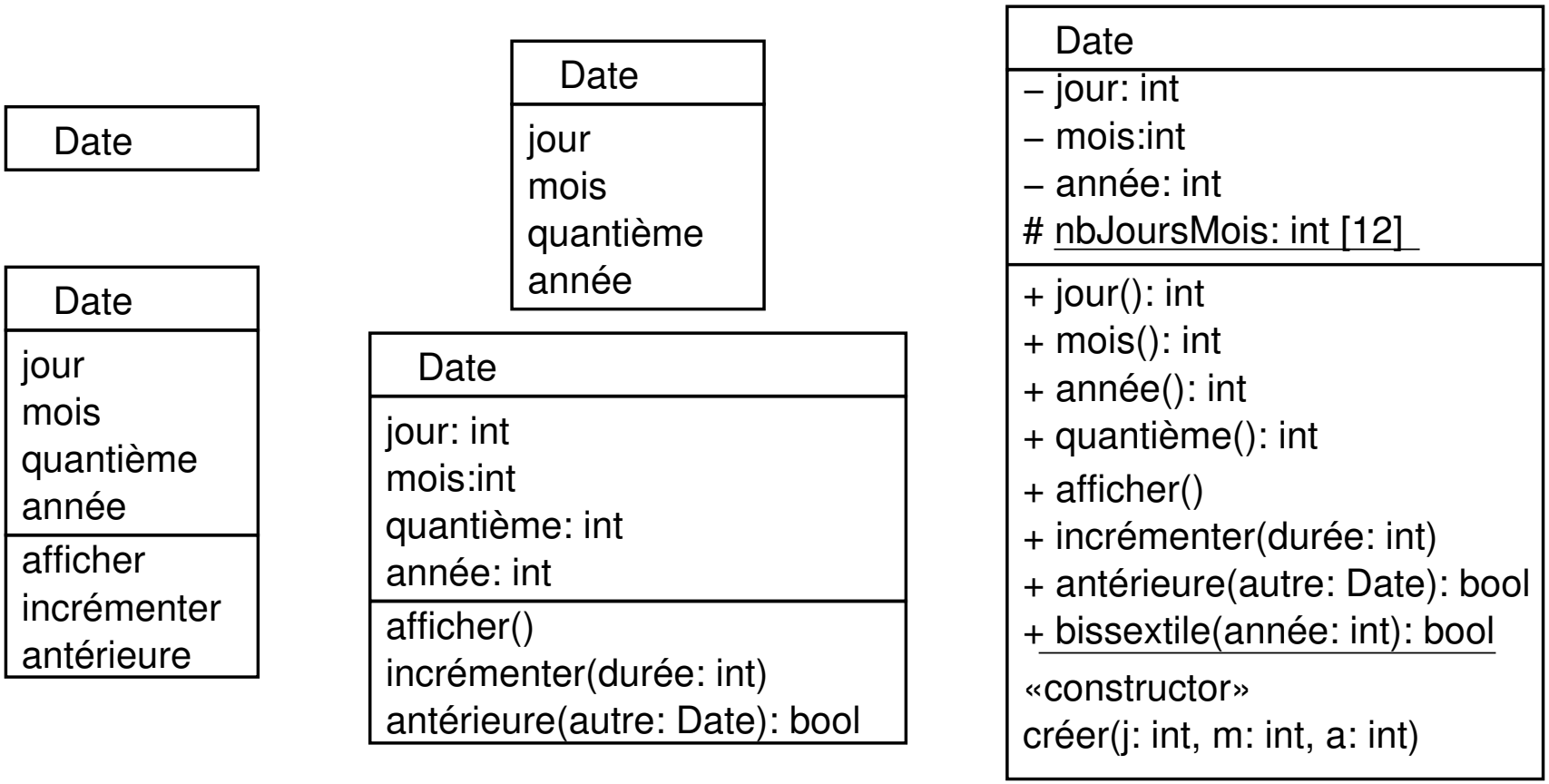
Remarque : Il y a complémentarité entre vue statique et vue dynamique :

- La vue statique définit les configurations possibles du système. (Mais elles ne sont pas nécessairement toutes atteignables.)
- La vue dynamique décrit les transitions possibles entre configurations.

Moyen :

- *Diagramme de classe :* décrit les classes et les relations ;
- *Diagramme d'objet :* décrit les objets et les liens. C'est une instance du diagramme de classe, une photographie d'un état particulier du système ;
- mais aussi diagrammes de vue de structure composite, de paquetage, de composant et de déploiement.

Différents niveaux de description d'une classe



Description des attributs

La forme générale de déclaration d'un attribut est :

```
[visibilité] nom [: type] [multiplicité] [= valeur-initiale] [{propriétés}]
```

```
origine                    -- seulement le nom
+ origine                  -- visibilité et nom
origine: Point = (0, 0)    -- nom, type et initialisation
nom : String [0..1]        -- nom, type et multiplicité
id: Integer {frozen}       -- nom, type et propriété
```

Les visibilités sont : + (**public**), – (**private**), # (**protected**) et ~ (paquetage).
Les propriétés sont :

- `changeable` : pas de restriction sur les modifications (défaut) ;
- `frozen` : pas de modification possible après initialisation (const de C++) ;
- `addOnly` : (si multiplicités > 1) seules de nouvelles valeurs peuvent être ajoutées, les anciennes ne peuvent être ni enlevées, ni modifiées.

Description des opérations

La forme générale de déclaration d'une opération est :

```
[visibilité] nom [(liste-paramètres)] [: type-retour]  [{propriétés}]
```

Les paramètres sont séparés par des virgules et sont de la forme :

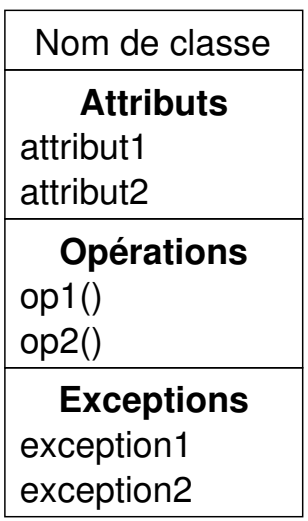
```
[direction] nom  : type  [= valeur-par-défaut]
```

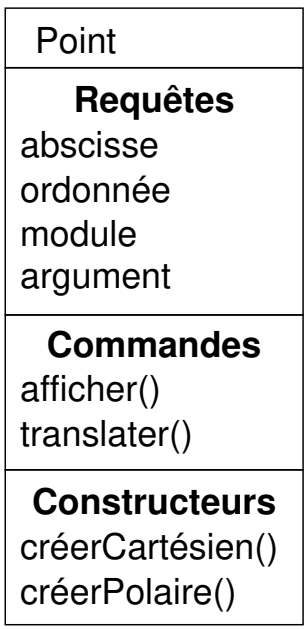
Les *directions* sont **in**, **out** et **inout** (sémantique d'Ada : entrée seule sans modification, sortie seule, entrée et sortie).
Les propriétés sont :

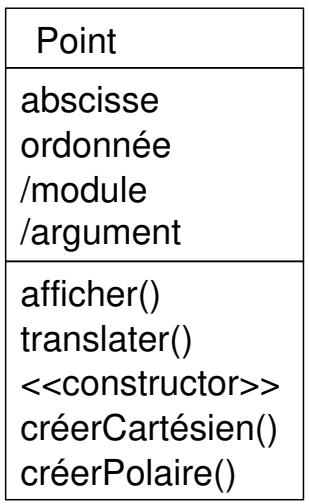
- `leaf` : opération non polymorphe (ne peut pas être redéfinie) ;
- `query` : fonction pure (sans effet de bord) ;
- `sequential`, `guarded`, `concurrent` : propriétés liées au parallélisme.

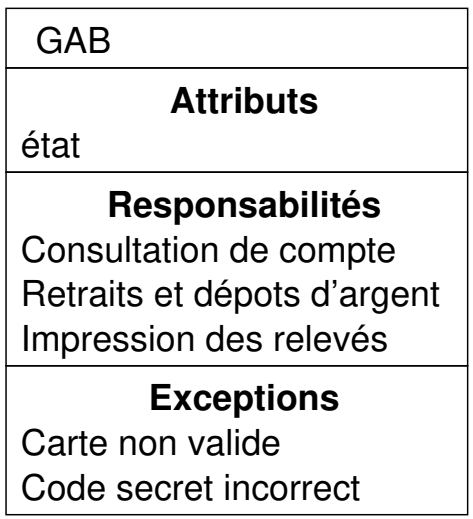
Autres représentations d'une classe

Il est possible d'adapter les rubriques qui décrivent une classe :









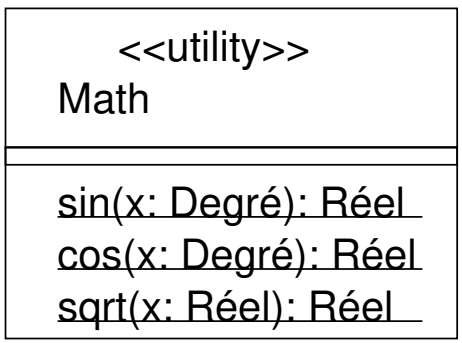
Le symbole / indique qu'un élément (attribut, etc.) peut être déduit des autres. On peut préciser entre accolades la règle de calcul.

Remarque : On peut choisir le nom du constructeur !

Classes de différentes natures

Des stéréotypes permettent de préciser la signification d'une classe :

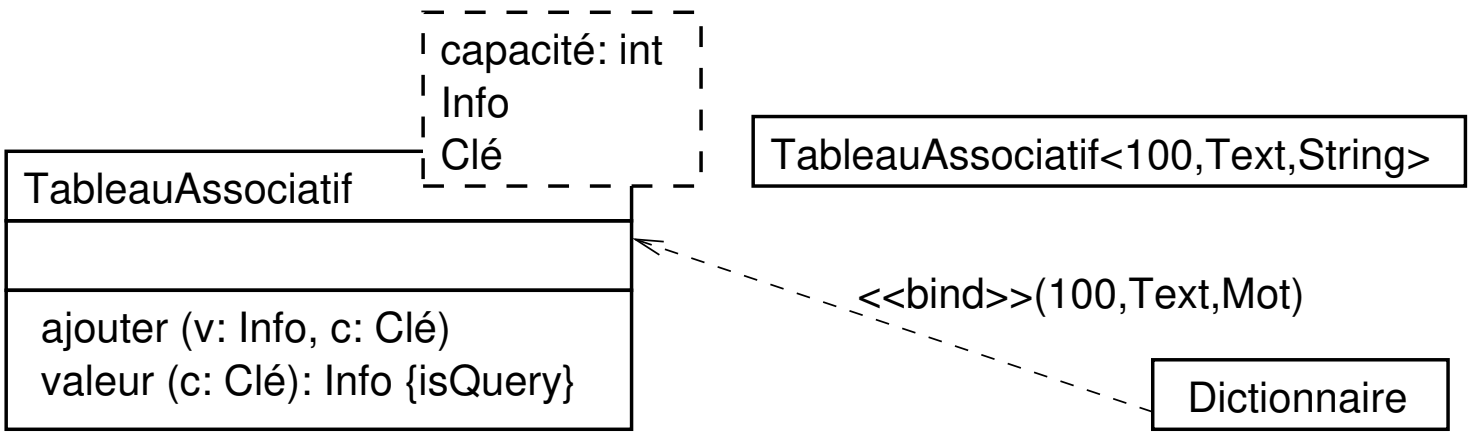
- «`utility`» : une classe qui correspond à une bibliothèque et rassemble essentiellement des méthodes de classe.



- «`exception`» : une classe qui décrit une exception.
- «`interface`» : il s'agit en fait d'une interface et non d'une classe.
  Remarque : On peut également utiliser «`type`».
- et bien d'autres...

Classe paramétrable (généricité)

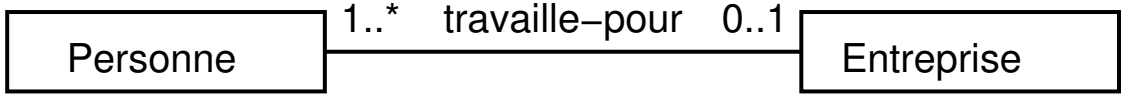
Une *classe paramétrable* est un modèle de classe paramétré par des classes et/ou des constantes (paramètre de généricité). On obtient une classe en fournissant une valeur pour chacun des paramètres de généricité de la classe paramétrable.
Exemple : Modélisation d'un tableau associatif paramétré par le nombre d'éléments qu'il peut contenir (capacité), le type des informations stockées et celui de la clé servant à retrouver une information.



Les relations d'association

Un système n'est pas constitué d'une seule classe mais d'un ensemble de classes qui interagissent entre elles et sont donc en relation.
On dit qu'il y a **relation d'association** entre deux classes, si la relation entre ces deux classes est stable, c'est-à-dire qu'elle dure dans le temps (non ponctuelle).

Exemple :



La relation d'association est :

- *bidirectionnelle* : elle peut être lue dans les deux sens ;
- *valuée* : sur les extrémités de la relation est précisé le nombre d'objets impliqués dans la relation. On parle de *multiplicité*.

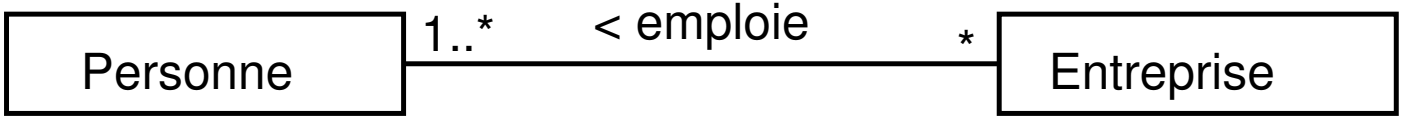
Sur l'exemple, on obtient :

- une personne travaille pour au plus une entreprise.
- une entreprise emploie plusieurs personnes, au moins une.

Nommer une relation

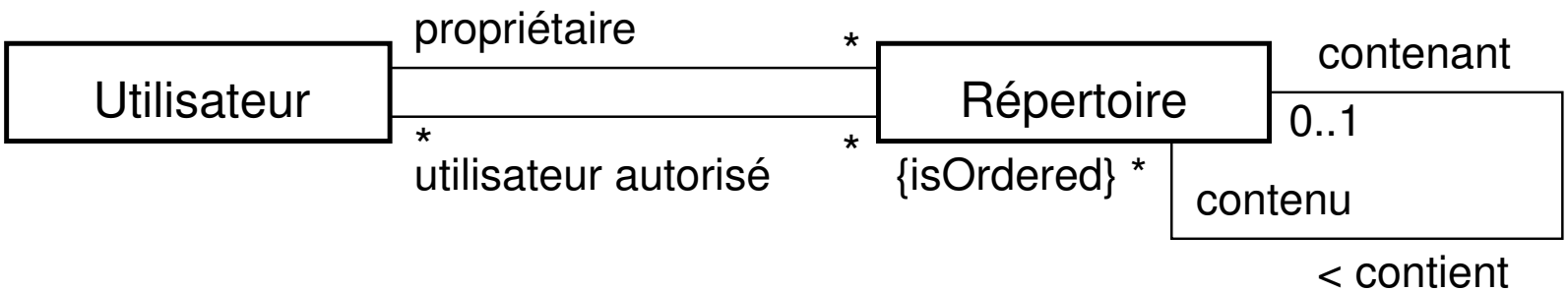
Il est possible de préciser (non exclusif) :

- le **nom de la relation** (par convention, les relations se lisent de gauche à droite et de haut en bas).
  Si le sens de lecture n'est pas naturel, il faut ajouter une flèche.



- le **rôle** que joue un objet dans la relation ;

Conseil : Préciser les rôles pour une relation réflexive.

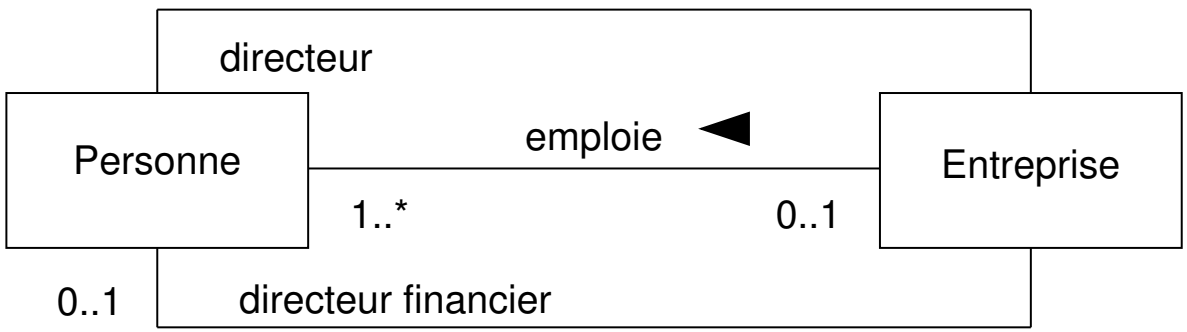


Multiplicité d'une association

La multiplicité est notée par une liste de possibilités séparées par des virgules. Une possibilité est :

- un entier : le nombre exact d'objets (ex : 4) ;
- un intervalle (1..4). Le caractère optionnel est noté par l'intervalle 0..1 ;
- $*$: un nombre quelconque y compris 0 ;
- $1..*$: un nombre quelconque hors 0.

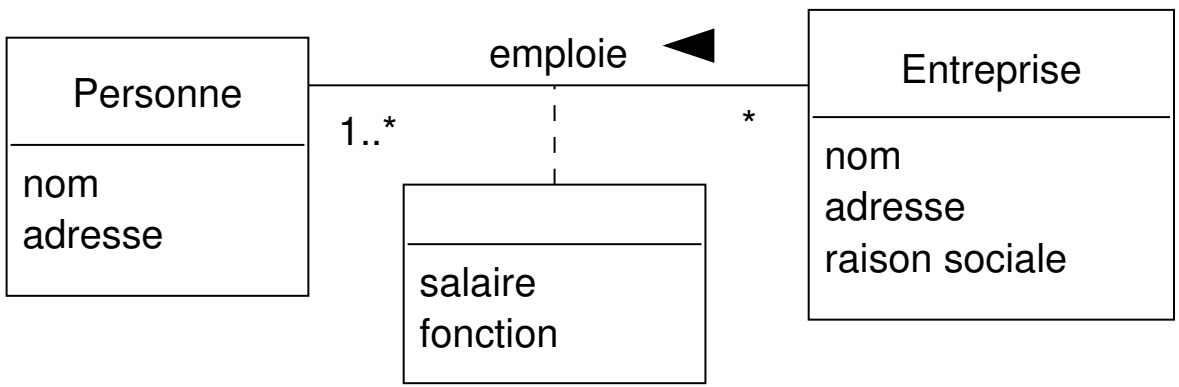
La multiplicité par défaut est (exactement) 1.



Remarque : La multiplicité peut ne pas être respectée en régime transitoire (création ou destruction d'un objet).

Attributs de relation

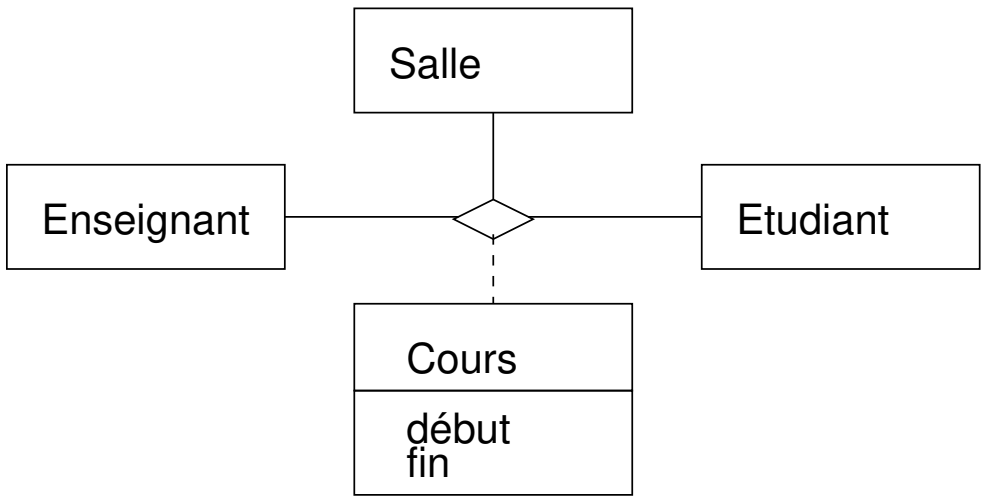
Définition : Un attribut de relation est un attribut qui caractérise la relation et pas seulement une de ses classes extrémités.
Remarque : Dans le cas d'une multiplicité 1, il est possible *mais non souhaitable* d'attacher l'attribut de relation à la classe (salaire sur Personne si elle ne peut travailler que dans une seule entreprise).



Remarque : Les attributs de relations peuvent être promus au rang de classe. Ici, une classe Poste, avec des méthodes telles que travailler, etc.

Relation ternaire et n-aire

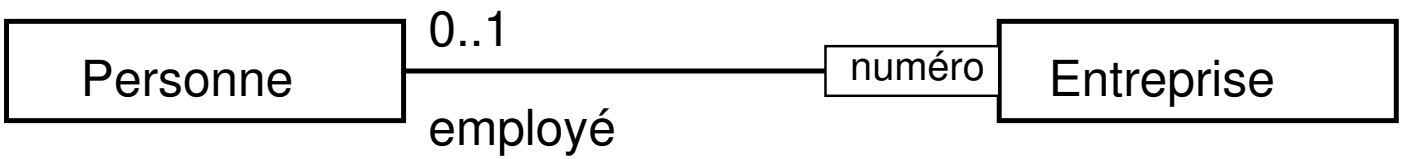
Une relation peut être ternaire et plus généralement n-aire.



Conseil : Éviter les associations n-aires.
On peut promouvoir une relation au rang de classe et imposer une contrainte sur ses relations (ici la classe Cours).

Qualification

Un *qualificatif* est un attribut spécial placé sur une extrémité d'une relation.



Le qualificatif « numéro » est une clé de la classe Entreprise qui permet d'atteindre un jeu d'objet Personne (ici au plus un).

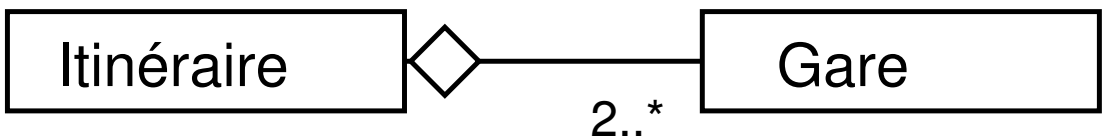
Intérêt : La qualification améliore la précision sémantique de la relation :

- elle *réduit la multiplicité* effective ($*$ est transformée en `0..1`, `numéro` est ici équivalent à une clé au sens base de données) ;
- elle *améliore la navigation* dans le réseau des objets (pour désigner une personne, il suffit d'avoir son numéro).

Remarque : Ceci permet de montrer que le numéro de la personne dans l'entreprise est unique. Comment l'indiquer si numéro est un attribut de Personne ou de la relation ?

La relation d'agrégation

Définition : Une relation d'agrégation est un cas particulier de relation d'association. C'est une association déséquilibrée, où une classe joue un rôle prépondérant. Elle correspond généralement à une relation tout ou parties (composé/composant).
Attention : Aucune sémantique précise n'est associée à l'agrégation. C'est une nuance que le concepteur peut faire apparaître.
Exemple :



Indication 1 : La définition d'une opération d'un objet agrégat repose sur les opérations des objets agrégés (ex : segment et points extrémités).
Indication 2 : Il y a agrégation si le « tout » dépend de l'existance de ses « parties » pour avoir un sens.

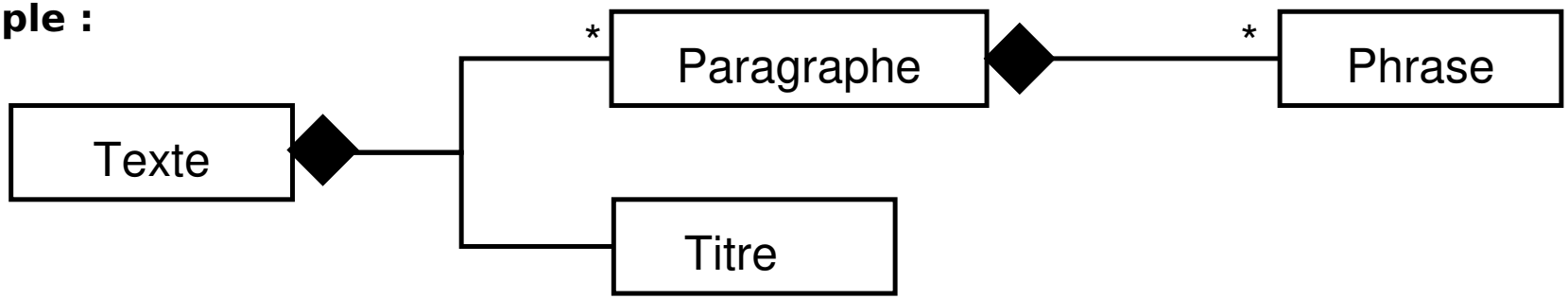
La relation de composition

La relation de composition est un cas particulier de la relation d'agrégation avec une sémantique plus forte :

> *Les durées de vie du composé et de ses composants sont liées.*

Conséquence : Un composant ne peut pas être partagé.

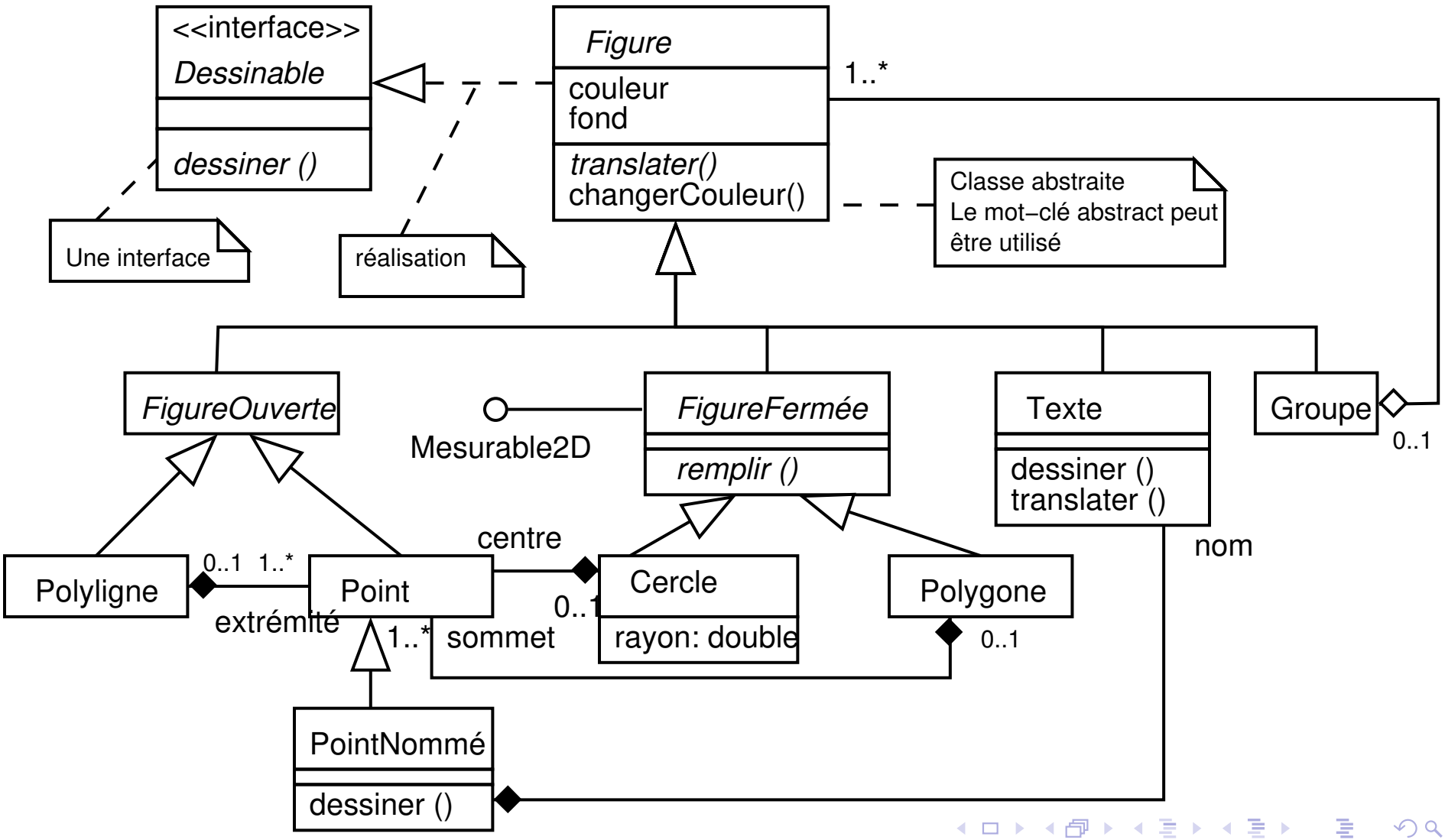
Exemple :



Remarque : Supprimer un texte, supprime son titre et ses paragraphes.

Convention : Les attributs (listés dans la rubrique attribut) correspondent à une relation de composition.

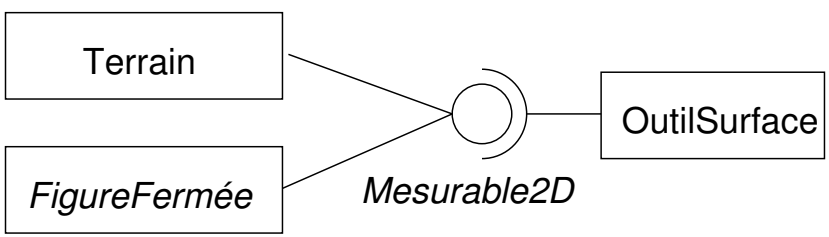
Relation de généralisation et relation de réalisation

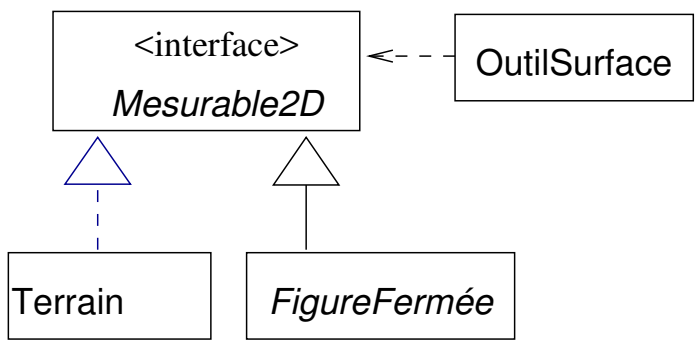


Relation de généralisation/spécialisation

Quelques remarques :

- Comme la relation d'héritage de Java, la relation de généralisation/spécialisation est une relation de sous-typage.
- Une classe abstraite, une méthode retardée ou une interface sont notées en italique. Il est cependant possible d'utiliser la contrainte `{abstract}`.
- `{polymorphic}` indique qu'une méthode est polymorphe (défaut).
- `{leaf}` interdit la redéfinition ou la dérivation (**`final`** en Java).
- les interfaces peuvent être représentées par un simple cercle avec leur nom dessous (notation dite balle-et-support ou sucette).





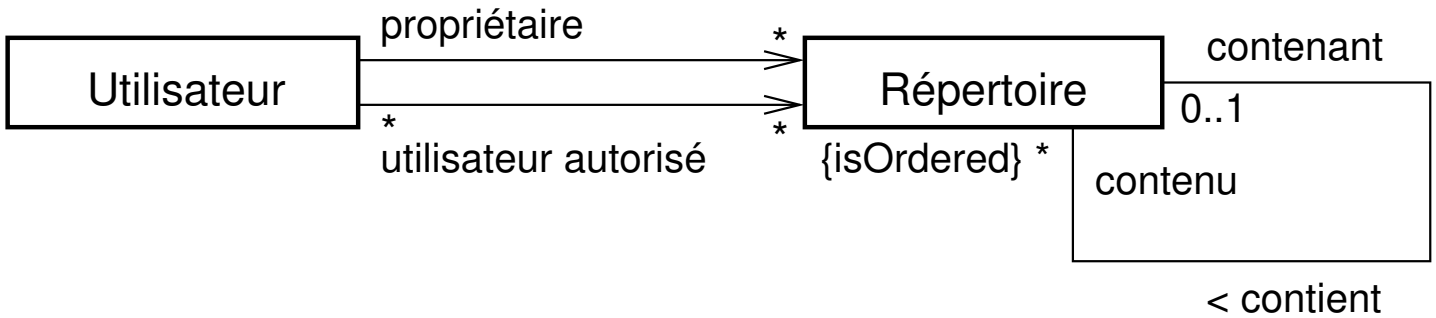
La relation de dépendance

Les relations d'association, d'agrégation, de composition, de généralisation et de réalisation sont des relations de dépendance particulières qui ont une sémantique forte et ont donc une notation particulière.
Il existe d'autres relations de dépendance qui sont surtout utiles dans les phases de conception / implantation. Elles n'ont pas de notation particulière (flèche avec traits interrompus) et se décrivent en utilisant un *stéréotype* :

- **bind :** instanciation des paramètres génériques ;
- **call :** une méthode d'une classe appelle une méthode d'une autre ;
- **derive :** un élément peut être calculé à partir d'un autre élément ;
- **instantiate :** une méthode d'une classe crée des instances d'une autre classe ;
- **parameter :** une relation entre une méthode et ses paramètres ;
- **trace :** lien entre deux modèles de niveaux différents ;
- **use :** un élément nécessite la présence d'un autre élément pour fonctionner correctement ;
- ...

Navigation dans un diagramme de classe

Les relations sont bidirectionnelles. Cependant, il est possible de préciser un sens de navigation.



Un utilisateur a accès à ses répertoires (donc à leurs caractéristiques) mais un répertoire ne peut pas accéder à son propriétaire.

Attention : Ce choix est introduit dans les phases de conception / implantation et n'a pas à être fait en phase d'analyse.

Dans le cas de l'agrégation ou de la composition, peut-on imposer un sens de navigation du composant vers le composé ?

Est-ce que la navigation a un sens pour la relation de généralisation ?

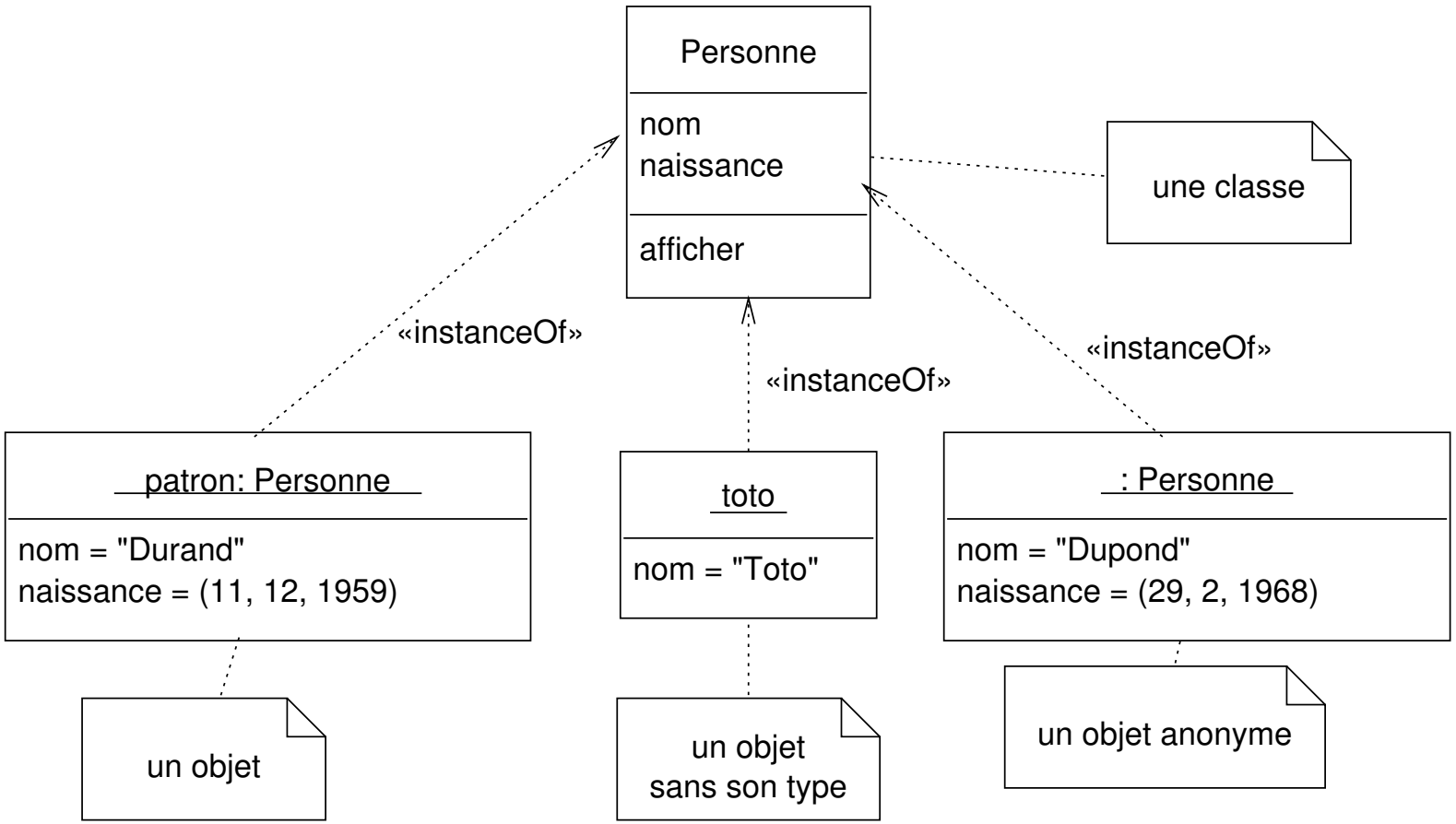
Le diagramme d'objet

Définition : Un diagramme d'objet est une instance du diagramme de classes. Il comprend :

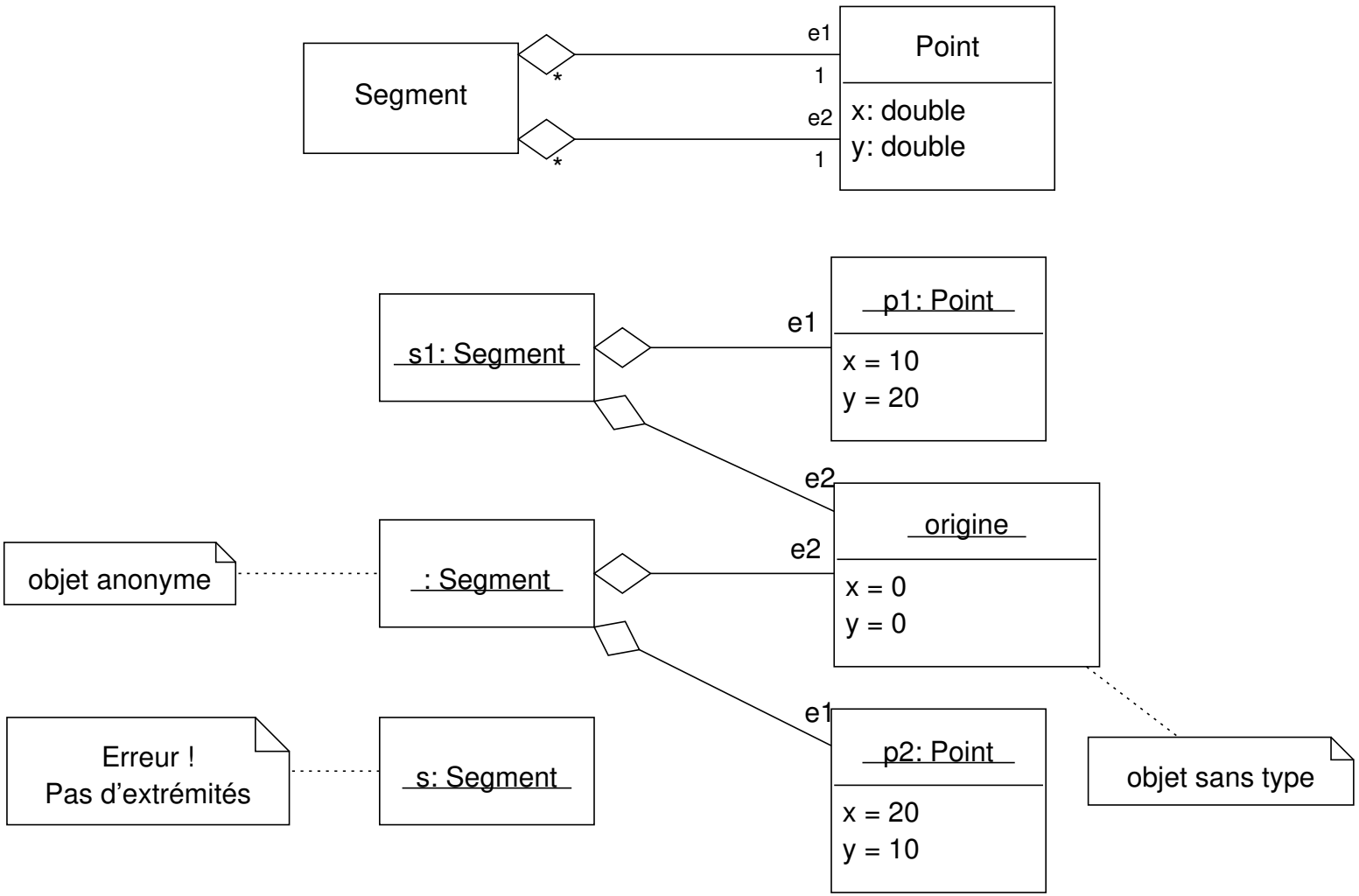
- des objets, instances des classes. Un objet possède un état propre (il a une valeur pour chaque attribut de sa classe) ;
- des liens, instances des relations, relient les objets.

Le diagramme d'objet doit respecter le diagramme de classe : si deux objets sont liés, c'est que leurs classes sont en relation (avec une contrainte de multiplicité compatible !).
Remarque : La représentation des objets en UML sera approfondie lors de la présentation du diagramme de communication (voir T. 60) qui est en fait une extension du diagramme d'objet.

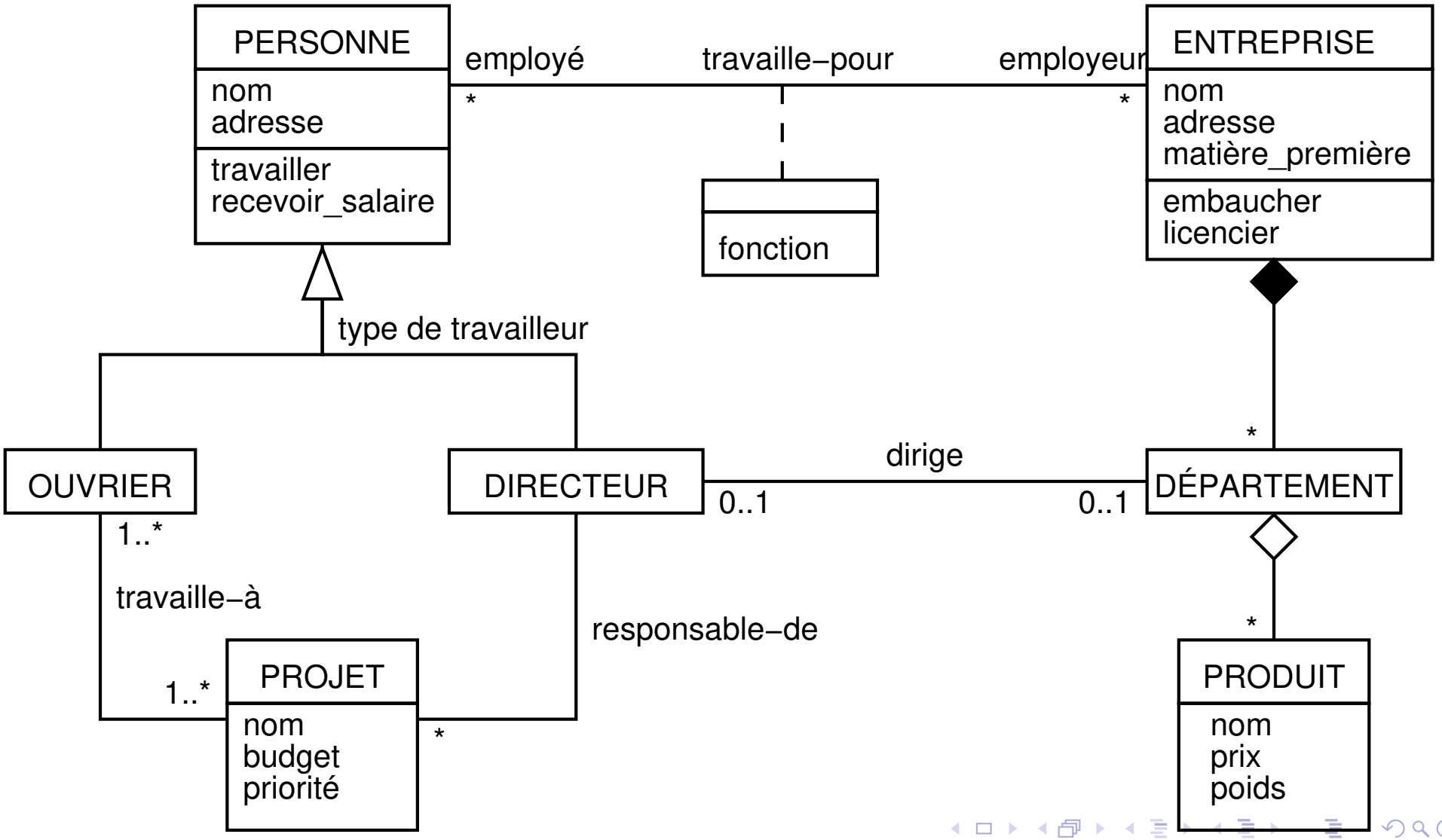
Représentation des objets



Exemple de diagramme d'objet



Exemple de diagramme de classe



Mécanismes d'extension

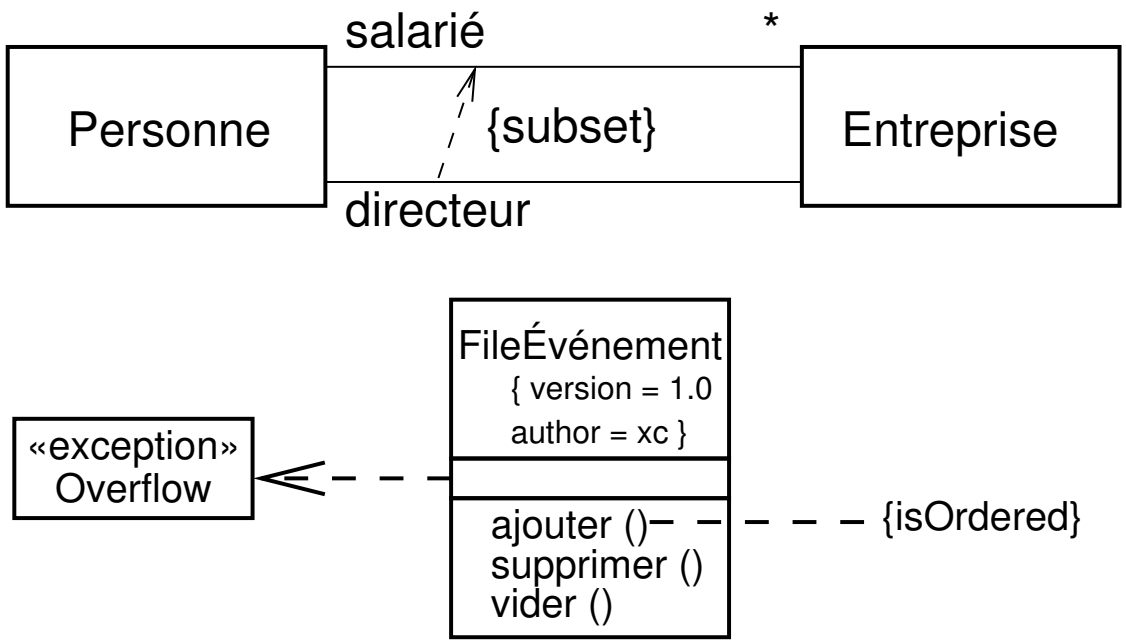
Principe : UML n'est pas un langage fermé. Il peut être étendu en utilisant trois constructions :

- les *stéréotypes* étendent le vocabulaire (mots-clés) de UML («`invariant`», «`instanceOf`», «`constructor`», «`exception`», etc.) ;
- les *valeurs marquées* (tagged values) ajoutent des attributs tels que la version, l'auteur, des choix de génération de code, etc.
- les *contraintes* étendent la sémantique de UML en en modifiant les règles ou en en ajoutant de nouvelles. Elles sont exprimées dans des annotations ou par des textes entre accolades.

Attention : Utiliser des extensions signifie s'éloigner de la norme UML et définir un dialecte avec tous les avantages et inconvénients associés.
Un tel dialecte s'appelle un **profil**.
Remarque : Ces mécanismes d'extension ne sont pas spécifiques du diagramme de classes.

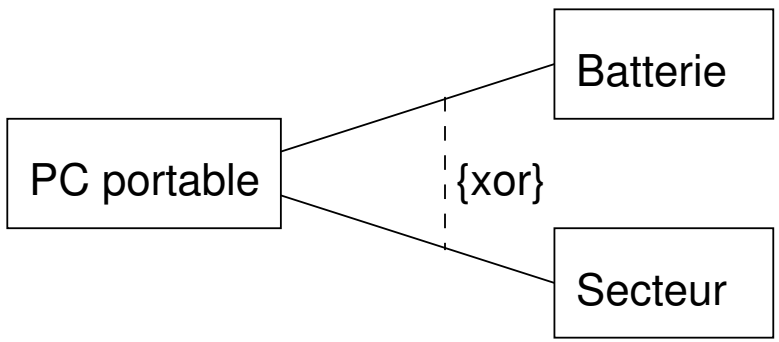
Les contraintes

Définition : Une contrainte exprime une propriété sur des entités UML (objets, classes, attributs, liens, relations).
Notation : Une contrainte est notée entre { }. Le texte peut être libre ou formel (par exemple OCL – cf T. 40 – ou langage de programmation).

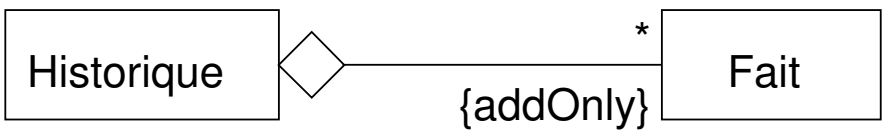


Autres exemples de contraintes

Cas de deux associations mutuellement exclusives :



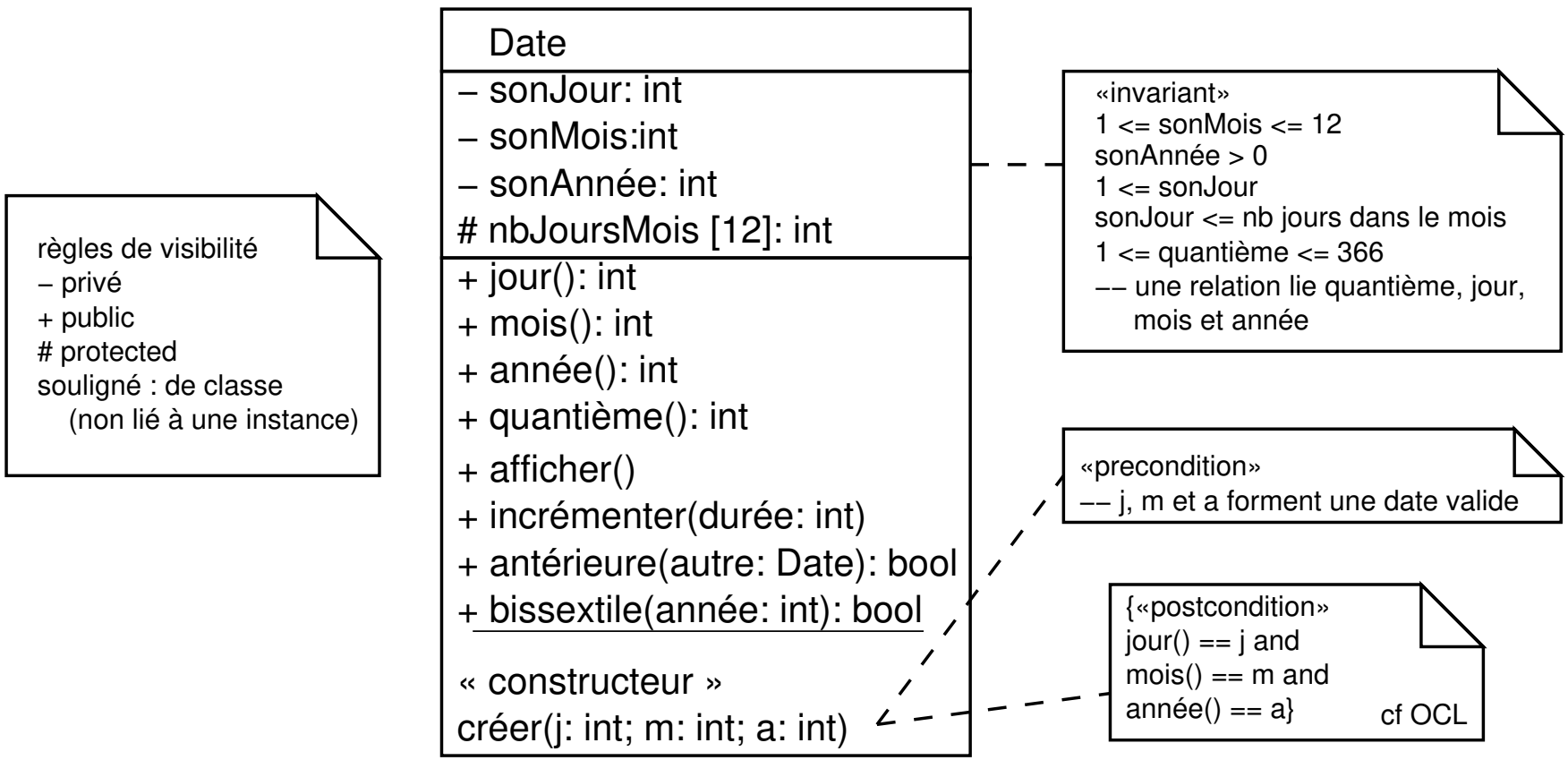
Remarque : Ceci évite d'introduire des sous-classes artificielles.
Préciser une contrainte sur une extrémité d'une association :



Remarque : On peut également utiliser : `frozen` et `changeable` (défaut).
Mais aussi `ordered` et `unique`.

Programmation par contrat (exemple de contraintes)

La programmation par contrat est possible grâce à des stéréotypes normalisés («`precondition`», «`postcondition`», «`invariant`»).



OCL : Object Constraint Language

OCL : Langage formel pour l'expression de contraintes, normalisé par l'OMG. Il est déclaratif, typé et sans effet de bord (sauf depuis 2.0 !).
But : Exprimer les contraintes d'un système qui ne peuvent pas être spécifiées par les notations d'UML comme par exemple :

- les invariants d'une classe ou d'un type ;
- les pré- et post-conditions d'opération ;
- les contraintes au sein d'une opération ;
- les expressions de navigation : contraintes pour représenter les chemins au sein de la structure de classes.

Les contraintes s'expriment dans le modèle et s'appliquent aux instances.
Elles ne peuvent pas modifier les instances car OCL est *sans effet de bord*...
$\Longrightarrow$ Elles ne doivent pas modifier les instances !

OCL : Contrats de la classe Date

```
context Date inv DateValide :
    self.année() > 0
    and 1 <= self.mois() and self.mois() <= 12
    and 1 <= self.jour() and self.mois() <=
        if self.mois() = 2 and Date::bissextile(self.année()) then
            29
        else
            Date::nbJoursMois[self.mois()]
        endif
    -- idem pour le quantième

context Date inv RelationQuantieme :
    -- exprimer la relation qui lie quantième à jour, mois et année

context Date::créer(j: int, m: int, a: int)
    pre annéeValide : a > 0
    pre moisValide : 1 <= m and m <= 12
    pre jourValide : 1 <= j and j <= if m = 2 and ...
    post initialisée : jour() = j and mois() = m and annee() = a

context Date::incrémenterAnnée()
    post self.année() = self@pre.année() + 1
    -- ou : post self.année() = self.année()@pre + 1
```

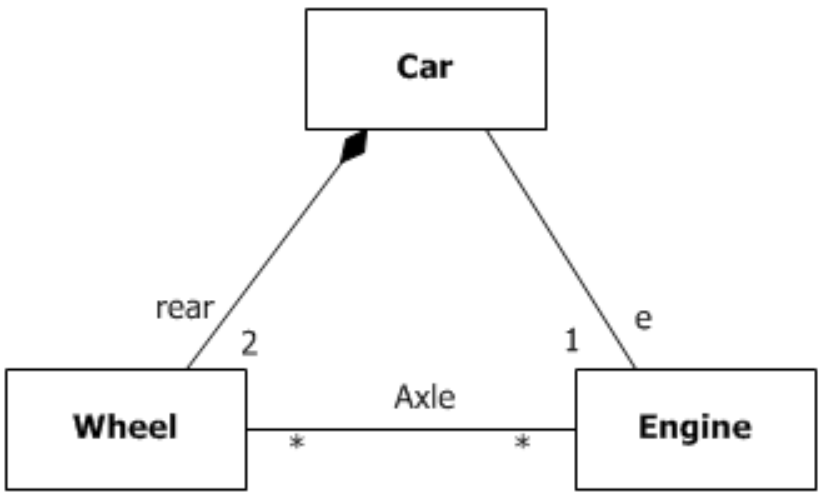
OCL et navigation

Le langage OCL permet de parcourir le diagramme de classe en suivant les relations (à condition de respecter les sens de navigation).
Pour les multiplicités > 1, des opérateurs prédéfinis permettent de manipuler le jeu d'instances correspondant :

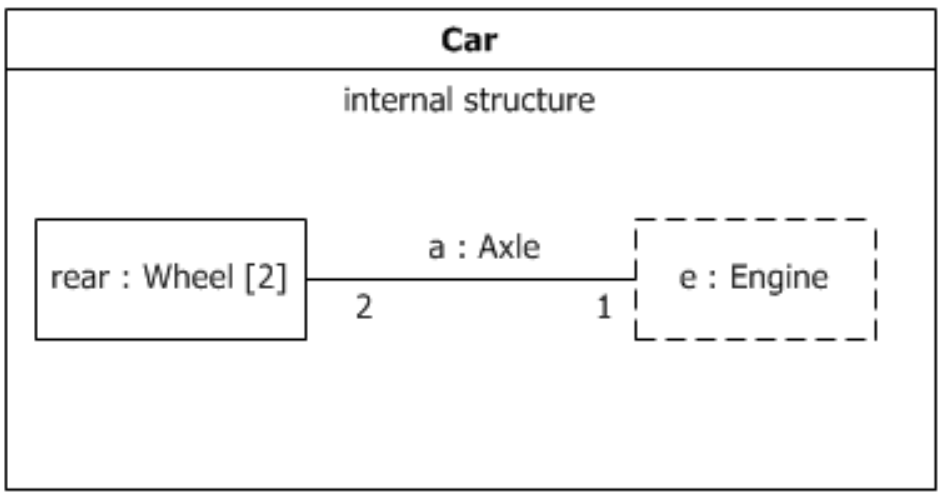
```
context Personne
    inv DeuxParentsMax: -- au plus deux parents
        self.parents->size() <= 2
    inv enfantDeSesParents: -- doit être un enfant de chacun de ses parents
        self.parents->forAll(p: Personne | p.enfants->count(self) = 1)
        -- ou : self.parents->forAll(p | p.enfants->count(self) = 1)
        -- ou : self.parents->forAll(enfants->count(self) = 1)

context Personne::estEnfantDe(p: Personne): boolean
    post: result = (self.parents->count(p) = 1)
    post: result = (p.enfants->count(self) = 1) -- redondant, inv. de Personne

context Personne::nbFilles(): boolean
    post result = self.enfants->select(sexe = #feminin)->size()
...
```

Diagramme de structure composite

Objectif : décrire l'organisation interne d'un élément statique complexe (p. ex. d'une classe).



(i)

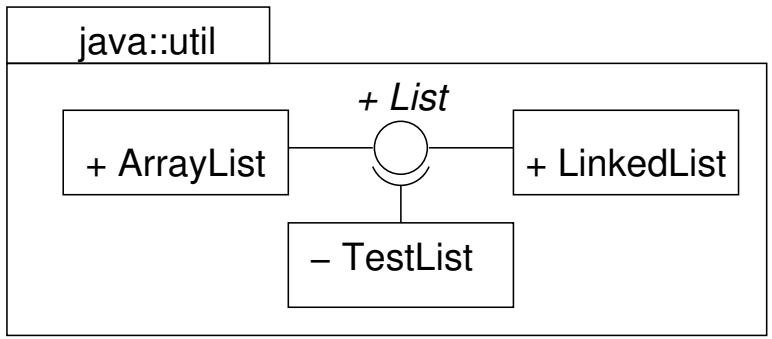


(ii)

- Extrait de [7, p. 185]
- Avec une structure composite (ii), on peut spécifier des contraintes locales (ici les multiplicités).
- trait plein = composition et trait interrompu = association/agrégation

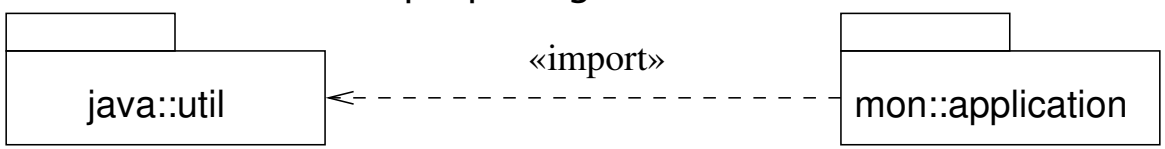
Diagramme de paquetage

Objectif : Montrer l'organisation logique des éléments du modèle.
Définition : Un paquetage est un regroupement sémantique d'éléments de modèles.



Relations entre paquetages :

- «`import`» : donne accès aux éléments d'un autre paquetage. Ces éléments sont considérés publics, donc accessibles d'autres paquetages.

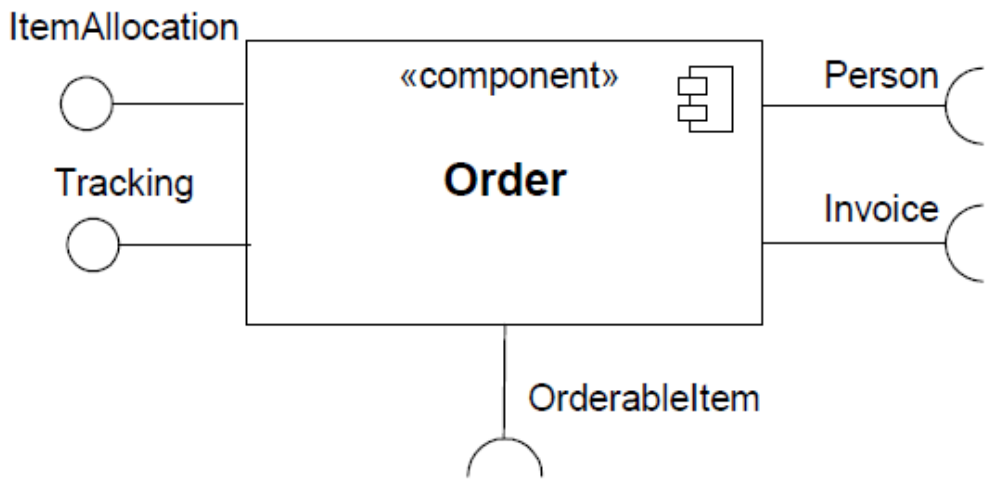


- «`access`» : comme «import» mais les éléments importés sont considérés privés.
- «`merge`» : copier virtuellement les éléments d'un paquetage dans un autre (fusion).

Diagramme de composant

Objectif : décrire les composants et leurs dépendances dans l'environnement de réalisation.
Un **composant** est un élément physique qui :

- représente une partie implémentée d'un système ;
  - «`document`» : un document quelconque ;
  - «`executable`» : un programme qui peut s'exécuter (sur un nœud) ;
  - «`file`» : un document contenant du code ou des données ;
  - «`library`» : une bibliothèque statique ou dynamique.
- réalise des interfaces (interfaces fournies) ;
- nécessite d'autres composants (interfaces requises) ;



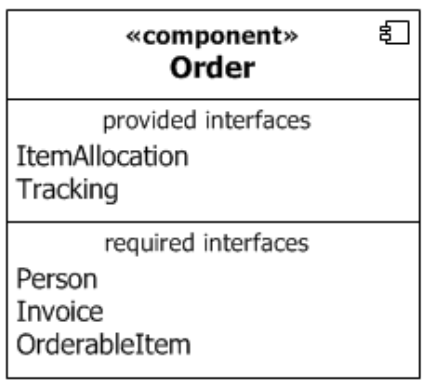
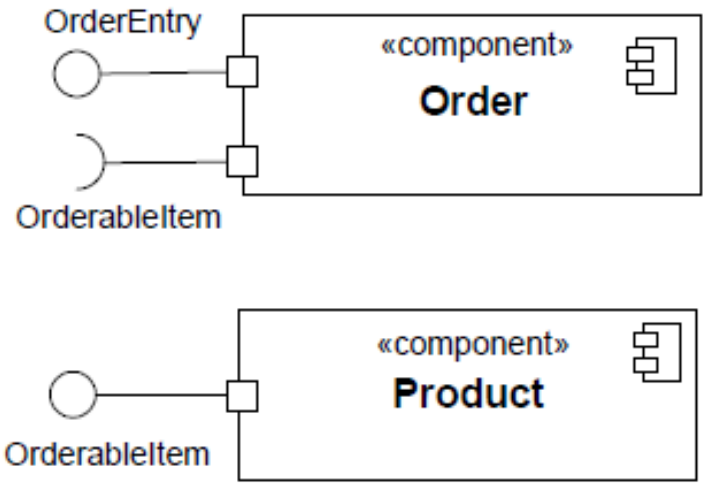


Diagramme de composants

Liens entre composants



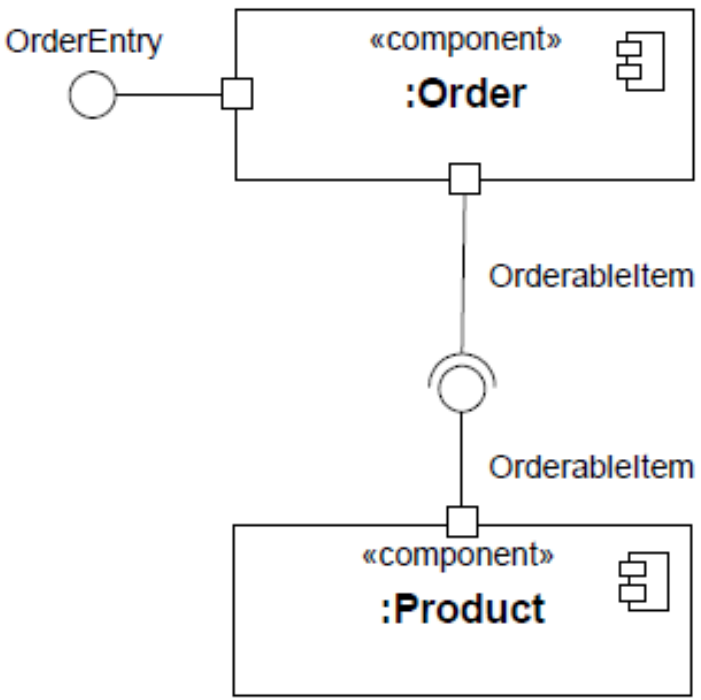


Diagramme de composants

Structure interne

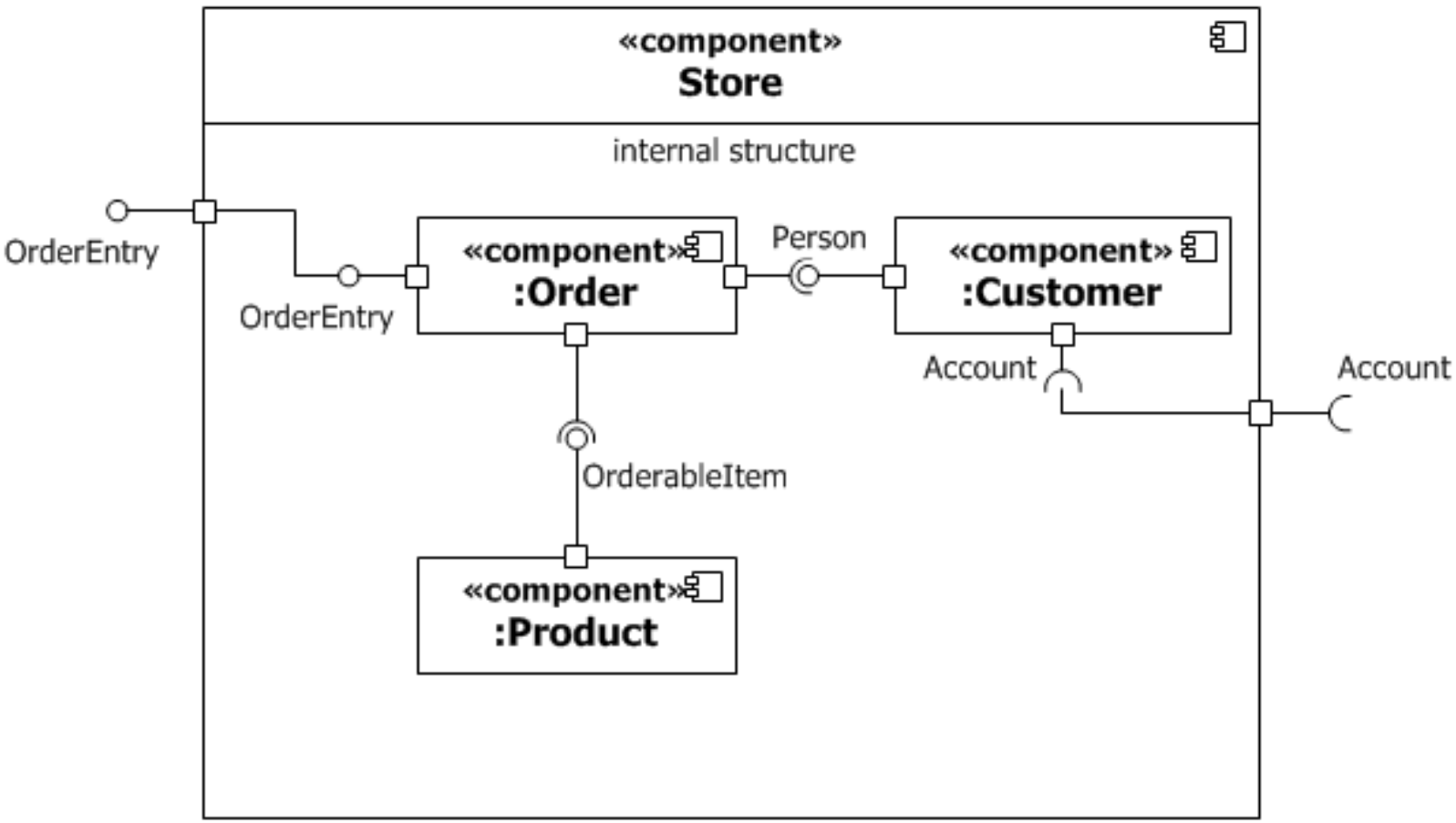
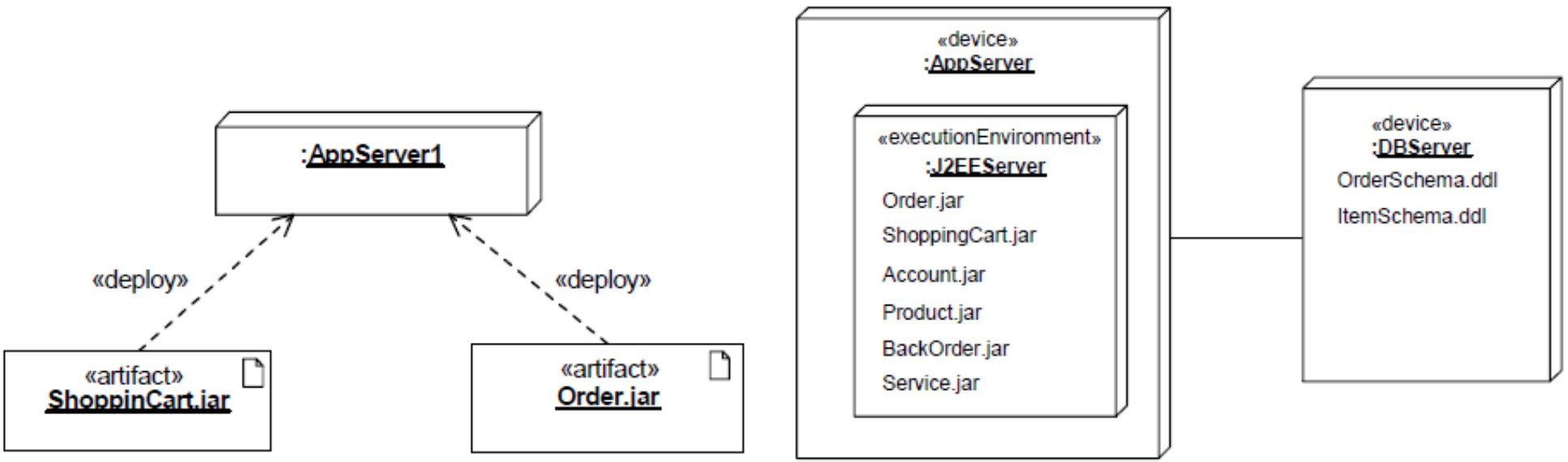


Diagramme de déploiement

Objectif : décrire :

- la disposition physique des différents matériels (les nœuds) qui font partie du système et
- la répartition des artefacts qui « vivent » sur ces matériels.

Un **artefact** est une unité d'implémentation physique (p. ex. un fichier, une instance de composant, un processus ou un objet).
Un **nœud** est une ressource liée à l'exécution (ordinateur, périphérique, mémoire...).



Construire le diagramme de classe (OMT)

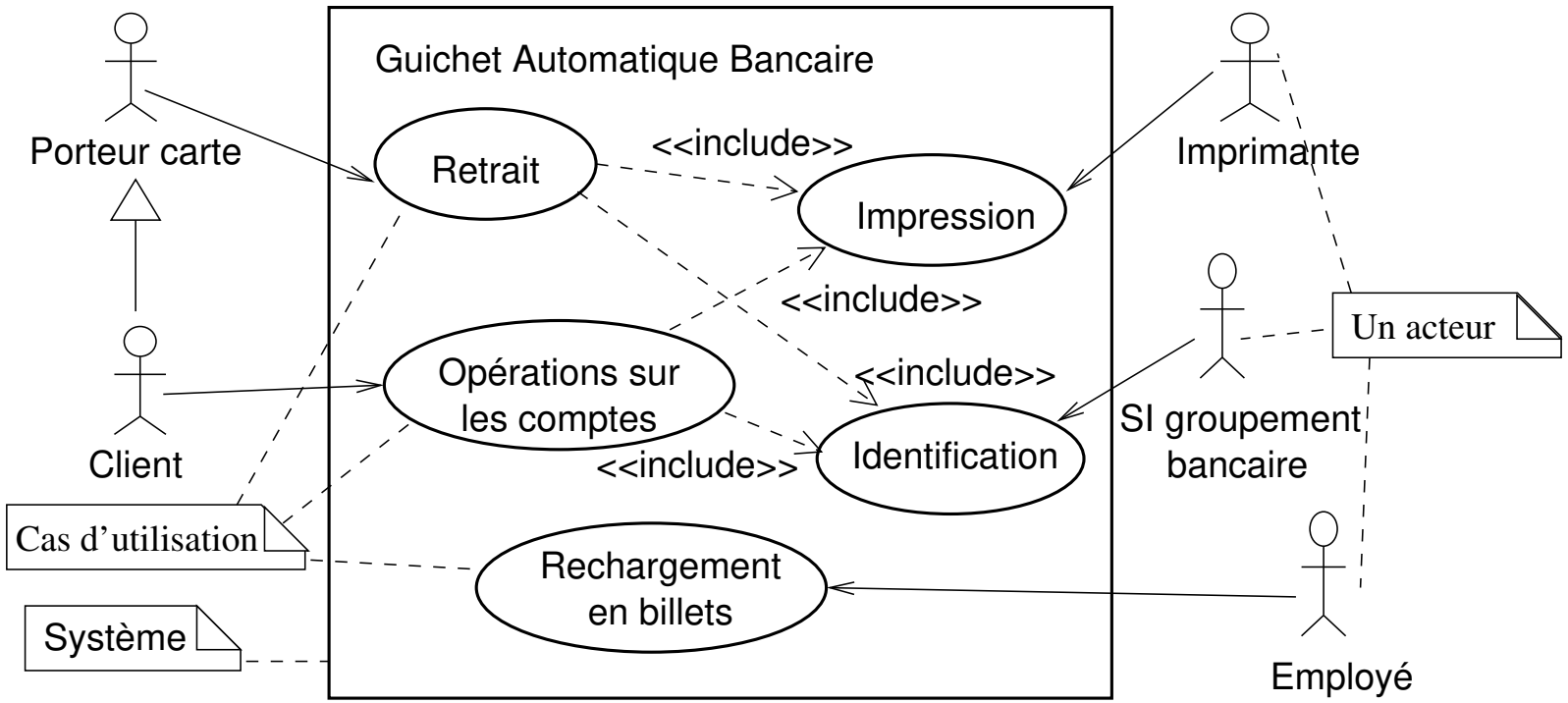
Les étapes suivantes sont issues de OMT [8] :

1. Lister les classes candidates.
2. Sélectionner les classes.
3. Préparer le dictionnaire des données.
4. Lister les relations entre classes.
5. Sélectionner les relations.
6. Construire le modèle objet.
7. Identifier les attributs.
8. Identifier les opérations.
9. Identifier les relations de généralisation/spécialisation.
10. Vérifier les chemins de données.

Problème : Comment trouver les classes candidates ? Et les relations ?

Diagramme de cas d'utilisation

Objectif : Modéliser les *fonctions* du système (cas d'utilisation) telles qu'elles apparaissent aux utilisateurs externes sans en révéler la structure.



Remarque : Les cas d'utilisations (use case) ont été proposés par Jacobson.

Les acteurs

Définition : Un acteur est un rôle joué par une personne ou une chose qui interagit avec le système.
Une personne physique peut donc correspondre à plusieurs acteurs (rôles).
On distingue différents types d'acteurs (exemple du GAB) :

- les *acteurs principaux* utilisent les fonctions principales (les clients) ;
- les *acteurs secondaires* effectuent des tâches administratives ou de maintenance (personne qui recharge la caisse) ;
- le *matériel externe* : dispositifs matériels qui font partie du domaine de l'application (l'imprimante) ;
- les *autres systèmes* : avec lesquels le système interagit (système du groupement bancaire).

Remarque : Possibilité de généralisation entre acteurs.

Définition d'un cas d'utilisation : exemple sur le GAB

Titre : Retrait d'espèces
But : un client réalise un retrait d'espèce sur le compte associé à sa carte.
Acteurs : porteur carte (principal), SI GB (secondaire), imprimante (mat.)
Début : Insertion d'une carte dans un GAB en état de fonctionnement.
Enchaînements : Une fois la carte insérée, le client entre son code, puis le montant du retrait. Après identification correcte de la carte et autorisation de l'opération par le SI du groupement bancaire, le ticket et la carte sont restitués. Une fois la carte récupérée, les billets sont distribués.
Fin : La carte et l'argent ont été récupérés.
Alternatif : Le client peut demander à ne pas avoir de ticket.
En cas de code erroné, le code est redemandé au client.
Exceptions : Le retrait n'est effectif que si le code est correct.
Le 3^{e}code erroné provoque la capture de la carte par le GAB.
Le client peut annuler le retrait.

Éléments d'un cas d'utilisation

Un cas d'utilisation est souvent décrit par des flots d'événements

- flot d'événements principal (exécution nominale, environ 80 % du cas) ;
- zéro ou plusieurs flots d'événements alternatifs ;
- zéro ou plusieurs flots d'exception (cas terminé incorrectement).

La description (textuelle) d'un cas d'utilisation comprend :

- le début du cas d'utilisation : événement déclenchant et condition ;
- la fin du cas d'utilisation : événement qui en cause l'arrêt et condition ;
- les échanges d'informations entre le système et les acteurs (paramètre des interactions) ;
- la chronologie et l'origine des informations ;
- les répétitions de comportement (en pseudo-code) ;
- les situations optionnelles (présentées de manière explicite).

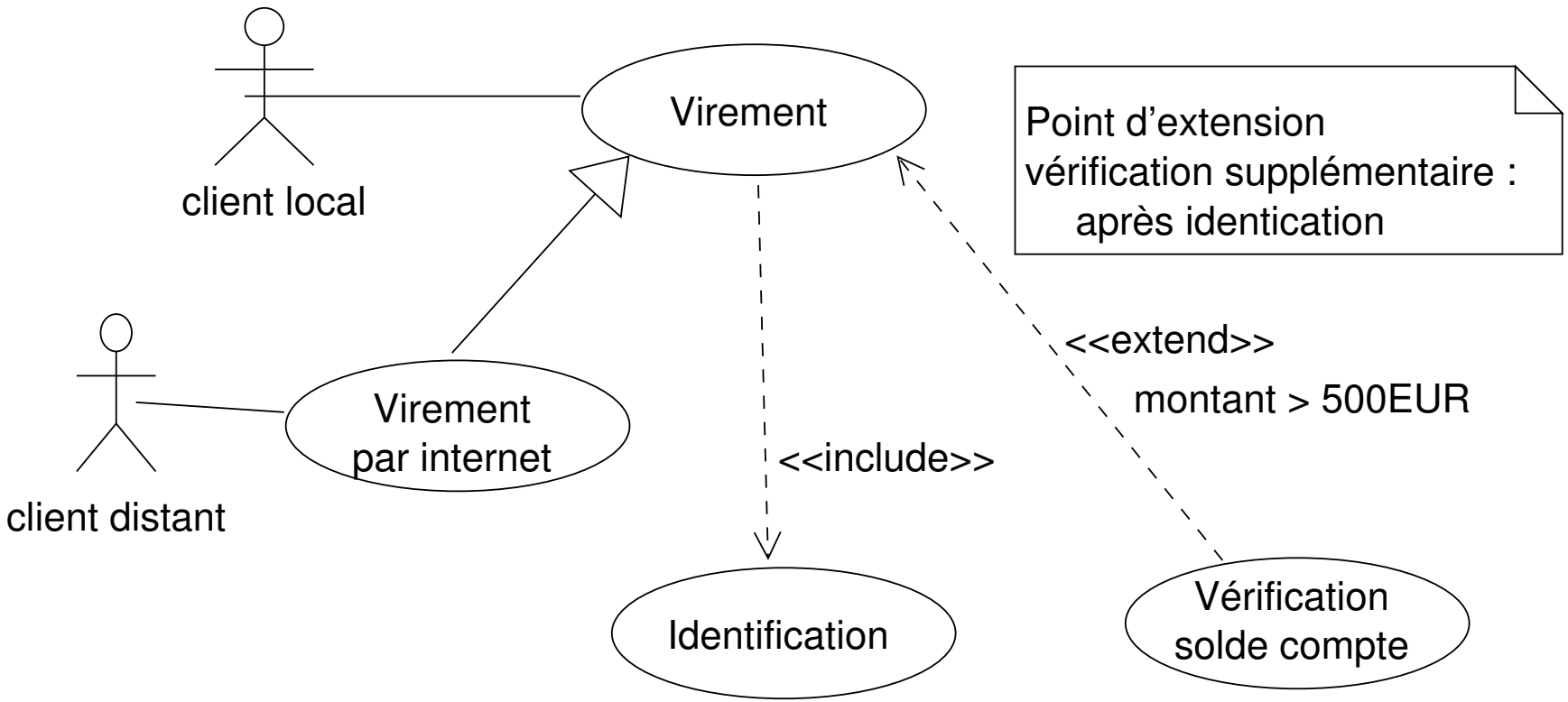
difficulté : Trouver le bon niveau de détail.

Relations entre cas d'utilisation

- *relation de généralisation* : un cas d'utilisation est une spécialisation d'un autre (qui peut être abstrait) ;
- *relation d'inclusion* «`include`» : le cas d'utilisation source comprend également le cas d'utilisation destination (obligatoire) ;
- *relation d'extension* «`extend`» : le cas d'extension source ajoute son comportement au cas d'utilisation destination (point d'extension). L'extension peut être soumise à condition.

Attention : Le vocabulaire est trompeur par rapport à Java : il ne faut pas confondre « extension » et « généralisation ».
Attention : La relation d'inclusion permet une décomposition fonctionnelle. Il ne faut pas en abuser et rester au niveau de la description des fonctionnalités attendues du système.

Exemples de relations entre cas d'utilisation



Intérêts des cas d'utilisation

Les cas d'utilisation permettent de

- capturer les besoins ;
- délimiter la frontière du système ;
- définir les relations entre le système et l'environnement ;
- permettre de dialoguer avec les clients (description textuelle)

Quelques remarques, conséquences et conseils :

- Un cas d'utilisation est une manière spécifique d'utiliser le système (fonctionnalité déclenchée par un utilisateur externe).
- La description textuelle (structurée) des cas d'utilsation peut être complétée par du pseudo-code, des diagrammes d'activité, etc.
- Il faut rester au niveau du problème (les besoins) et non d'une solution.
- Un cas d'utilisation doit faire apparaître les actions réalisées par les acteurs dans le cadre de leur métier (pas de manipulation d'IHM !).

Les scénarios

Définition : Un scénario est une instance d'un cas d'utilisation. Il décrit un exemple d'interaction possible entre le système et les acteurs.
But : Valider un cas d'utilisation (et trouver les cas d'utilisation !).
Principe : Définir plusieurs scénarios pour un cas d'utilisation :

- un scénario représentant l'exécution nominale ;
- des scénarios nominaux moins fréquents ;
- des scénarios d'exception (qui ne permettent pas de terminer correctement le cas d'utilisation).

Remarque : Un cas d'utilisation est le regroupement de plusieurs scénarios suivant un critère fontionnel.
Formalisation : Un scénario est formalisé par un diagramme d'interaction (diagramme de séquence ou diagramme de communication).

Diagramme de communication

Objectif : Décrire les *interactions* entre objets pour un scénario d'un cas d'utilisation en privilégiant la *structure spatiale* des objets.
Un diagramme de communication est une réalisation d'un cas d'utilisation.

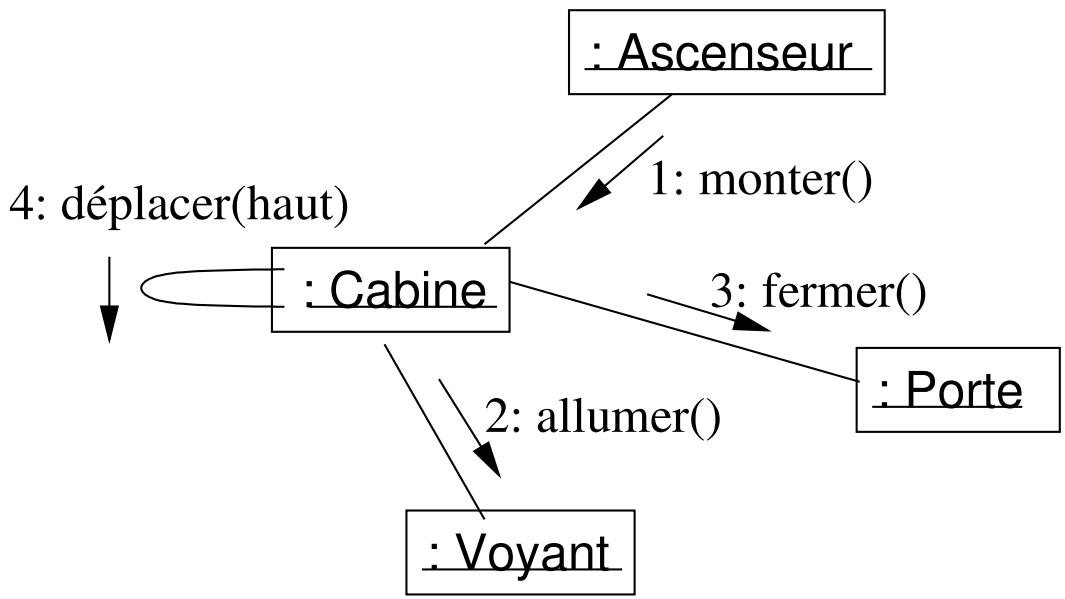
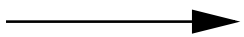
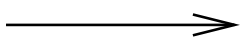


Diagramme d'objet = diagramme de communication sans envoi de messages

Envoi de message

Message : information échangée (passage de paramètre ou valeur de retour).
Un **envoi de message** met en jeu un objet émetteur et un objet récepteur.
Il est symbolisé par une flèche sur un lien entre les deux objets.
Différents types d'envoi de message :

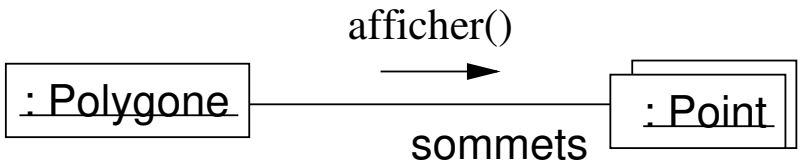
-  : *Flot de contrôle synchrone* (appel de procédure)
-  : *Flot de contrôle asynchrone*

Remarque : En utilisant le mécanisme d'extension, il est possible de définir des envois de message minutés.
Syntaxe d'un message :

```
séquence ':' [result ':= '] nom_message(arguments)
```

Complément sur la représentation des objets

- O / R : C : un objet O instance de la classe C qui a le rôle R.
- Un double rectangle représente un groupe d'objets :



- : Ordinateur [calcul] : un objet anonyme de la classe Ordinateur dans l'état « calcul ».
- stéréotypes «copy» (objet obtenu par copie avec évolution indépendante de la source) et «become» (changement de machine à état d'un objet).

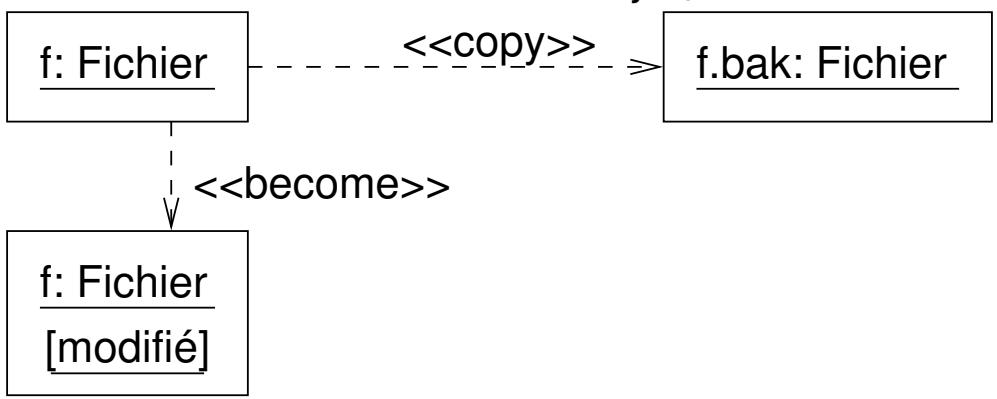
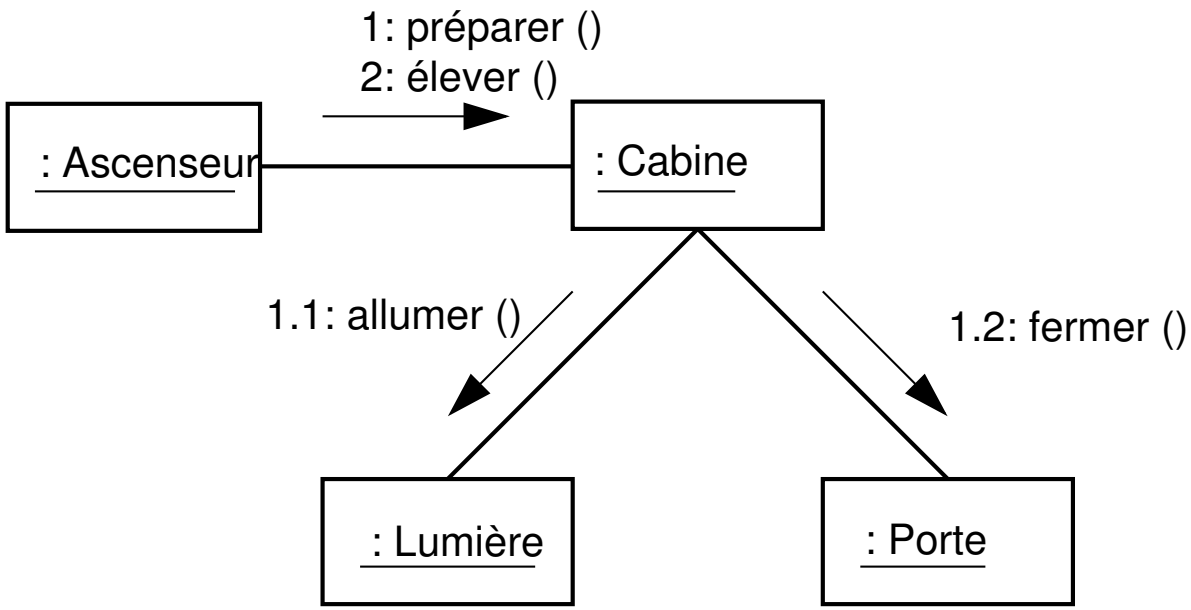


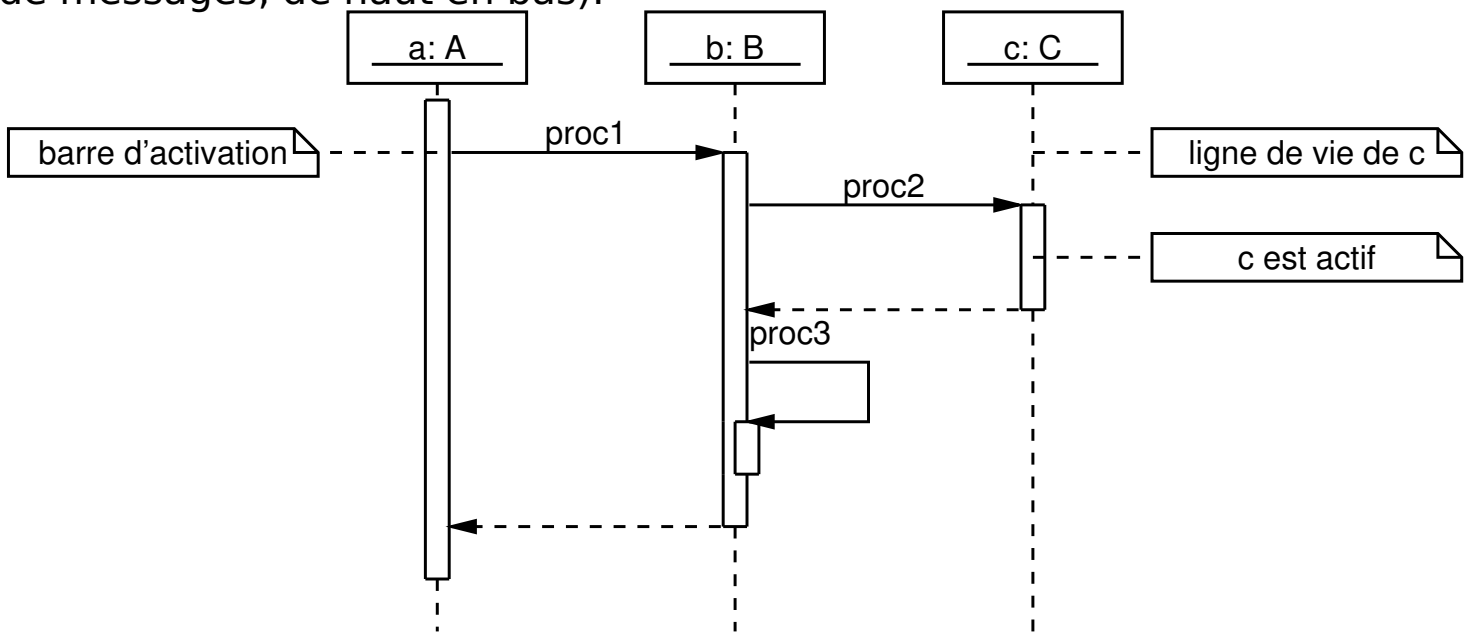
Diagramme de communication : séquence



Les séquences peuvent être emboitées.

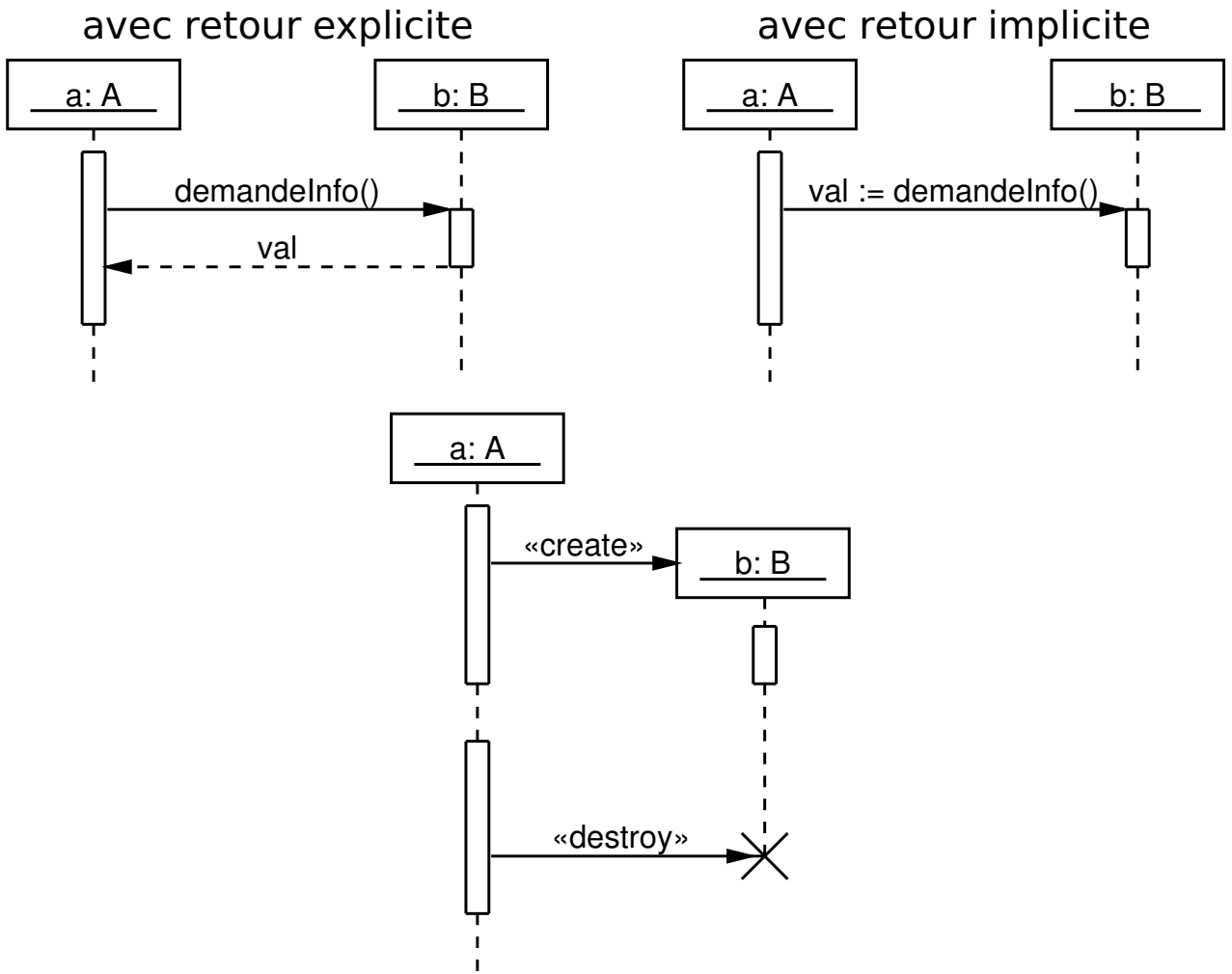
Diagramme de séquence

Objectif : Décrire les interactions entre objets en privilégiant l'aspect temporel (chronologie des envois de messages, de haut en bas).

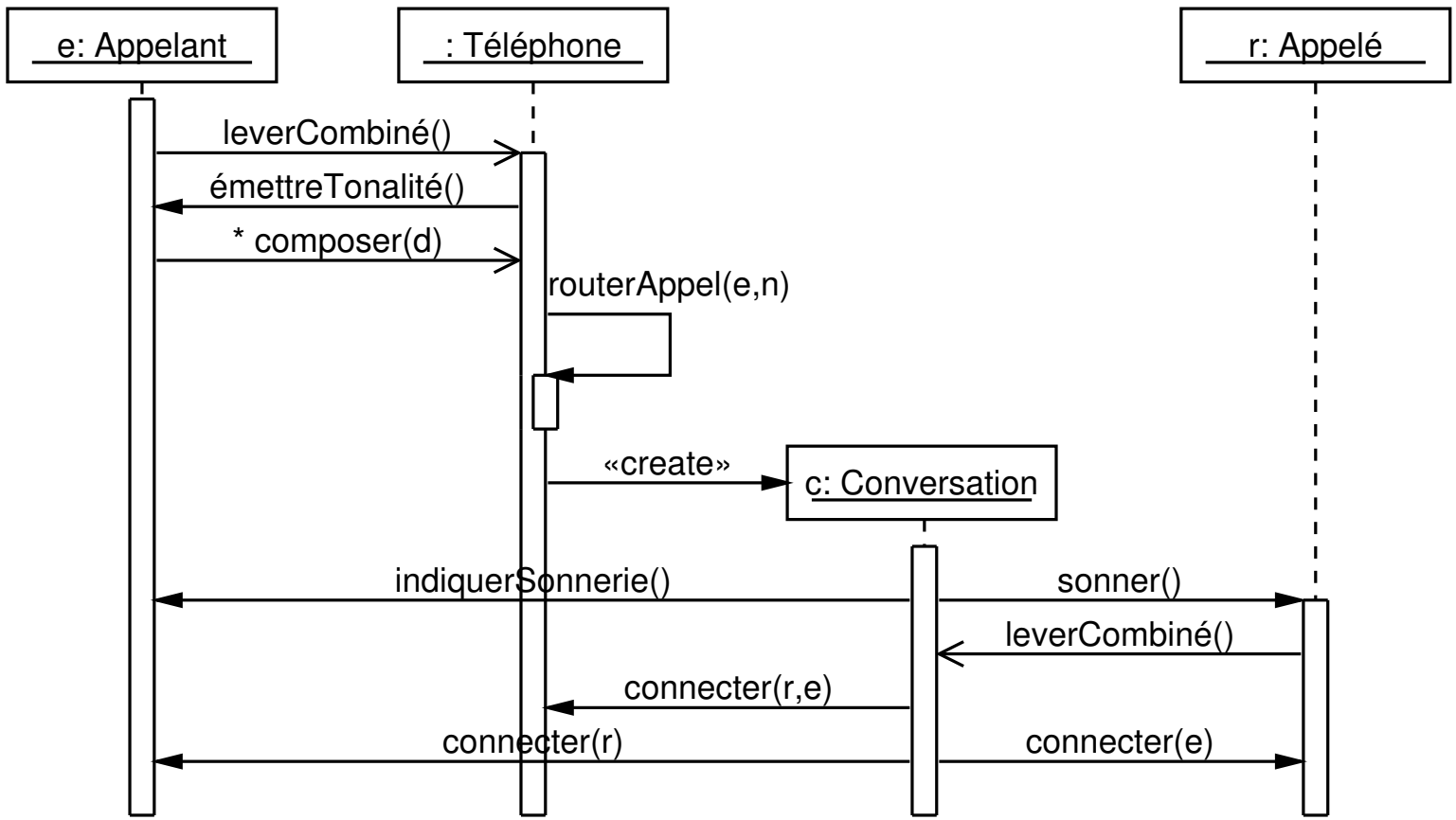


Barre d'activation : période d'activité d'un objet

Retour de méthodes synchrones



Exemple plus complet avec messages asynchrones

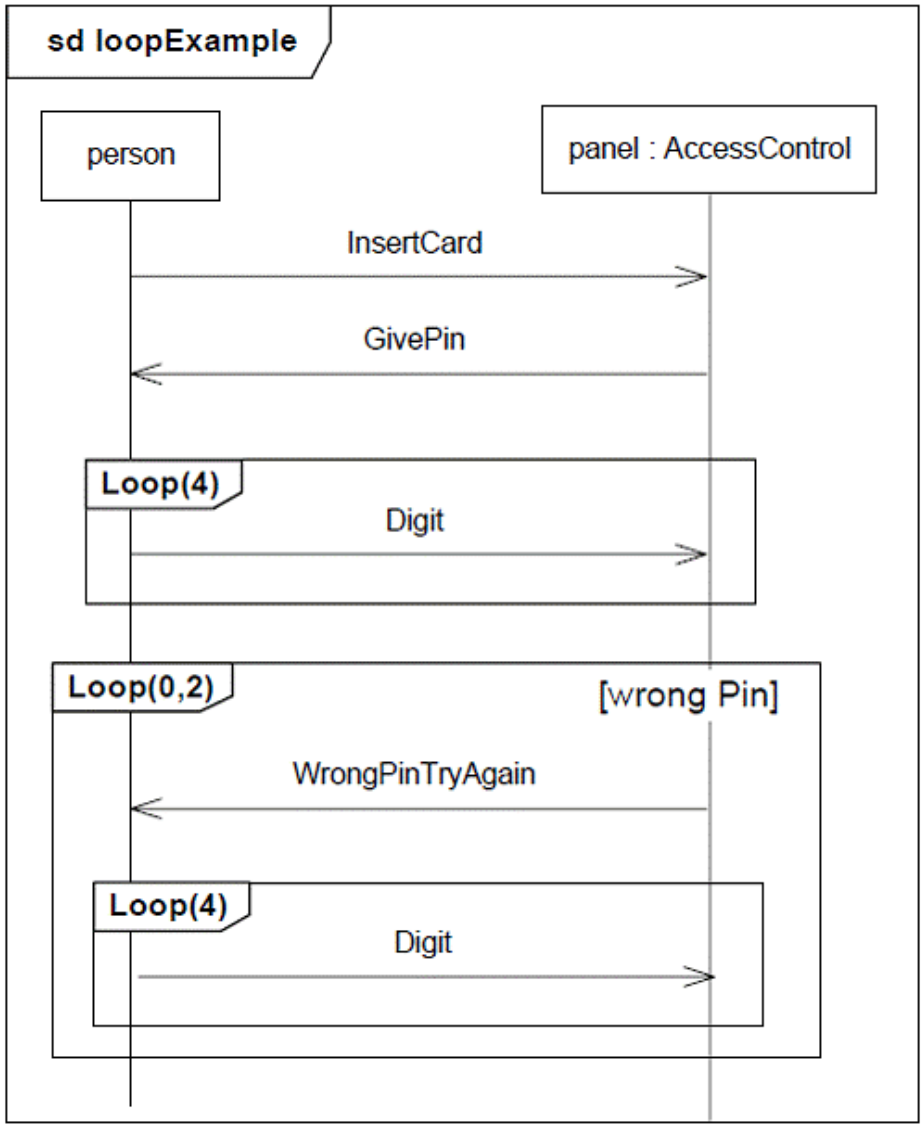


Cadre d'interactions

Introduits en UML 2, les cadres d'interaction permettent de capturer sur un même diagramme de séquence plusieurs scénarios.

On se rapproche d'un diagramme d'activité (car tend à décrire un comportement de manière exhaustive).

Opérateurs possibles : alt, opt, par, loop, negative, sd, etc.



Des raffinages aux diagrammes de séquence

```
Comment « Arbitrer un jeu entre un devin et un maître » ?
    Demander au maître de choisir un nombre
    Indiquer au devin les limites du jeu
    Répéter
        Demander au devin de faire une proposition
        Demander au maitre une indication
        Donner au devin l'indication
    JusquÀ nombre trouvé
    Féliciter le devin
```

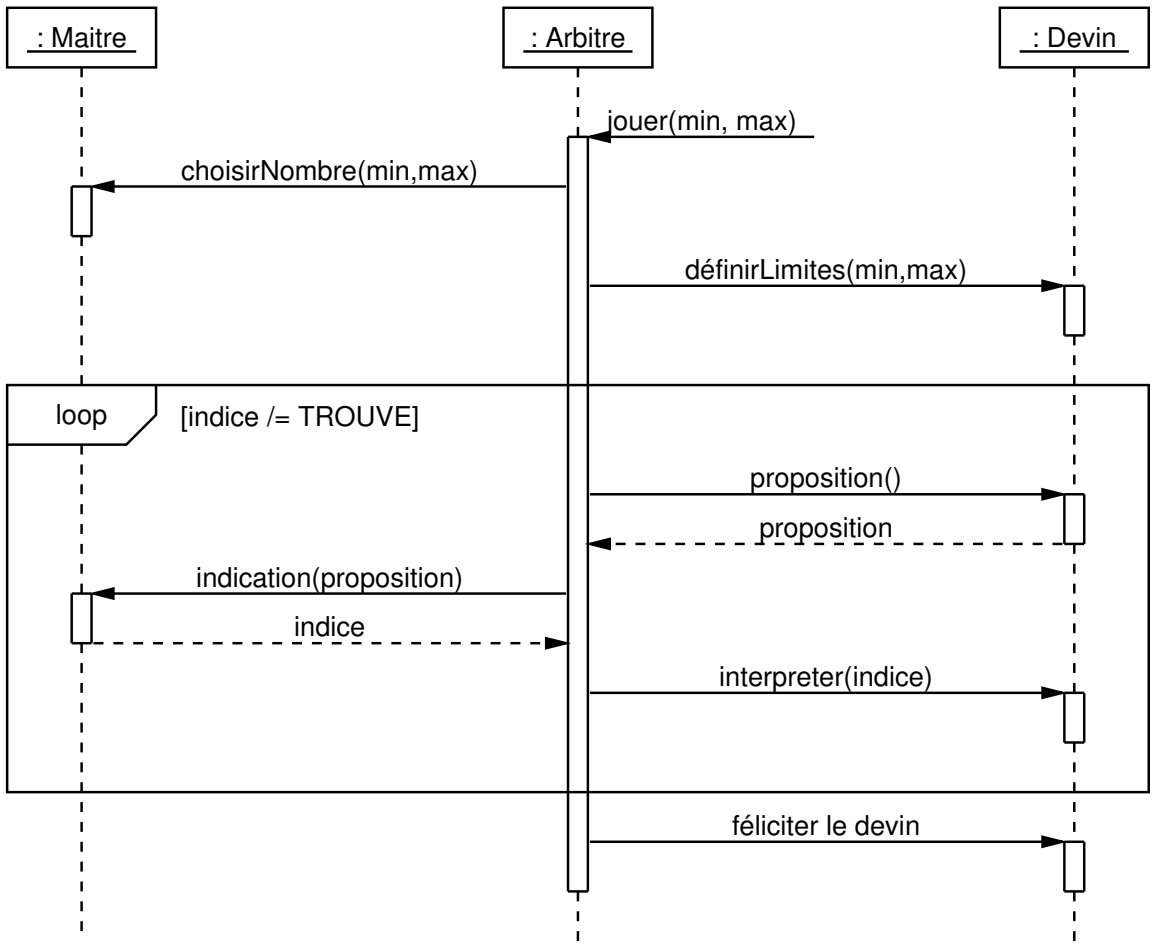
- Les participants sont explicités : devin, maître... et arbitre
- La règle « une étape est un verbe à l'infinitif » est souvent remplacée par « une étape est de la forme : sujet verbe complément »

```
L'arbitre demande au maître de choisir un nombre
L'arbitre indique au devin les limites du jeu
...
```

Comment représenter les structures de contrôle sur un diagramme de séquence ?

Les cadres d'interaction (frame), depuis UML 2

Vers l'expression d'un algorithme



Les cadres d'interaction (frame), depuis UML 2 (2)

Vers l'expression d'un algorithme

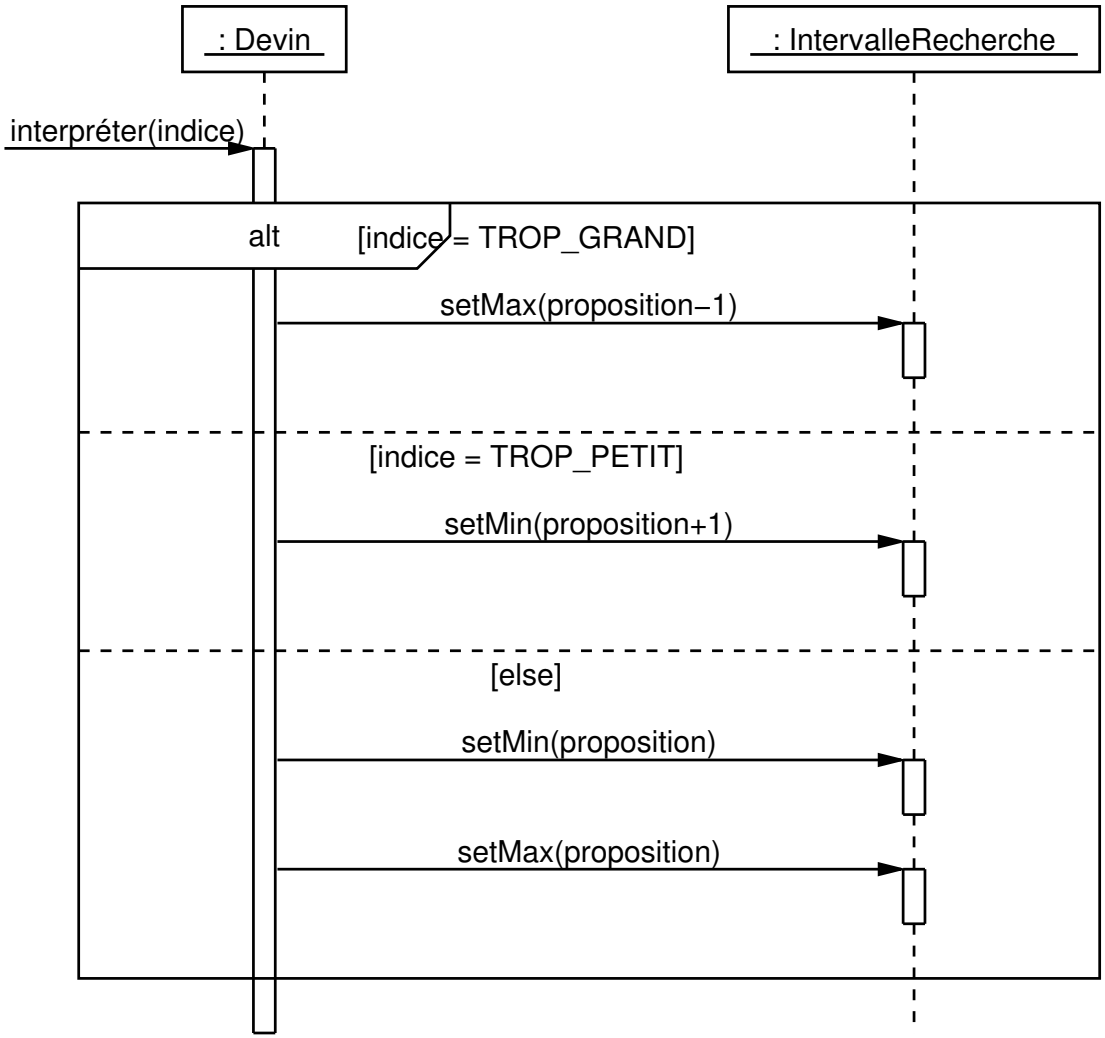


Diagramme de classe correspondant

Du diagramme de séquence, on peut déduire des éléments du diagramme de classe :

- les classes : type des objets du diagramme de séquence
- les méthodes : message sur une interaction (dans la classe cible)
- les dépendances entre classes (mais pas le type de relation)

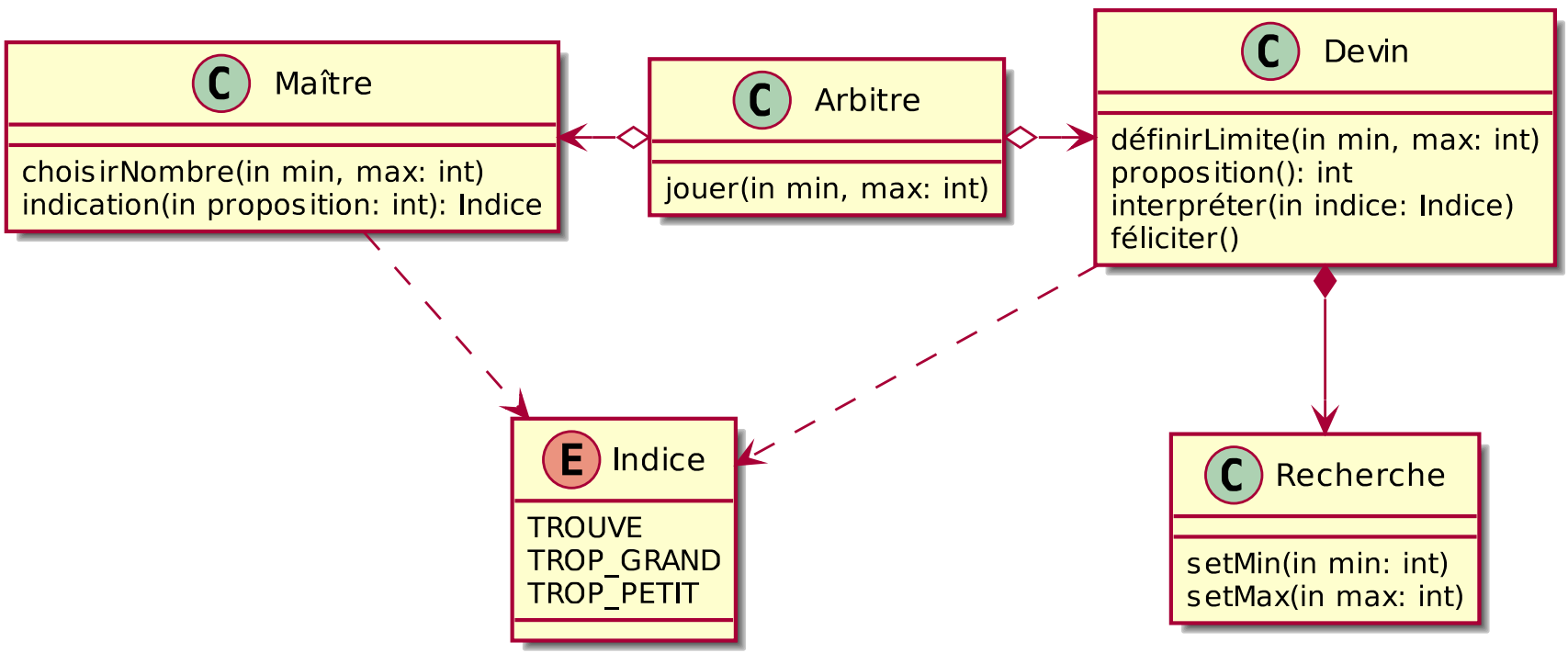


Diagramme de machine à états

Objectifs : décrire le comportement d'une classe (ou d'un sous-système), en particulier :

- les différents états possibles de ses objets,
- les transitions possibles entre ces états et les événements déclenchant
- les séquencements possibles de ses opérations.

Les constituants du diagramme de machine à états sont :

- les événements (stimuli externes ou internes) ;
- les états (valeurs des objets) ;
- les changements d'états (transitions) ;
- les opérations (actions ou activités).

Les constituants du modèle dynamique

Événements : un *événement* est quelque chose qui :

- est instantané ;
- se produit à un moment (occurrence) ;
- transmet de l'information d'un objet à un autre (attribut).

Exemple : « Mettre une pièce », « Choisir une boisson »

Scénario : séquence d'événements correspondant à une exécution potentielle.
Exemple : Scénario pour un distributeur de boissons

1. mettre 20 centimes
2. mettre 10 centimes
3. choisir du café
4. prendre le café

Les constituants du modèle dynamique (suite)

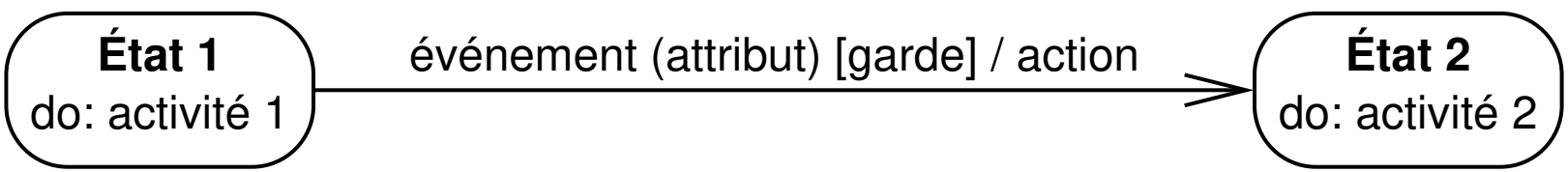
État : abstraction des valeurs attributs et des liens d'un objet
Un état définit la réponse d'un objet à un événement.
Exemple : un automate est dans l'état « inactif » si aucune pièce n'a été insérée, « récupération » quand de l'argent est inséré, etc.
Opération : on distingue deux types d'opérations :

- *action :* opération instantanée associée à un événement ;
- *activité :* opération qui dure, associée à un état.

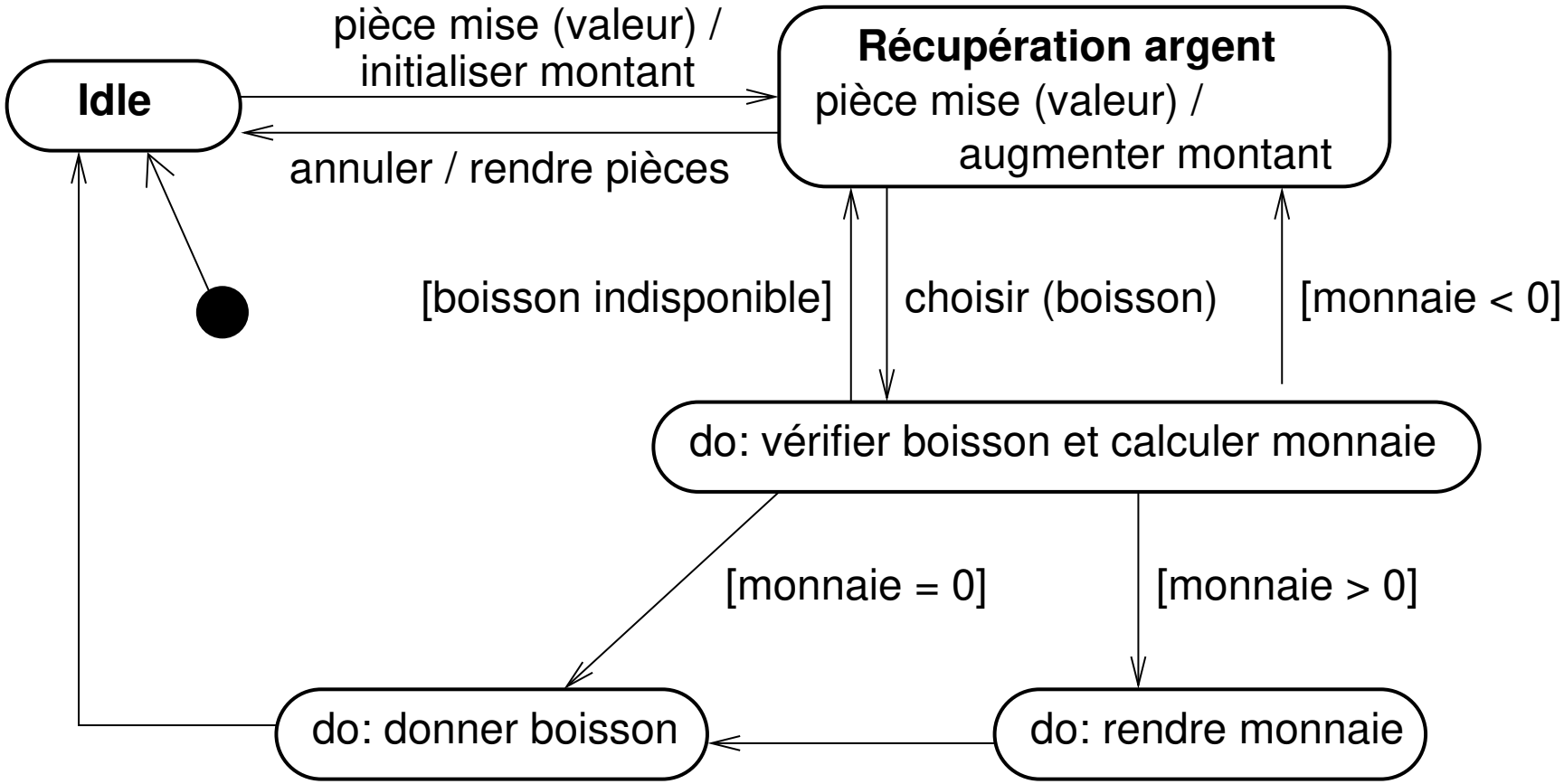
Exemple : « comptabiliser la pièce mise » est une action. « Donner la boisson » est une activité.
Attention : En UML 2, une action est élémentaire et non interruptive et une activité un ensemble d'actions.

Les constituants du modèle dynamique (fin)

Diagramme de machine à états : diagramme qui relie les états au moyen de transitions qui sont franchies sur l'occurrence d'événements.
Condition : expression booléenne portant sur les informations transportées par l'événement et sur les valeurs des attributs ou liens de l'objet.
Une condition peut être utilisée pour garder une transition.
Schéma général :

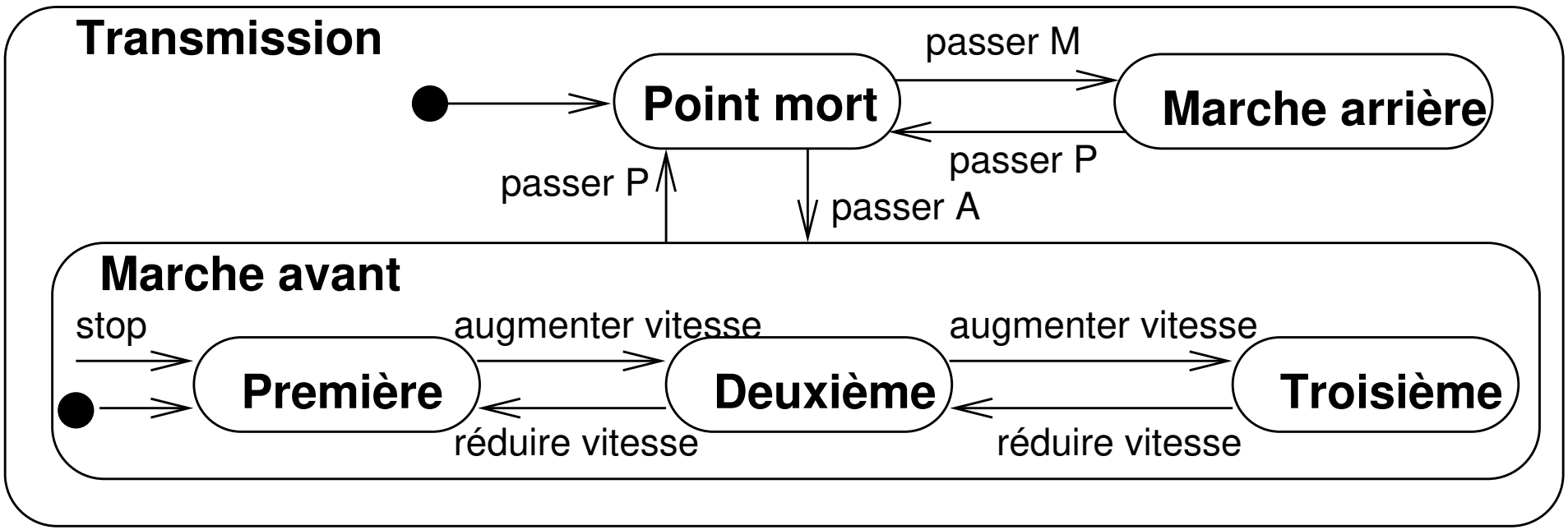


Exemple de diagramme de machine à états



Les états composites (structuration « OU »)

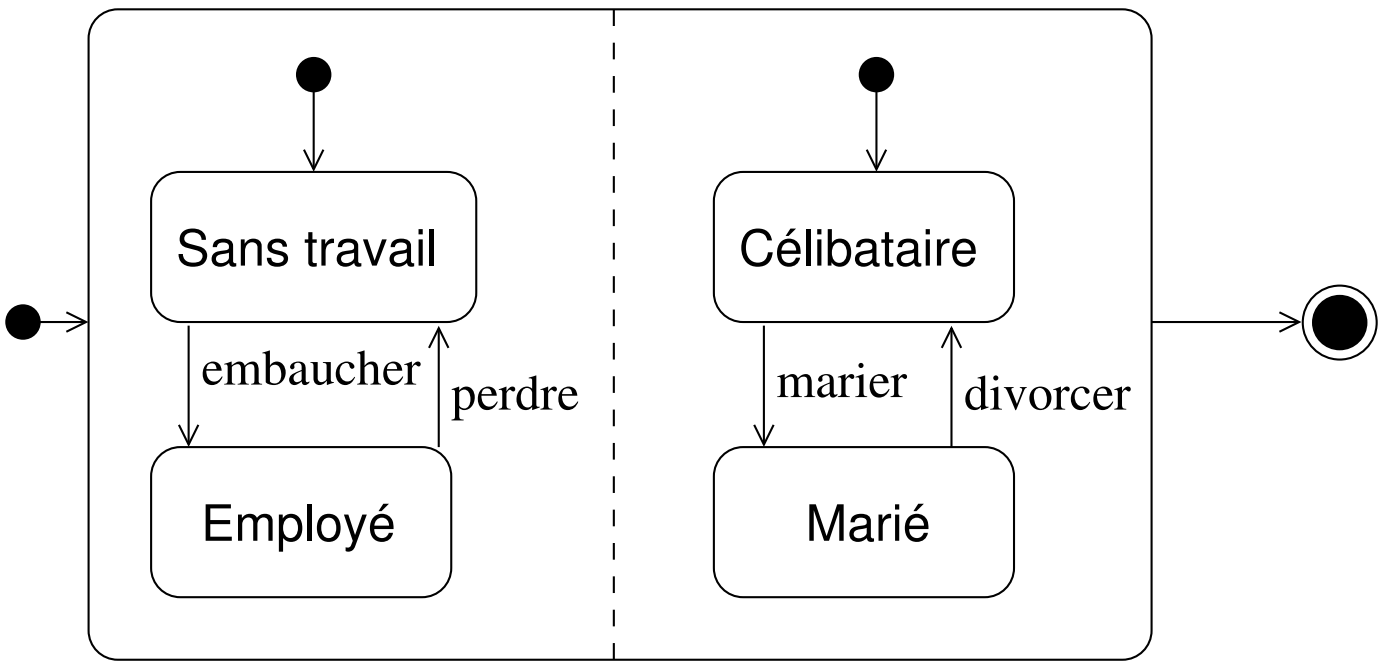
Définition : Un état composite (ou englobant) correspond à la décomposition d'un état en sous-états (régis par un diagramme de machine à états).
On parle également de généralisation d'un état (pour l'état englobant).



Remarque : Un seul sous-état est actif (OU).

Les états concurrents (structuration « ET »)

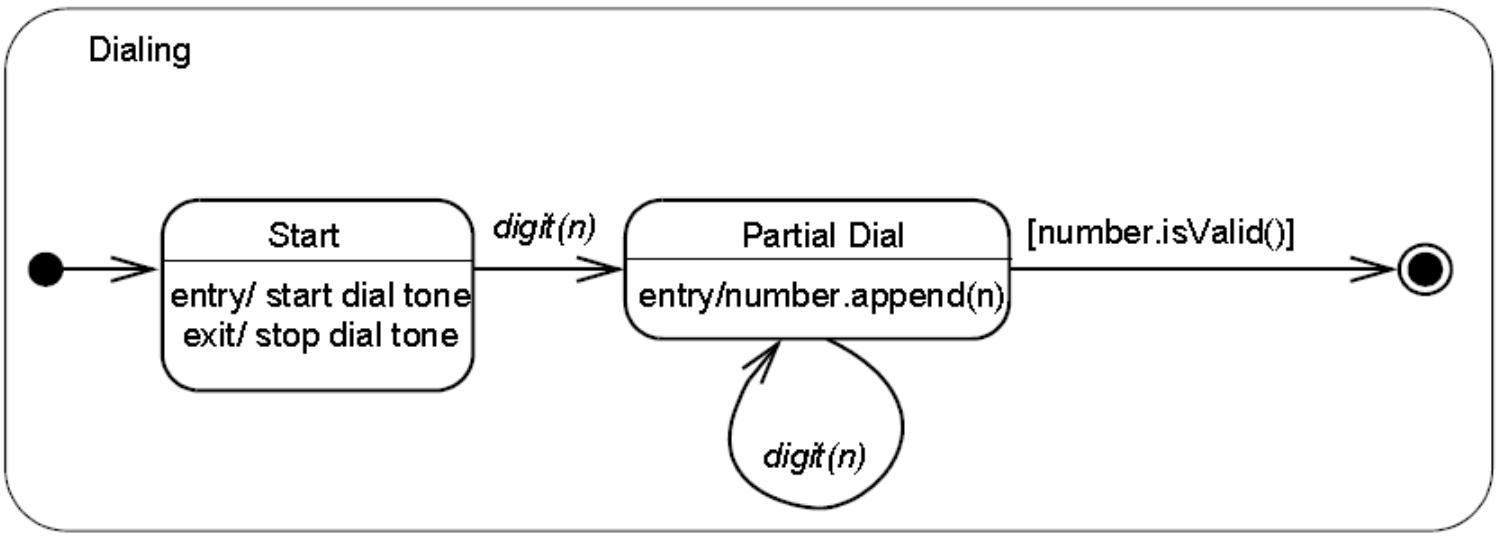
Définition : Un état peut contenir plusieurs sous-états concurrents appelés *régions*. L'objet est simultanément dans tous les sous-états (ET).
Chaque région est décrite par un diagramme de machine à états.



Autres concepts

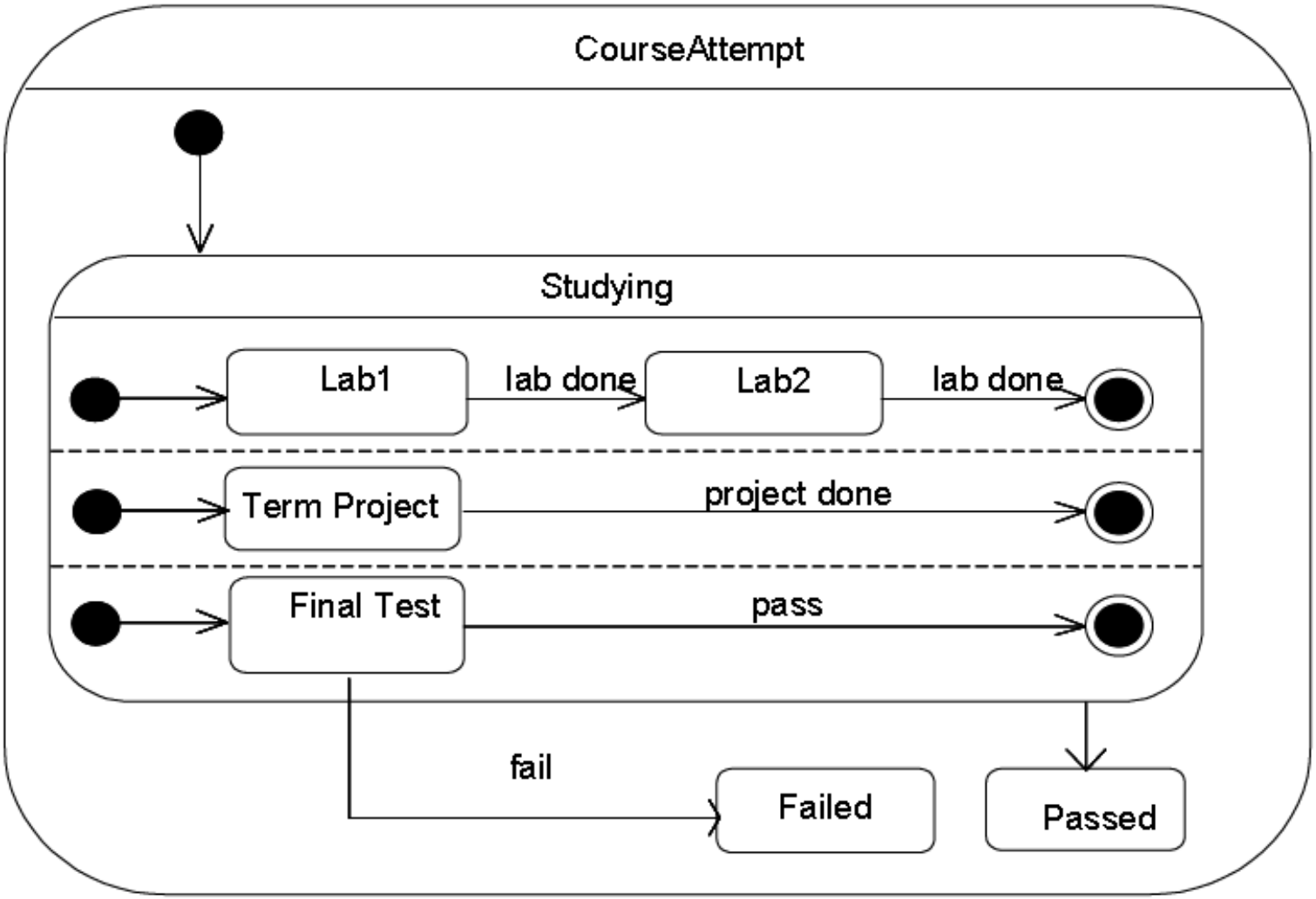
Transitions automatiques : on peut définir une transition de sortie d'un état avec activité sans préciser d'événement.
Dès que l'activité est terminée, la transition est tirée.
Actions associées à un état :

- actions d'entrée : exécution (instantanée) de l'action lors de l'entrée dans l'état.
- actions de sortie : exécution de l'action au moment de quitter l'état.
- action interne : exécution d'une action sur événement sans changer d'état.



Envoi d'événement : un objet peut exécuter l'envoi d'un événement vers un autre objet.

Participation à un cours



Téléphone

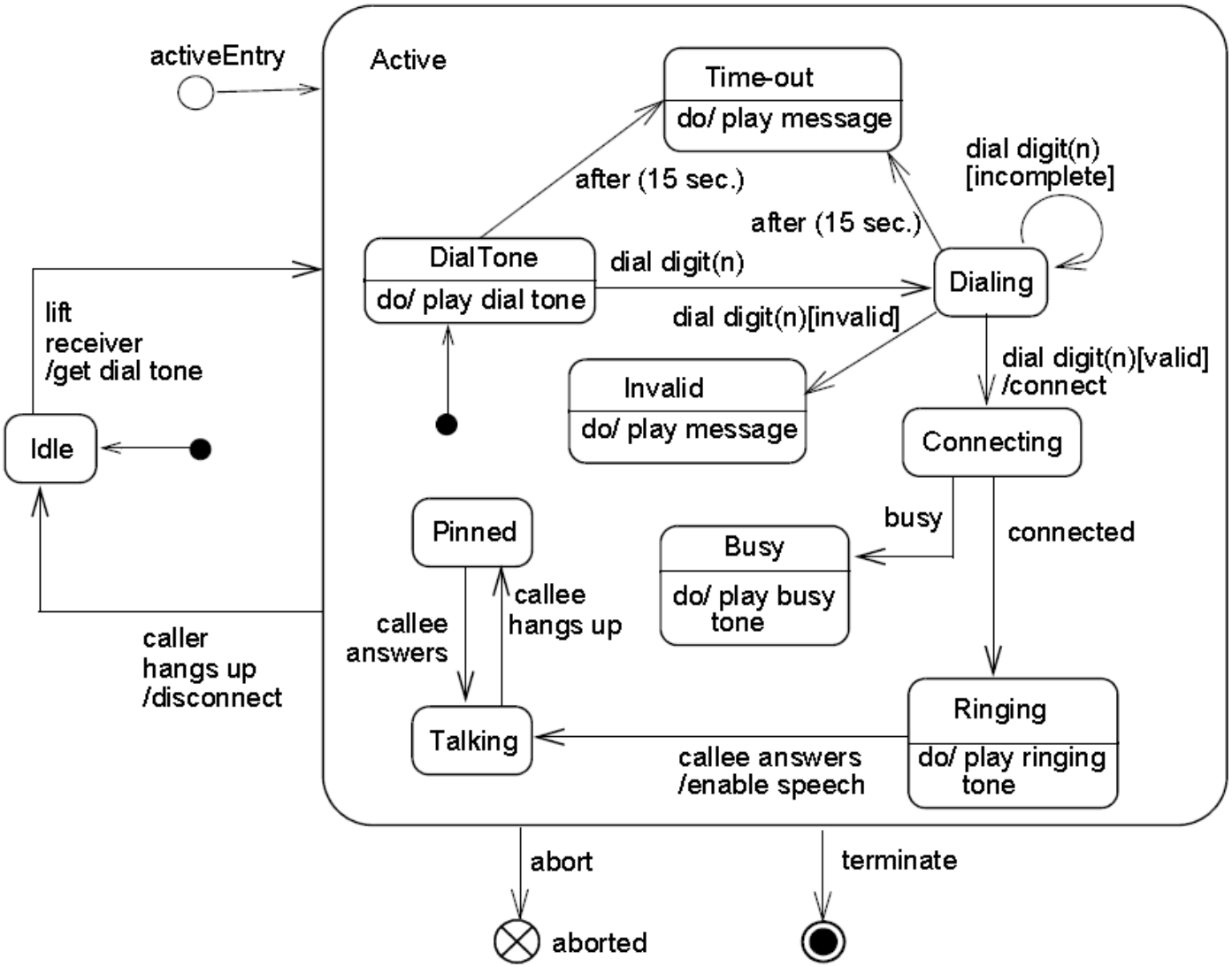
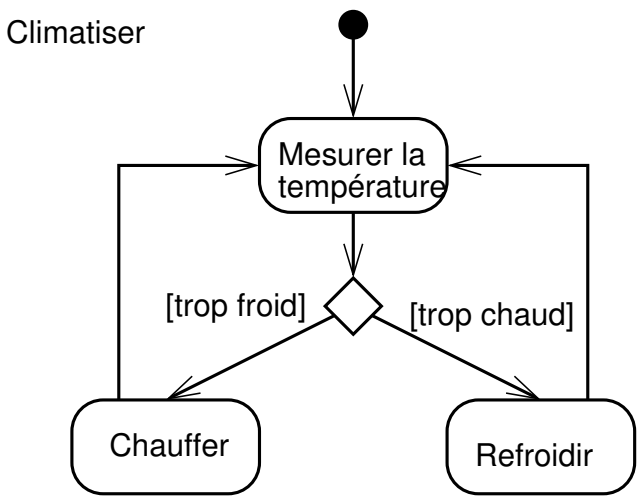
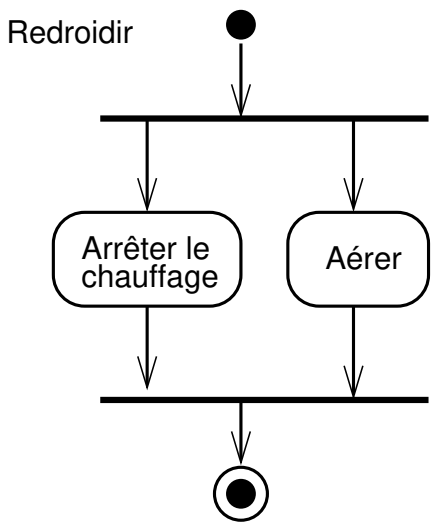

activeEntry
Active
Time-out
do/ play message
after (15 sec.)
dial digit(n)
[incomplete]
after (15 sec.)
DialTone
do/ play dial tone
dial digit(n)
Dialing
dial digit(n)[invalid]
lift
receiver
/get dial tone
Invalid
do/ play message
dial digit(n)[valid]
/connect
Idle
Connecting
Pinned
busy
connected
Busy
do/ play busy
tone
callee
hangs up
callee
answers
caller
hangs up
/disconnect
Ringing
do/ play ringing
tone
Talking
callee answers
/enable speech
abort
terminate
aborted

Diagramme d'activité

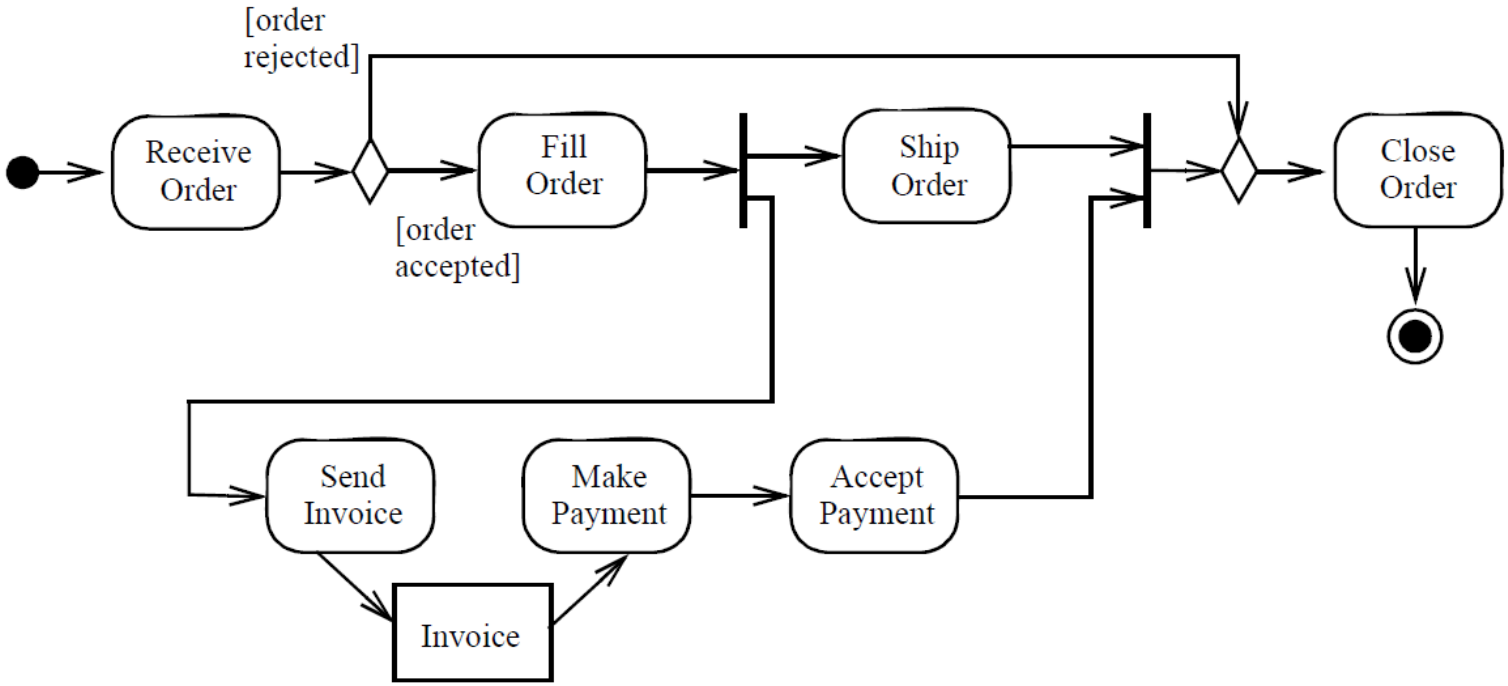
Objectif : Décrire les activités (séquentielles ou parallèles) d'un calcul, d'un processus, d'un workflow, etc.

- activité qui peuvent être décomposées
- noeud de décision (losange)
- point de débranchement (fork) et de jonction (join) pour activités parallèles
- garde sur les transitions

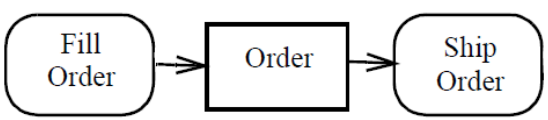


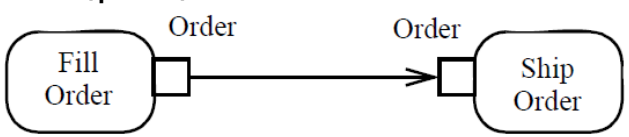


Flots de contrôle et flots de données : Traitement d'une commande



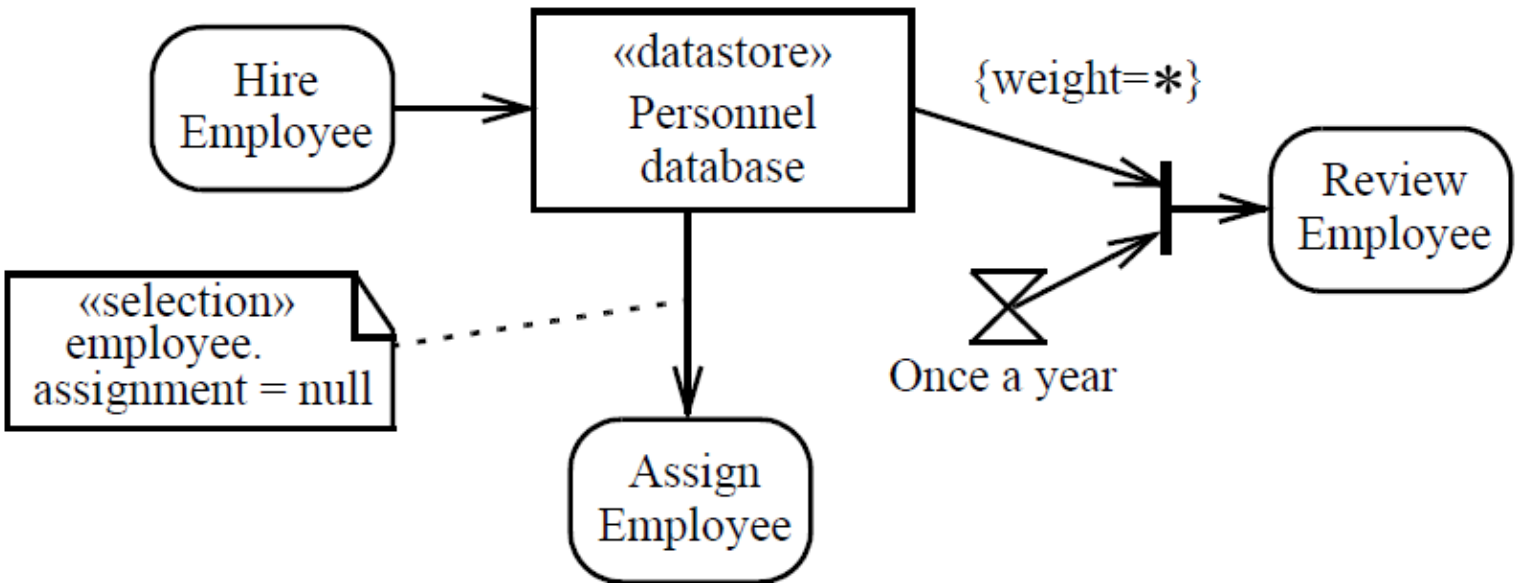
- Flot de données (data flow) : on explicite les données échangées (Invoice)
  Souvent, on utilise des connecteurs (pins)





- Flot de contrôle (control flow) : l'activité suivante commence quand la précédente est terminée

Événements temporels



- Sablier pour événements temporels
- Le poids d'un arc indique combien d'éléments doivent être présents pour le franchir (sémantique des réseaux de Petri)

Bilan des diagrammes dynamiques

Objectif : Décrire le comportement du système
Moyens :

- *Diagramme de cas d'utilisation :* description des fonctions du système du point de vue des utilisateurs externes ;
- *Diagrammes d'interaction :* décrire une séquence de messages partiellement ordonnés échangés entre objets en privilégiant :
  - l'organisation spatiale des objets : *diagramme de communication*
  - la chronologie des envois de messages : *diagramme de séquence*
- Diagrammes permettant une description exhaustive d'un comportement :
  - *Diagramme de machine à états :* comportement d'une classe ;
  - *Diagramme d'activité :* description des activités d'un calcul.