

UML 2.0

**Département Informatique
IUT de Villetaneuse**



Plan

- Introduction à la modélisation objet (*1 séance*)
- Diagrammes UML
 - Diagramme de cas d'utilisation (*1 séance*)
 - Diagramme de classes et d'objets (*2 séances*)
 - Diagrammes d'interaction (*2 séances*)
 - Diagramme d'états-transitions (*1 séance*)
 - Diagramme d'activités (*1 séance*)
 - Diagrammes de composants et de déploiement (*1 séance*)
- Object constraint language (*2 séances*)
- Design patterns (*2 séances*)



Bibliographie UML

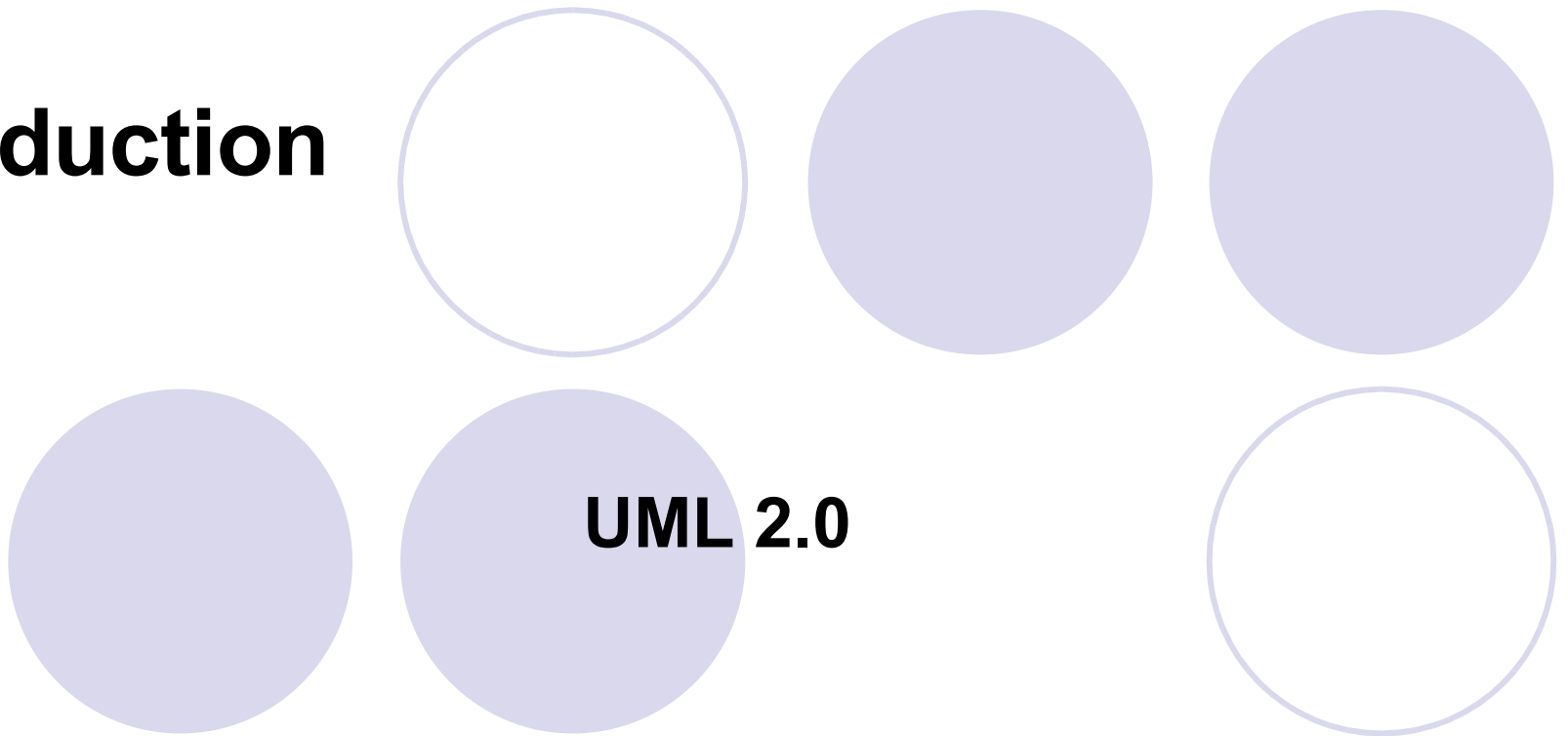
- « **UML 2.0, guide de référence** »
 - *James Rumbaugh, Ivar Jacobson, Grady Booch*
 - Editions Campus Press (2005)
- « **UML 2.0** »
 - *Benoît Charoux, Aomar Osmani, Yann Thierry-Mieg*
 - Editions Pearson, Education France (2005)
- « **UML 2.0 Superstructure** » et « **UML 2.0 Infrastructure** »
 - *OMG (Object Management Group)*
 - <http://www.uml.org> (2004).

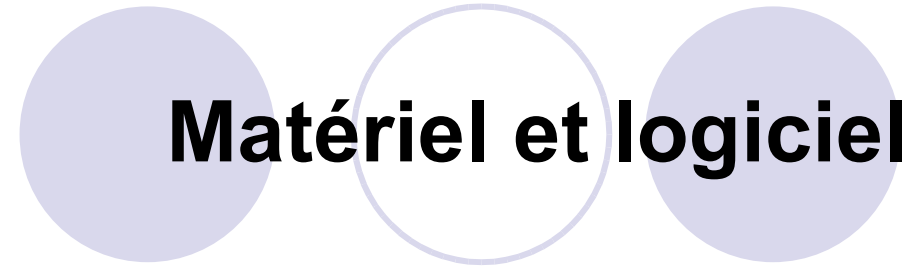
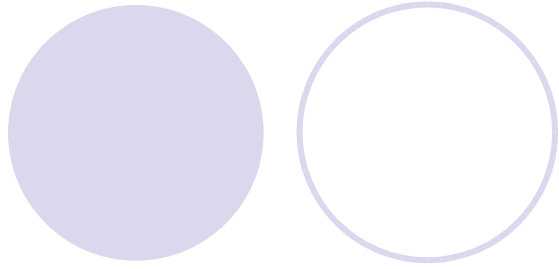


Bibliographie OCL

- « **UML 2.0 Object Constraint Language (OCL)** »
 - *OMG (Object Management Group)*
 - `http://www.uml.org` (2004).
- Cours de Jean-Marie Favre, IMAG
 - `http://www-adele.imag.fr/~jmfavre`

Introduction





Matériel et logiciel

- Systèmes informatiques :
 - 80 % de *logiciel* ;
 - 20 % de matériel.
- Depuis quelques années, la fabrication du matériel est assurée par quelques fabricants seulement.
 - Le matériel est relativement fiable.
 - Le marché est standardisé.
- Les problèmes liés à l'informatique sont essentiellement des problèmes de logiciel.



Spécificité du logiciel

- Un produit immatériel, dont l'existence est indépendante du support physique :
 - Semblable à une *oeuvre d'art* (roman, partition...).
- Un objet technique fortement contraint qui fonctionne ou ne fonctionne pas, avec une structure complexe :
 - Relève des modes de travail du *domaine technique*.
- Un cycle de production différent :
 - La reproduction pose peu de problèmes, seule la première copie d'un logiciel a un coût.
 - Semblable au *génie civil* (ponts, routes...).



Un processus de fabrication original

- Le logiciel partage des propriétés contradictoires avec différents types de produit.
 - Les possibilités de réutiliser les savoir-faire des autres domaines sont parfois limitées.
- Compte tenu du cycle de production, il faut bien faire tout de suite.

« La qualité du processus de fabrication est garante de la qualité du produit »



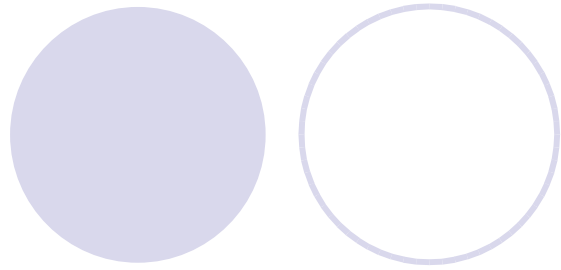
La « crise du logiciel »

- Étude sur 8 380 projets (Standish Group, 1995) :
 - Succès : 16 % ;
 - Problématique : 53 % (budget ou délais non respectés, défaut de fonctionnalités) ;
 - Échec : 31 % (abandonné).
- Le taux de succès décroît avec la taille des projets et la taille des entreprises.
- *Génie Logiciel* (Software Engineering) :
 - Comment faire des logiciels de qualité ?
 - Qu'attend-on d'un logiciel ? Quels sont les critères de qualité ?

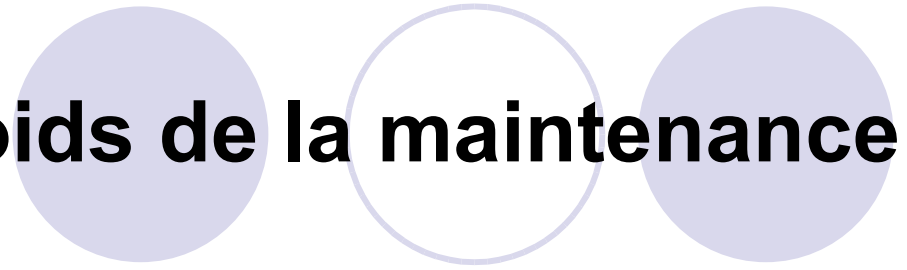


Critères de qualité d'un logiciel

- Utilité :
 - Adéquation entre le logiciel et les besoins des utilisateurs ;
- Utilisabilité ;
- Fiabilité ;
- Interopérabilité :
 - Interactions avec d'autres logiciels ;
- Performance ;
- Portabilité ;
- Réutilisabilité ;
- Facilité de maintenance :
 - Un logiciel ne s'use pas,
 - Pourtant, la maintenance absorbe un très grosse partie des efforts de développement.



Poids de la maintenance



| | Répartition effort dév. | Origine des erreurs | Coût de la maintenance |
|---------------------------|----------------------------|------------------------|---------------------------|
| Définition des besoins | 6% | 56% | 82% |
| Conception | 5% | 27% | 13% |
| Codage | 7% | 7% | 1% |
| Intégration Tests | 15% | 10% | 4% |
| Maintenance | 67% | | |

(Zeltovitz, De Marco)



Principes utilisés dans le Génie Logiciel

- *Généralisation* :
 - Regroupement d'un ensemble de fonctionnalités semblables en une fonctionnalité paramétrable (généricité, héritage).
- *Structuration* :
 - Façon de décomposer un logiciel (utilisation d'une méthode ascendante ou descendante).
- *Abstraction* :
 - Mécanisme qui permet de présenter un contexte en exprimant les éléments pertinents et en omettant ceux qui ne le sont pas.

Principes utilisés dans le Génie Logiciel

- *Modularité :*
 - Décomposition d'un logiciel en composants discrets.
- *Documentation :*
 - Gestion des documents en incluant identification, acquisition, production, stockage et distribution.
- *Vérification :*
 - Détermination du respect des spécifications établies sur la base des besoins identifiés dans la phase précédente du cycle de vie.



*« La qualité du processus de fabrication
est garante de la qualité du produit. »*

- Pour obtenir un logiciel de qualité, il faut en maîtriser le processus d'élaboration.
 - La vie d'un logiciel est composée de différentes étapes.
 - La succession de ces étapes forme le *cycle de vie* du logiciel.
 - Il faut contrôler la succession de ces différentes étapes.



Étapes du développement

- Étude de faisabilité ;
- Spécification :
 - Déterminer les fonctionnalités du logiciel ;
- Conception :
 - Déterminer la façon dont le logiciel fournit les différentes fonctionnalités recherchées ;
- Codage ;
- Tests :
 - Essayer le logiciel sur des données d'exemple pour s'assurer qu'il fonctionne correctement ;
- Maintenance.



Types de cycle de vie

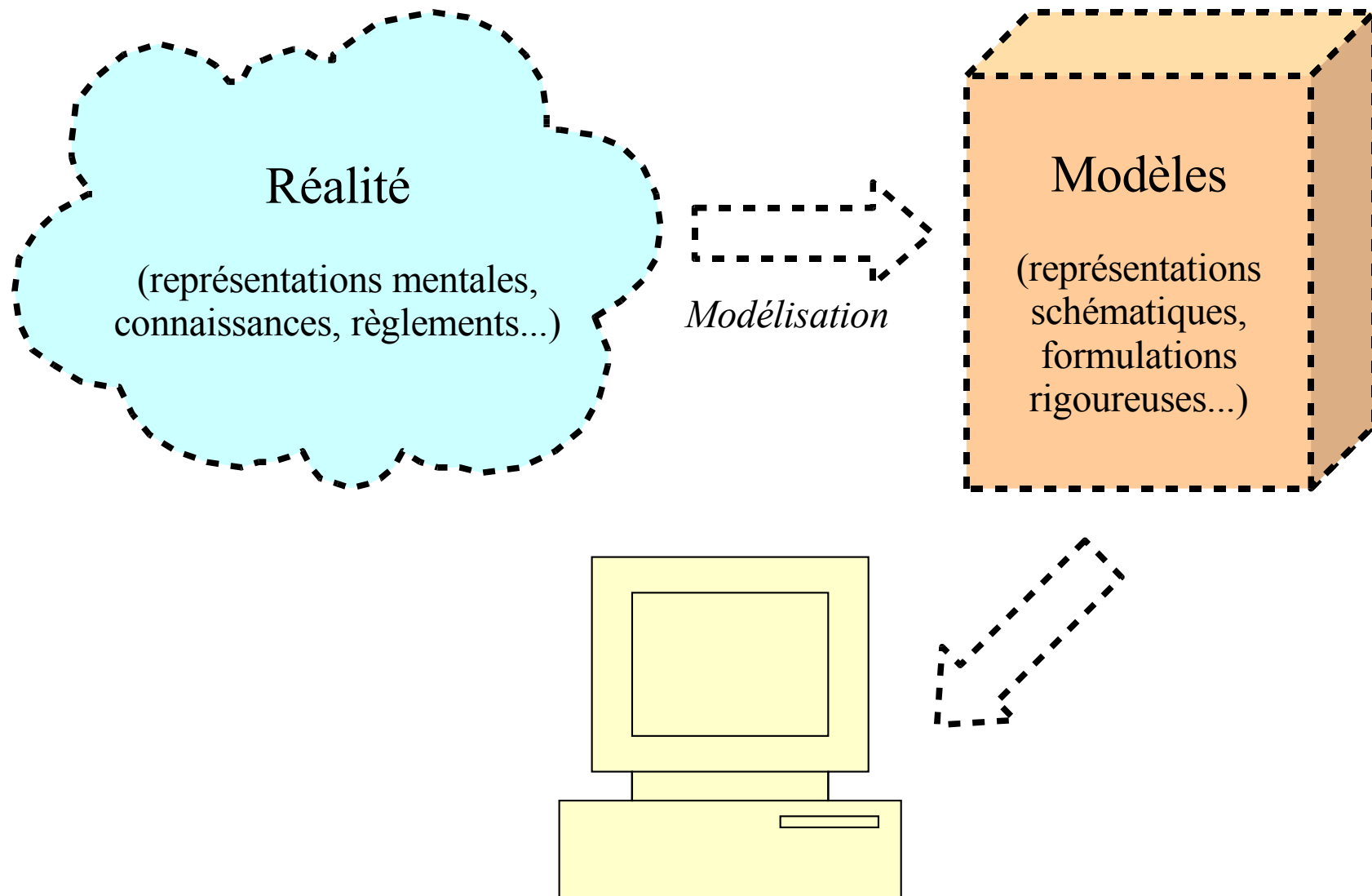
- Développement « *en cascade* » :
 - Chaque étape ne débute que lorsque la précédente est achevée.
- Développement « *en V* » :
 - A chaque étape correspond une étape de test spécifique.
 - Les tests peuvent être définis avant les phases de test.
- Développement *non linéaire* :
 - On commence par développer un sous ensemble des fonctionnalités du logiciel, de manière linéaire ;
 - On itère un modèle linéaire pour développer *incrémentalement* de plus en plus de fonctionnalités, jusqu'à achèvement du projet.

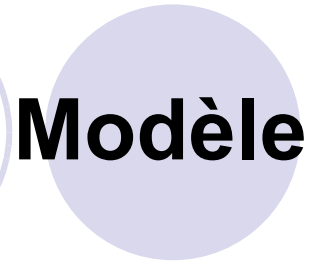
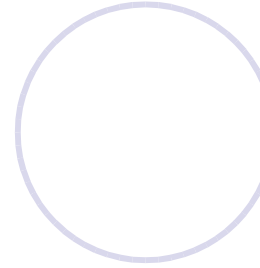
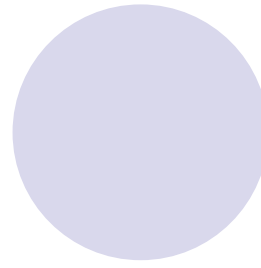
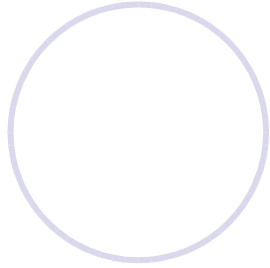
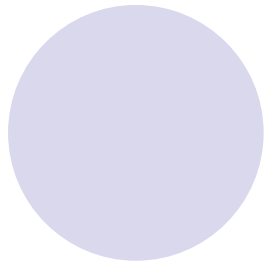


Démarche et modèles

- *Démarche* : succession d'étapes pour :
 - Mieux maîtriser le déroulement d'un projet ;
 - Meilleure visibilité pour les utilisateurs sur certains résultats intermédiaires et garantir que le résultat final sera celui attendu.
- A chaque étape, on produit des *modèles* à différents niveaux d'abstraction, un utilisant un formalisme :
 - Un langage formel ;
 - Un langage semi-formel généralement graphique ;
 - Un langage naturel ;

Modélisation





- Un modèle est une *représentation abstraite* de la réalité qui exclut certains détails du monde réel.
 - Il permet de réduire la complexité d'un phénomène en éliminant les détails qui n'influencent pas son comportement de manière significative.
 - Il reflète ce que le concepteur croit important pour la compréhension et la prédiction du phénomène modélisé, les limites du phénomène modélisé dépendent des objectifs du modèle.



Intérêt de la modélisation

- Modéliser un système avant sa réalisation permet de :
 - Comprendre le fonctionnement du système ;
 - De maîtriser sa complexité ;
 - D'assurer sa cohérence ;
 - Pouvoir communiquer au sein de l'équipe de réalisation .
- Dans le domaine de l'ingénierie du logiciel, le modèle :
 - Permet de bien répartir les tâches et d'automatiser certaines d'entre elles ;
 - Est un facteur de réduction de coût et de délais ;
 - D'assurer un bon niveau de qualité et une maintenance efficace.



Propriétés attendues

- Le choix du modèle a une influence capitale sur les solutions obtenues. Il faut donc bien choisir les modèles employés.
 - Les modèles doivent pouvoir décrire un système à différents niveaux d'abstraction.
 - Les modèles doivent avoir un comportement proche de la réalité.
 - Un seul modèle est souvent insuffisant. Les systèmes non-triviaux sont mieux modélisés par un ensemble de modèles indépendants.
- Selon les modèles employés, la *démarche de modélisation* n'est pas la même.

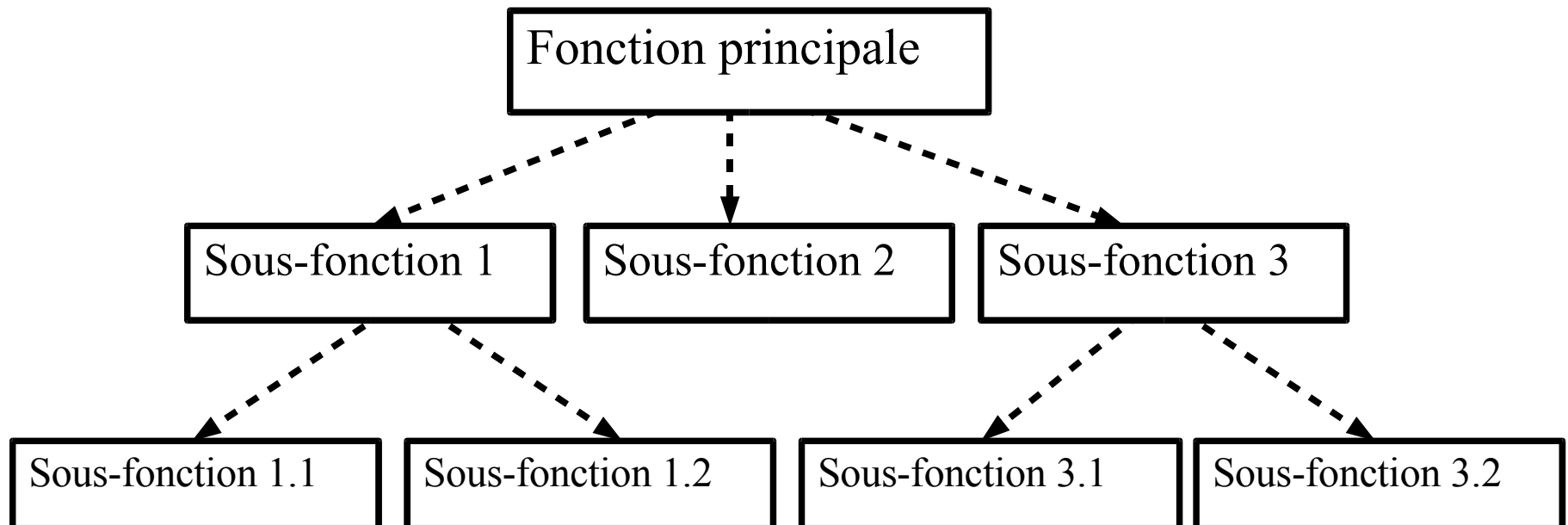


Approches de la modélisation dans le domaine du logiciel

- Approche *fonctionnelle*.
 - Approche traditionnelle utilisant des procédures et des fonctions.
 - Les grand programmes sont ainsi décomposés en sous programmes.
- Approche *orientée objets*.
 - On identifie les éléments du système et on en fait des objets.
 - On cherche à faire collaborer ces objets pour qu'ils accomplissent la tâche voulue.

Modélisation par décomposition fonctionnelle

- Approche *descendante* :
 - Décomposer la fonction globale jusqu'à obtenir des fonctions simples à appréhender et donc à programmer.
- La *fonction* donne la forme du système





Critique de l'approche fonctionnelle

- *Avantages :*
 - Organisée, réfléchie, logique.
 - Ordonnée, réduit la complexité.
- *Inconvénients :*
 - Comment assurer l'évolution du logiciel ?
 - Comment réutiliser les parties déjà développées ?
 - Comment structurer les données ?



Approche orientée objets

- La *Conception Orientée Objet* (COO) est la méthode qui conduit à des architectures logicielles fondées sur les **objets du système**, plutôt que sur la fonction qu'il est censé réaliser.
- La structure du système lui donne sa forme.
- On peut partir des objets du domaine (briques de base) et remonter vers le système global
 - *Approche ascendante*
 - Attention, l'approche objet n'est pas seulement ascendante



Critique de l'approche orientée objet

- *Avantages :*
 - Simple : peu de concepts de base ;
 - Raisonnement par abstraction sur les objets du domaine (voir critères de qualité).
- *Inconvénients :*
 - Parfois moins intuitive que l'approche fonctionnelle.
 - Rien dans les concepts de base objets ne dicte comment modéliser la structure objet d'un système de manière pertinente.
 - L'application des concepts objets nécessite de l'expérience et une grande rigueur pour éviter les ambiguïtés, les incompréhensions, etc.



Penser et programmer objet

- La *pensée objet* doit être dissociée de la programmation objet.
 - Le langage objet ne présente qu'une manière particulière d'implémenter certains concepts.
 - Ils ne valident en rien la conception.
- Connaître des *langages de programmation* comme Java ou C++ n'est pas suffisant.
 - Il faut savoir s'en servir en utilisant un moyen de modélisation, d'analyse et de conception adéquat.
 - Sans les langages de haut niveau, il est difficile de comparer plusieurs solutions de modélisation objet.



Utilisation de l'approche objet

- Pour une utilisation efficace de l'approche objet, il faut :
 - Une *démarche d'analyse* et de conception objet.
 - Ne pas effectuer une analyse fonctionnelle et se contenter d'une implémentation objet ;
 - Un *langage* permettant de représenter les concepts abstraits, de limiter les ambiguïtés et de faciliter l'analyse :
 - Définition des différents *modèles* permettant de décrire tous les aspects du système,
 - Indépendamment des langages de programmation.



Langages de modélisation

- Il est nécessaire de disposer d'un langage de modélisation qui définisse :
 - La sémantique des concepts
 - Une notation pour la représentation de concepts
 - Des règles d'utilisation des concepts et de construction
- L'industrie du logiciel dispose de nombreux langages de modélisation
 - Adaptés aux systèmes *procéduraux* (MERISE...)
 - Adaptés aux systèmes *temps réel* (ROOM, SADT...)
 - Adaptés aux systèmes à *objets* (OMT, Booch, UML...)
- Le rôle des outils est primordial pour l'utilisabilité des langages de modélisation.



Langages objets

- Langages de programmation :
 - SIMULA(1967), Smalltalk(1972), C++(1985), JAVA (1995)
 - Entre 89 et 94, le nombre de LOO est passé de moins de 10 à une cinquantaine et choisir un langage adéquat devenait difficile ;
- Langages de modélisation :
 - Dans le même temps, des *méthodes objet* sont apparues (Booch, OOSE, OMT...), OMT étant la plus largement utilisée.
 - Chacune s'appuyait sur un langage de modélisation particulier.



Unified Modeling Language

- Au milieu des années 90, les auteurs de Booch, OOSE et OMT ont décidé de créer un langage de modélisation unifié.
 - Modéliser un système des concepts à l'exécutable, en utilisant les techniques orientée objet ;
 - Réduire la complexité de la modélisation ;
 - Utilisable par l'homme comme la machine.
 - Représentation graphiques mais suffisamment disposant des qualités formelles pour être traduit automatiquement en LOO.
- Officiellement UML est né en 1994.
- UML est dans sa version 2 depuis 2005.

Diagramme de cas d'utilisation

UML 2.0



Spécifier les besoins

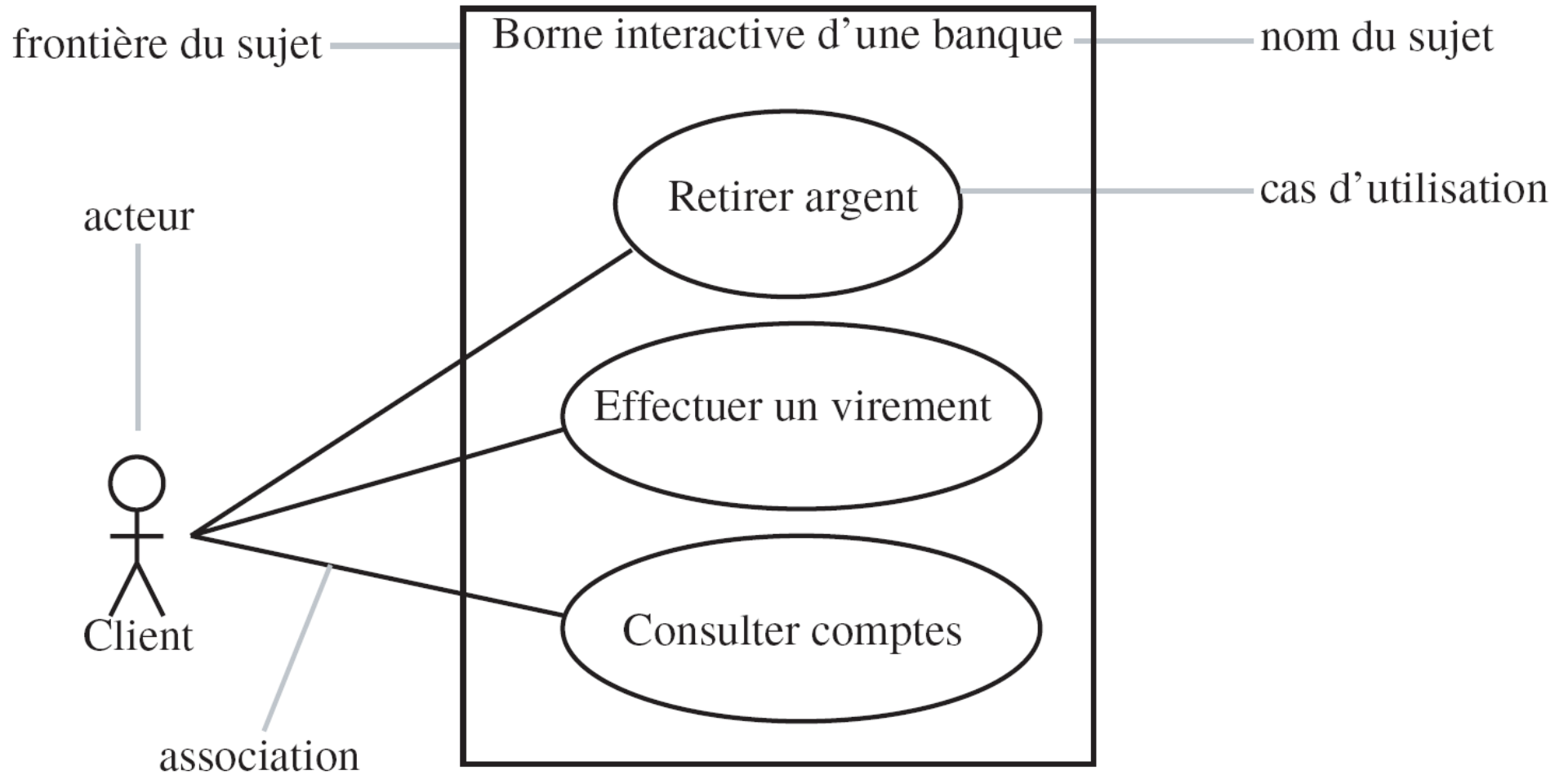
- Le *diagramme des cas d'utilisation* est le premier modèle à construire
- Il permet de capturer les besoins des utilisateurs
 - Ne pas négliger cette étape permet de produire un logiciel conforme aux attentes des utilisateurs.
 - Pour rédiger les cas d'utilisation, on peut se fonder sur des entretiens avec les utilisateurs.

The header features a row of five circles. The first, third, and fifth circles are solid light purple. The second and fourth circles are white with a light purple outline. The text 'Cas d'utilisation' is positioned over the third and fourth circles.

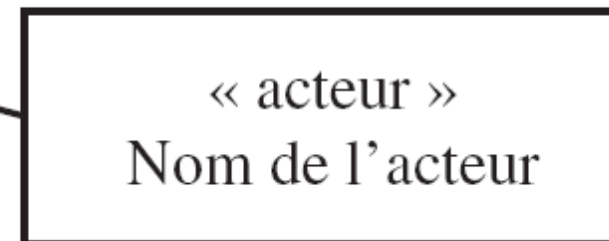
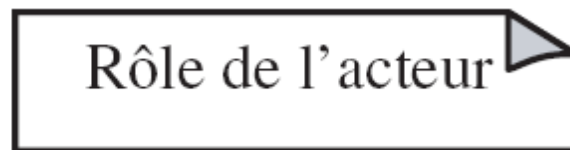
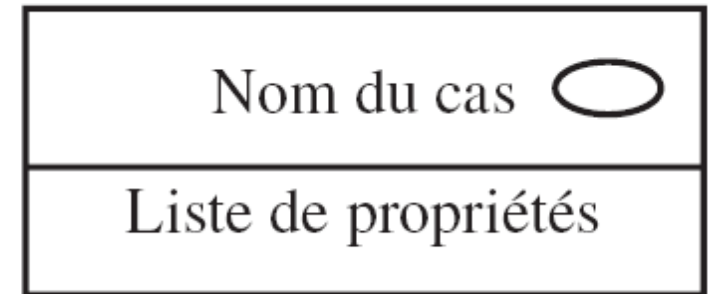
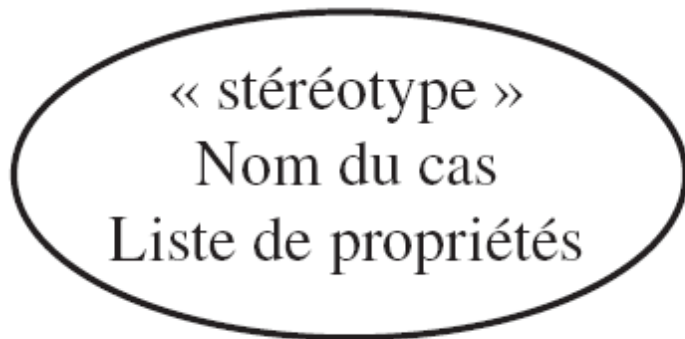
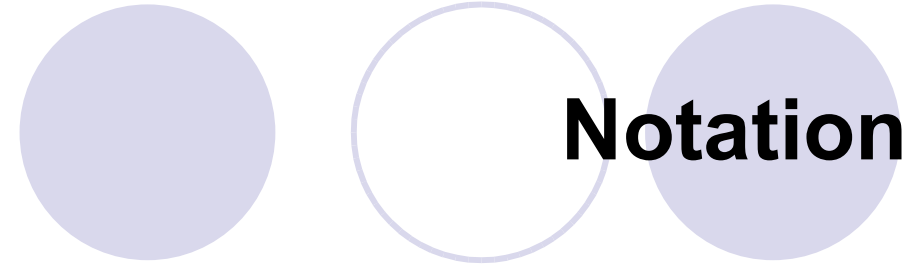
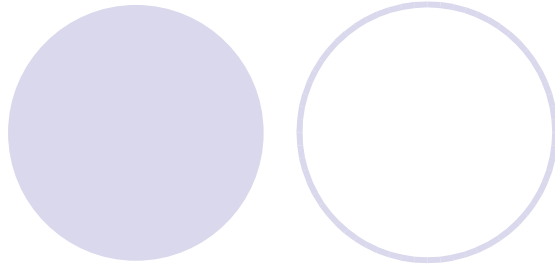
Cas d'utilisation

- Un *cas d'utilisation* est une manière spécifique d'utiliser un système.
- Les *acteurs* sont à l'extérieur du système ; ils modélisent tout ce qui interagit avec lui.
 - *Attention* : définir les bornes/frontières du système est primordial.
 - Tous les acteurs interagissent directement avec le système
- Un cas d'utilisation réalise un service de bout en bout, avec un déclenchement, un déroulement et une fin, pour l'acteur qui l'initie.

Diagramme de cas d'utilisation



Notation





Stéréotypes et classeurs

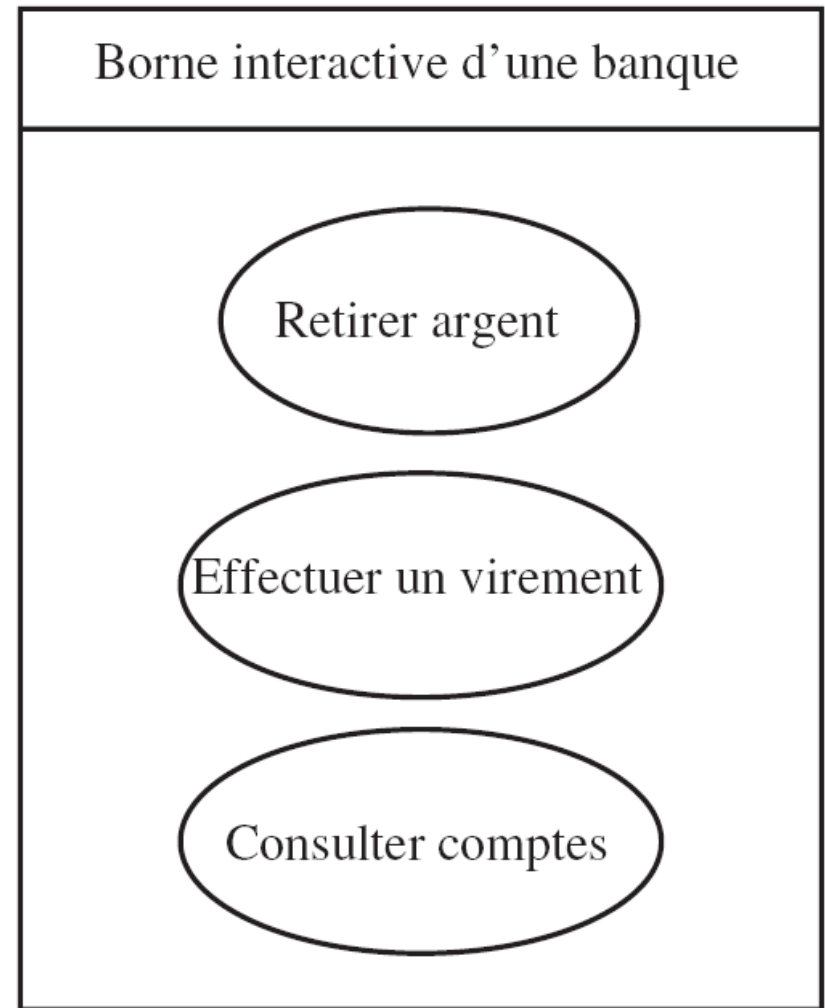
- Un *stéréotype* représente une variation d'un élément de modèle existant.
- Un *classeur* est un élément de modélisation qui décrit une unité comportementale ou structurelle.
 - Les acteurs et les cas d'utilisation sont des classeurs. Cette notion englobe aussi les classes, des parties d'un système, etc.
 - Exemple de classeur représentant l'ensemble des cas d'utilisation :



Borne interactive d'une banque

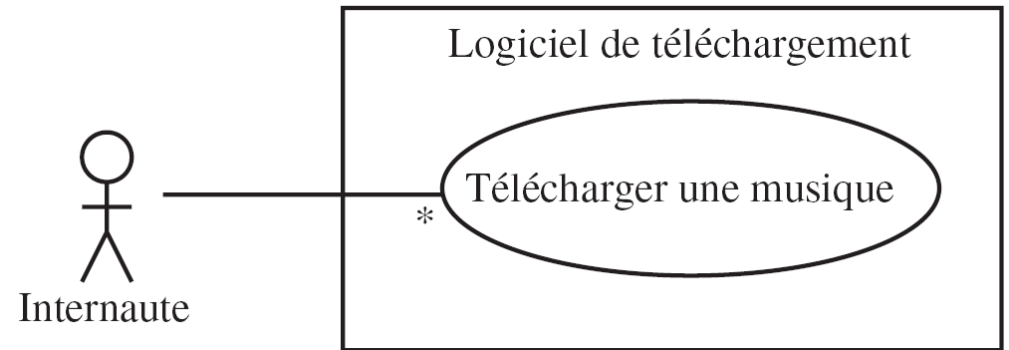
Représentation des classeurs

- Un classeur se représente par un *rectangle* contenant éventuellement des *compartiments*
- La figure ci-contre montre comment il est possible de faire figurer explicitement des cas d'utilisation dans un classeur.



Relations entre cas d'utilisation en acteurs

- Les acteurs impliqués dans un cas d'utilisation lui sont liés par une *association*.
- Un acteur peut utiliser plusieurs fois le même cas d'utilisation.
- Les *multiplicités* (ici *) indiquent le nombre d'instances possibles de l'association.



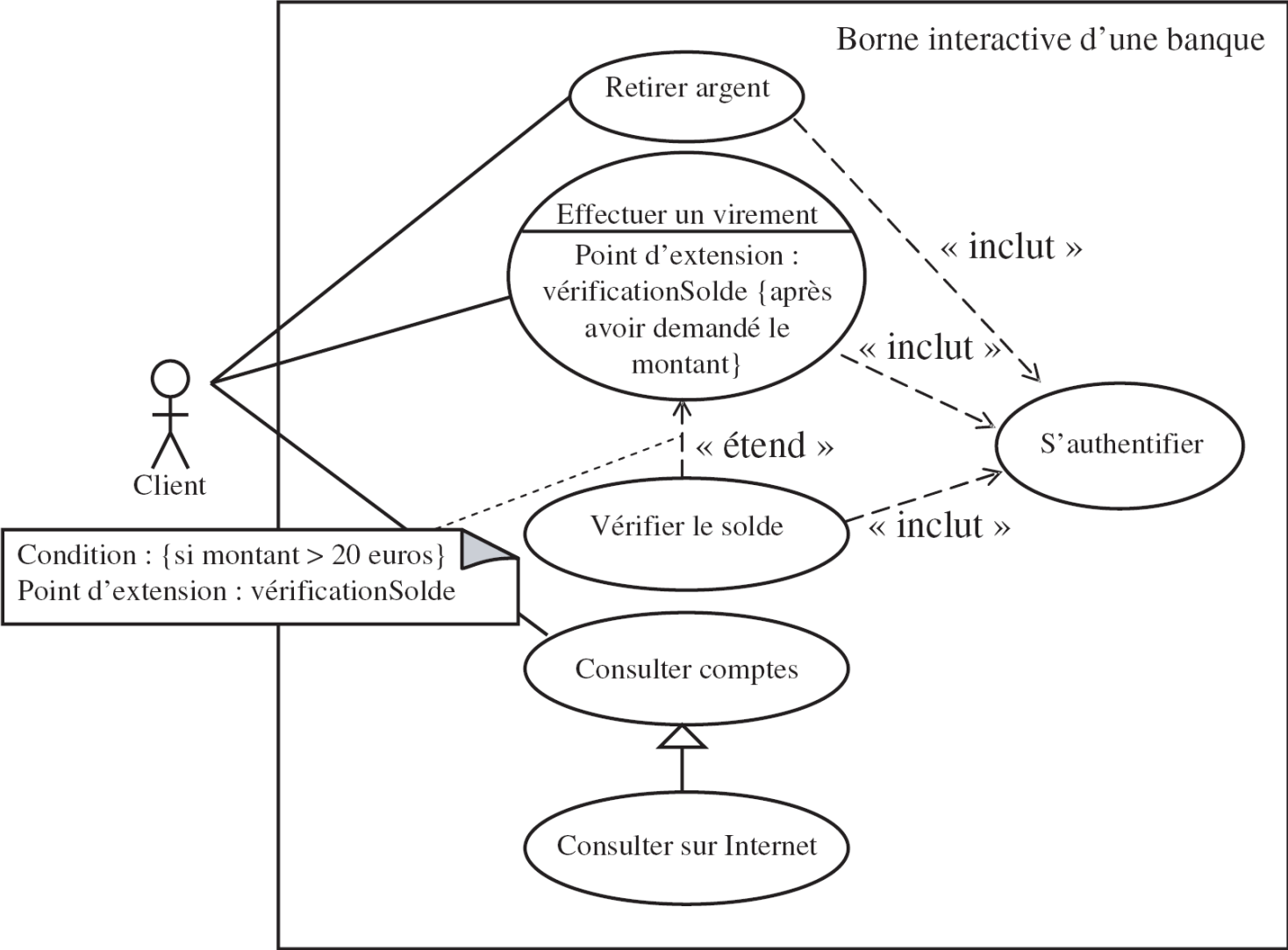


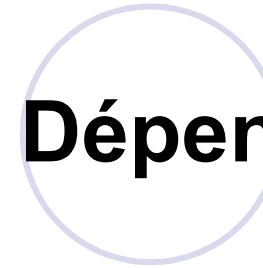
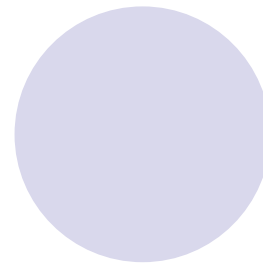
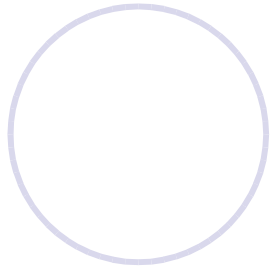
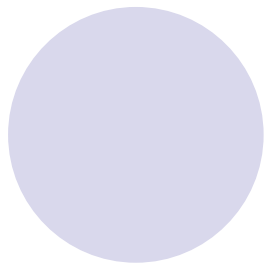
Relations entre cas d'utilisation

- *Inclusion* : un cas A inclut un cas B si le comportement décrit par le cas A inclut le comportement du cas B. Lorsque A est sollicité, B l'est obligatoirement, comme une partie de A.
 - Exemple : la connexion à un serveur inclut une identification. Il n'y a jamais de connexion sans identification.
- *Extension* : si le comportement de A peut être étendu par le comportement de B, on dit alors que B étend A. Exécuter A peut éventuellement entraîner l'exécution de B. Dans le déroulement de A, le moment où l'on peut avoir besoin d'exécuter A est un *point d'extension*. Une extension est souvent soumise à *condition*.
 - Contrairement à l'inclusion, l'extension est *optionnelle*.
- *Généralisation* : un cas A est une généralisation d'un cas B si A est un cas particulier de B (« une sorte de »).



Représentation des relations entre cas d'utilisation





Dépendances

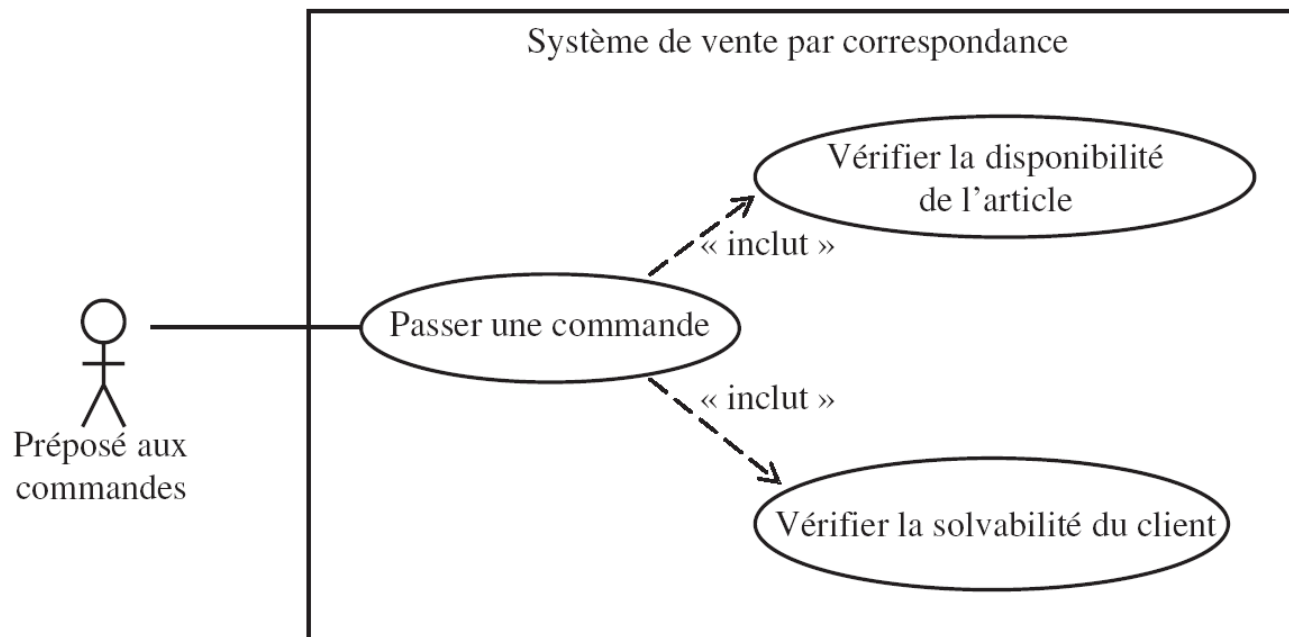
- Les inclusions et les extensions sont représentées par des *dépendances*.
 - Lorsqu'un cas B inclut un cas A, B dépend de A.
 - Lorsqu'un cas A étend un cas B, B dépend aussi de A.
 - On note la dépendance par une flèche pointillée $B \dashrightarrow A$ qui se lit « B dépend de A ».

Attention au sens des flèches !

- Une dépendance est souvent stéréotypée. Ici,
 - le stéréotype « include » indique que la relation de dépendance est une inclusion
 - le stéréotype « extend » indique une extension

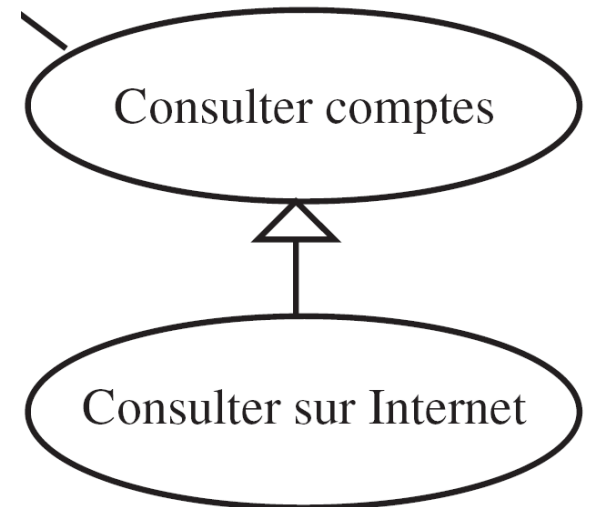
Réutilisabilité et décomposition

- Dans l'exemple précédent, les relations entre cas permettaient la *réutilisabilité* du cas « s'authentifier ».
- Dans l'exemple suivant, on procède à la *décomposition* d'un cas complexe en cas plus simples.



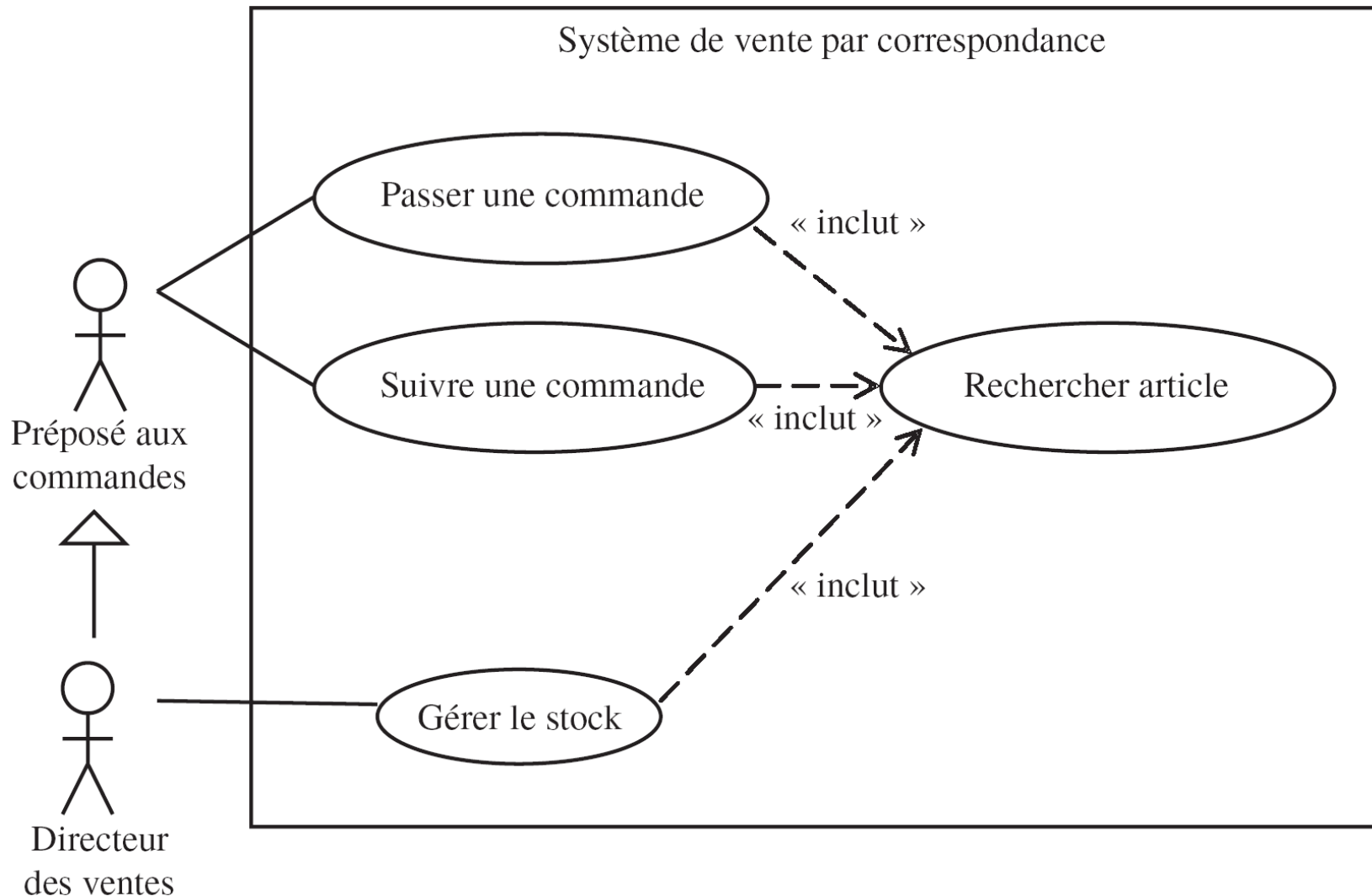
Généralisation

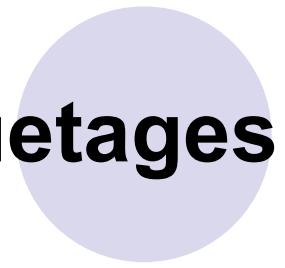
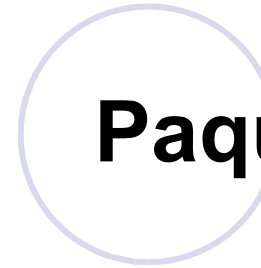
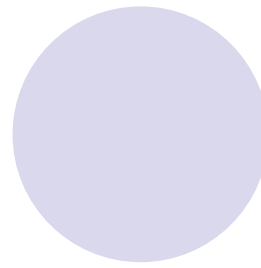
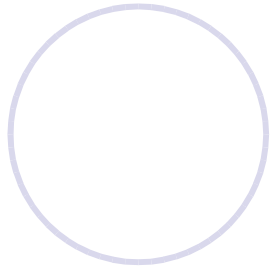
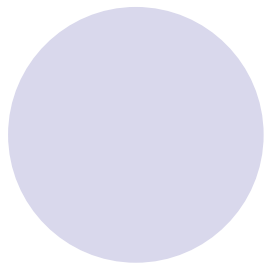
- « Consulter sur Internet » est un cas particulier de « consulter comptes ».
- La flèche pointe vers l'élément général.
- Cette relation de généralisation/spécialisation est présente dans la plupart des diagrammes UML et se traduit par le concept d'*héritage* dans les langages orientés objet.



Relations entre acteurs

- Une seule relation possible : la *généralisation*.



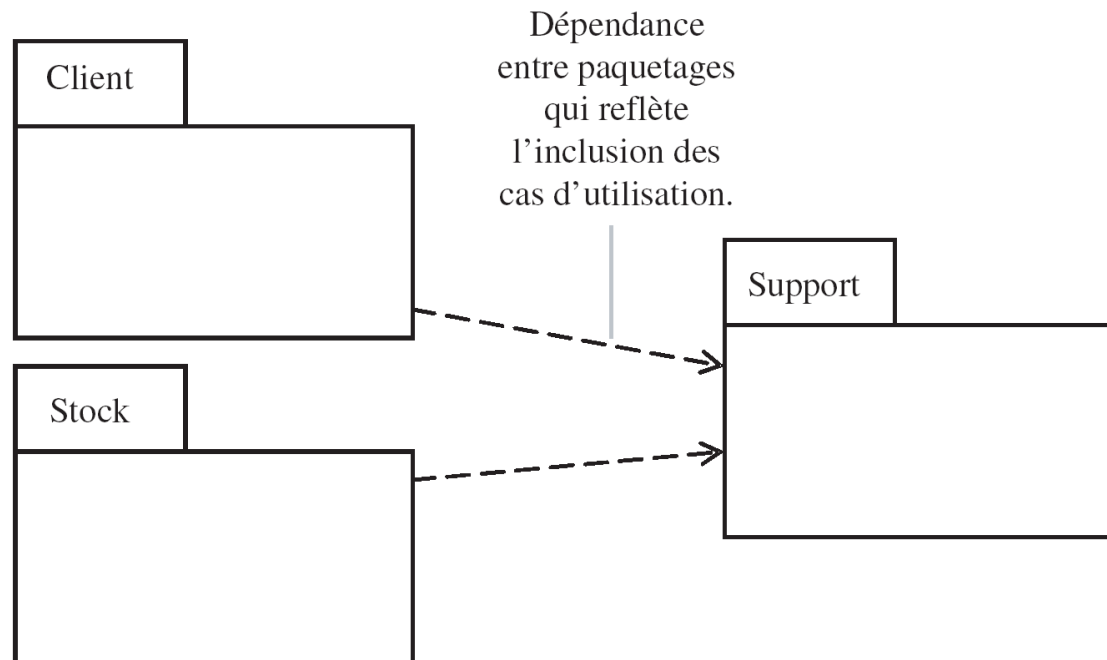


Paquetages

- Un *paquetage* permet d'organiser des éléments de modélisation en groupes. Un paquetage peut contenir des classes, des cas d'utilisations, des interfaces, etc.
- Les paquetages peuvent être imbriqués les uns dans les autres.
 - Pour accéder au contenu de tels paquetages, il faut utiliser des deux-points comme séparateur des noms de paquetage.
 - Par exemple, si un paquetage B est inclus dans un paquetage A et contient une classe X, il faut écrire A::B::X pour pouvoir utiliser la classe X en dehors du contexte du paquetage B.

Regroupement de cas d'utilisation

- Le regroupement de cas en paquetages peut se faire par acteur ou par domaine fonctionnel.
- Un diagramme de cas d'utilisation peut contenir plusieurs paquetages et des paquetages peuvent être inclus dans d'autres paquetages.





Identification des acteurs

- Les principaux acteurs sont les utilisateurs du système.
 - Un acteur correspond à un *rôle*, pas à une personne physique
 - Une même personne physique peut être représentée par plusieurs acteurs si elle a plusieurs rôles
 - Si plusieurs personnes jouent le même rôle vis-à-vis du système, elles seront représentées par un seul acteur
- En plus des utilisateurs, les acteurs peuvent être :
 - des périphériques manipulés par le système (imprimantes...) ;
 - des logiciels déjà disponibles à intégrer dans le projet ;
 - des systèmes informatiques externes au système mais qui interagissent avec lui, etc.
- Pour faciliter la recherche des acteurs, on se fonde sur les *frontières* du système.



Acteur principal et secondaire

- L'acteur est dit « *principal* » pour un cas d'utilisation lorsque le cas d'utilisation rend service à cet acteur.
- Les autres acteurs sont dits « *secondaires* ».
 - Un cas d'utilisation a au plus un acteur principal, et un ensemble – éventuellement vide – d'acteurs secondaires.
 - On peut stéréotyper les relations entre un cas et ses acteurs secondaires (stéréotype « secondary »).
- Par exemple, l'acteur principal d'un cas de retrait d'argent dans un DAB est la personne qui fait le retrait, tandis que la banque qui vérifie le solde du compte est un acteur secondaire.
 - Les acteurs secondaires sont sollicités par le système alors que le plus souvent, les acteurs principaux ont l'initiative des interactions



Recenser les cas d'utilisation

- Il n'y a pas une façon unique de repérer les cas d'utilisation.
 - Il faut se placer du point de vue de chaque acteur et déterminer comment il se sert du système, dans quels cas il l'utilise, et à quelles fonctionnalités il doit avoir accès.
 - Il faut éviter les redondances et limiter le nombre de cas en se situant au bon niveau d'abstraction (par exemple, ne pas réduire un cas à une action).
 - Il ne faut pas faire apparaître les détails des cas d'utilisation, mais il faut rester au niveau des grandes fonctions du système.
- Trouver le bon niveau de détail pour les cas d'utilisation est un problème difficile qui nécessite de l'expérience.



Description des cas d'utilisation

- Le diagramme de cas d'utilisation décrit les grandes fonctions d'un système du point de vue des acteurs, mais n'expose pas de façon détaillée le dialogue entre les acteurs et les cas d'utilisation.

The header features a series of five circles. The first two are on the left, followed by a gap, then three more circles on the right. The third circle from the left (the first after the gap) is filled with a light purple color and contains the text 'Description textuelle' in bold black font. The other four circles are empty with a light purple outline.

Description textuelle

- *Identification :*

- Nom du cas : retrait d'espèces en euros
- Objectif : détaille les étapes permettant à un guichetier d'effectuer l'opération de retrait d'euros pour un client.
- Acteurs : Guichetier, système central (secondaire)
- Date : 18/05
- Responsables : Toto
- Version : 1.0

- *Séquencements :*

- Le cas d'utilisation commence lorsqu'un client demande le retrait d'espèces en euros
- Pré-conditions
 - Le client possède un compte (donne son numéro de compte)



Description textuelle

- Enchaînement nominal
 1. Le guichetier saisit le numéro de compte client
 2. L'application valide le compte auprès du système central
 3. L'application demande le type d'opération au guichetier
 4. Le guichetier sélectionne un retrait d'espèces de 200 euros
 5. Le système «guichet» interroge le système central pour s'assurer que le compte est suffisamment approvisionné
 6. Le système central effectue le débit du compte
 7. Le système notifie au guichetier qu'il peut délivrer le montant demandé
- Enchaînements alternatifs
 - ...
- Post-conditions
 - Le guichetier ferme le compte
 - Le client récupère l'argent



Description textuelle

- Post-conditions
 - Le guichetier ferme le compte
 - Le client récupère l'argent
- *Rubriques optionnelles*
 - Contraintes non fonctionnelles
 - Fiabilité : les accès doivent être extrêmement sécurisés
 - Confidentialité : les informations concernant le client ne doivent pas être divulguer
 - Contraintes liées à l'interface homme-machine :
 - Donner la possibilité d'accéder aux autres comptes du client
 - Ne pas accéder à plusieurs comptes au même temps
 - Toujours demander la validation des opérations de retrait

Description à l'aide d'un diagramme d'interaction

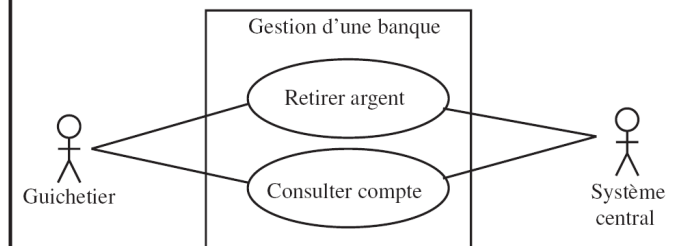
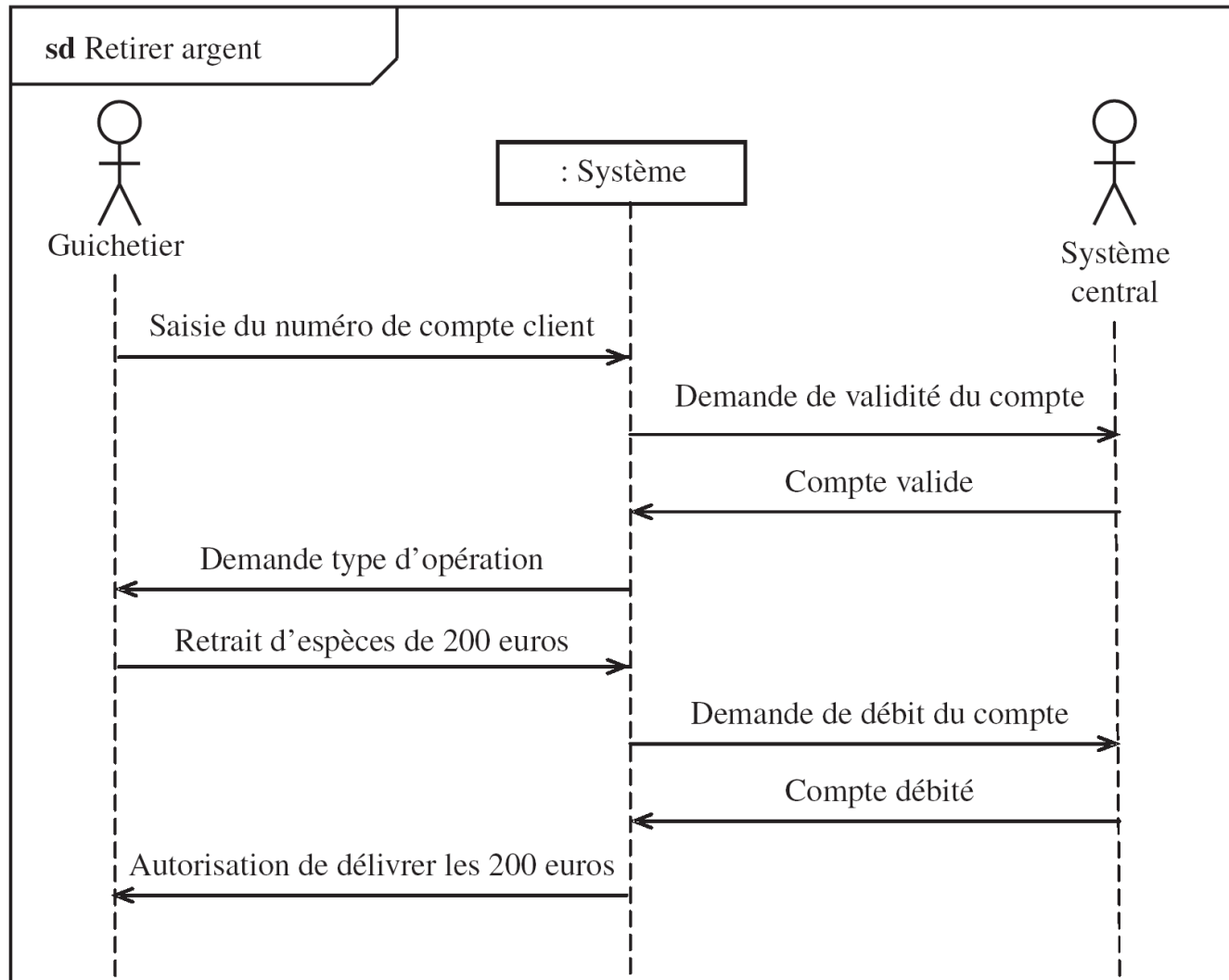
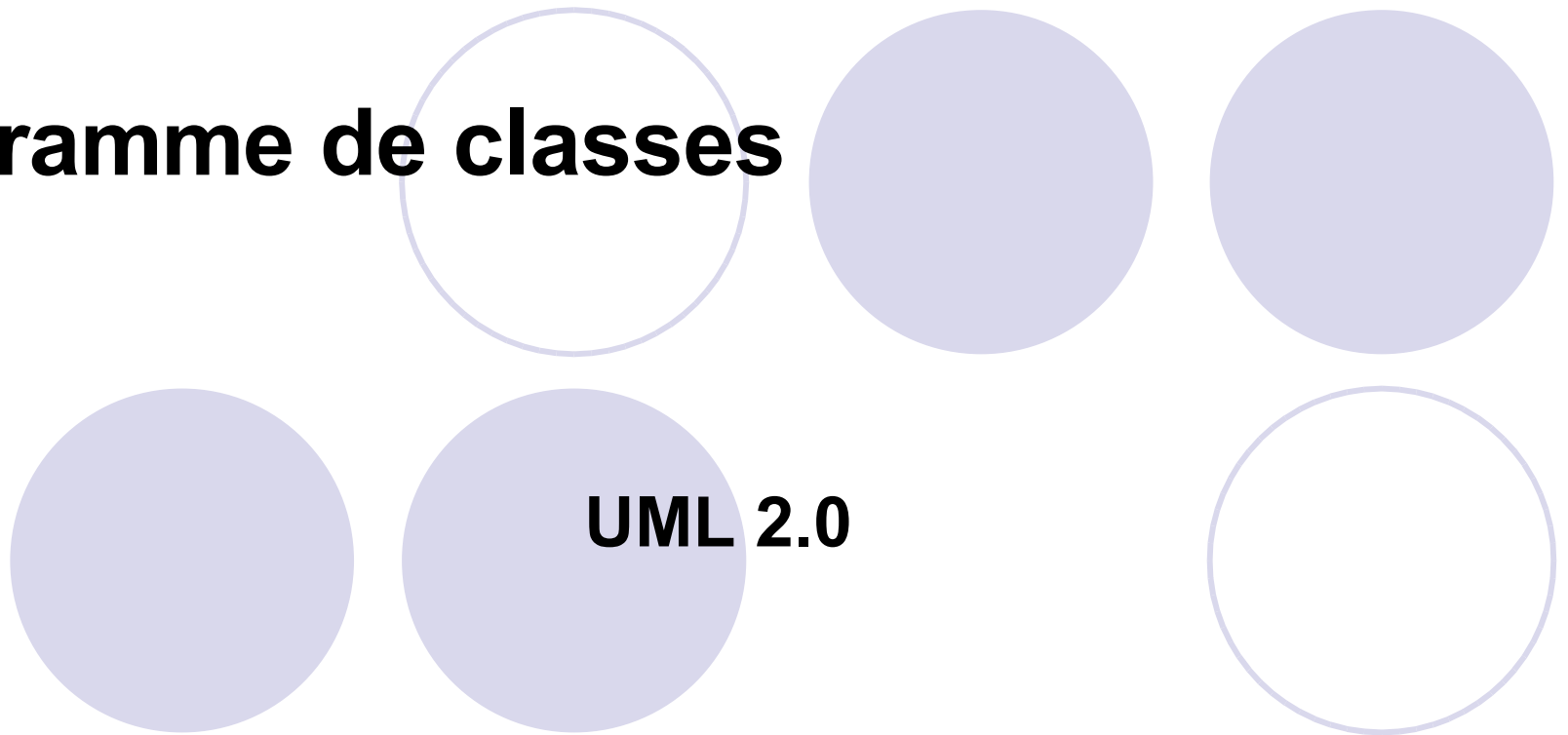


Diagramme de classes





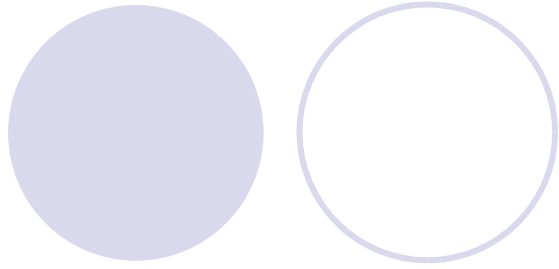
Objectif

- Chaque langage de Programmation Orienté Objets (POO) donne un moyen spécifique d'implémenter le paradigme objet. UML permet de :
 - Définir le problème indépendamment du langage (pointeurs ou pas, héritage multiple ou pas, etc.)
 - De mener une Conception Orientée Objet (COO)
 - Il est possible, dans un langage comme C++ ou Java, de concevoir un programme syntaxiquement juste sans adopter une approche objet.
- Le diagramme des classes permet de fournir une représentation abstraite des objets du système qui vont interagir ensemble pour réaliser les cas d'utilisation.



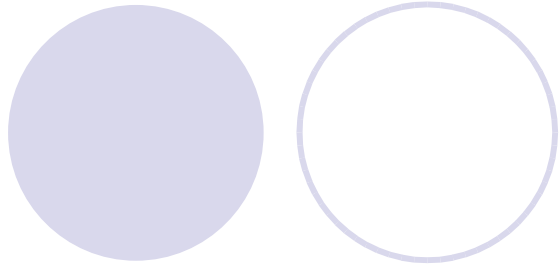
Concepts et instances

- Une *instance* est une concrétisation d'un *concept* abstrait.
 - Exemples :
 - Concept : Amitié / Instance : Jean est l'ami de François
 - Concept : Stylo / Instance : le stylo que vous utilisez à ce moment précis est une instance du concept stylo : il a sa propre forme, sa propre couleur, son propre niveau d'usure, etc.
- Un *objet* est une instance d'une *classe*
 - Classe : Chat / Objet : le chat du voisin du troisième étage.
 - La classe spécifie la manière dont tous les objets de même type seront décrits (forme, couleur et niveau d'usure d'un stylo).
- Un *lien* est une instance d'*association*
 - Association : Concept « travailler pour » qui lie deux classes Personne et Entreprise
 - Lien : instance qui lie Jean avec l'entreprise SNCF

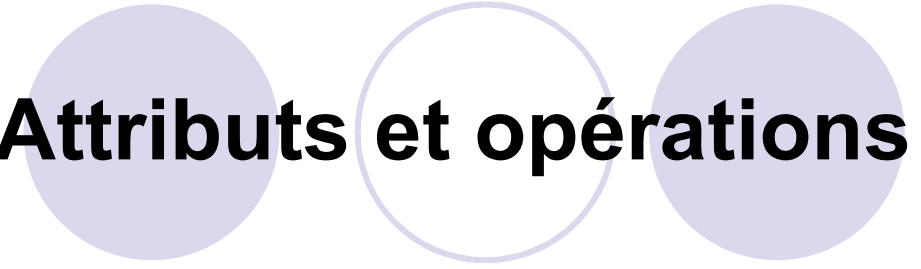


Classes et objets

- Une *classe* est la description d'un ensemble d'objets ayant une sémantique, des attributs, des méthodes et des relations en commun. Un *objet* est une instance d'une classe.
- Une classe est un classeur modélisant la structure d'un objet. Elle spécifie l'ensemble des caractéristiques qui composent un objet.
 - Une classe est composée d'un *nom*, d'*attributs* et d'*opérations*.
 - Selon l'avancement de la modélisation, ces informations ne sont pas forcément toutes connues.
- D'autres compartiments peuvent être ajoutés : responsabilités, exceptions, etc.

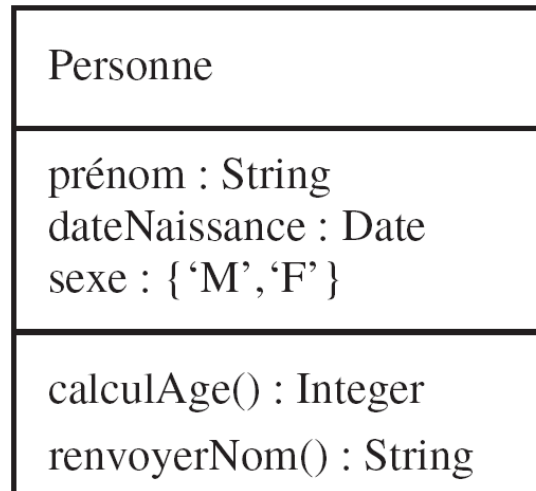


Attributs et opérations



- Les attributs et les opérations sont les *propriétés* d'une classe.
 - Un *attribut* décrit une donnée de la classe.
 - Les types des attributs et leurs initialisation ainsi que les modificateurs d'accès peuvent être précisés dans le modèle
 - Les propriétés décrites par les attributs prennent des valeurs lorsque la classe est instanciée.
 - Une *opération* est un service offert par la classe (un traitement que les objets correspondant peuvent effectuer). Une classe dispose d'un ensemble d'opérations éventuellement vide.

Représentation d'une classe



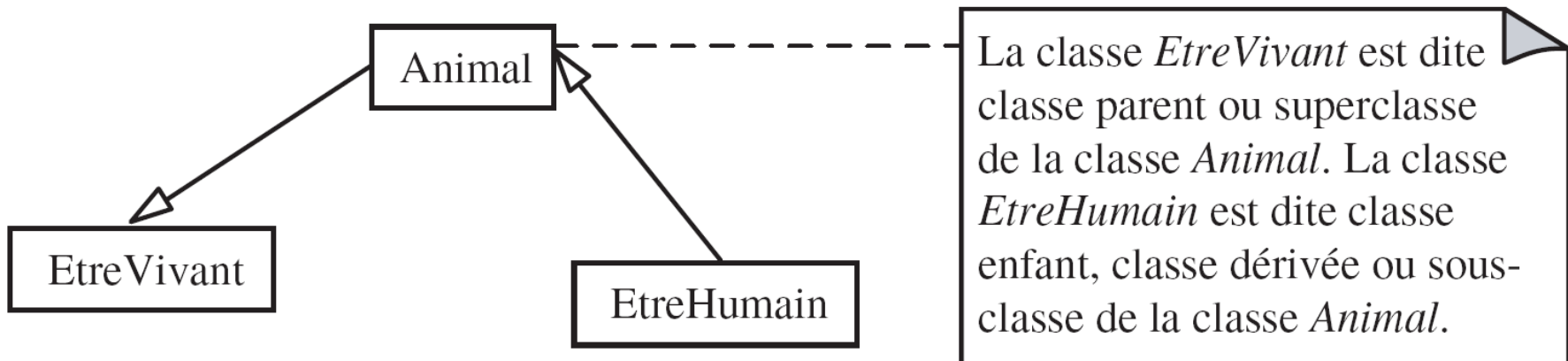
Le nom de la classe doit être significatif et complet. Il commence par une majuscule. S'il est composé de plusieurs mots, la première lettre de chaque mot doit être une majuscule. Personne

Liste des attributs de la classe avec les modificateurs d'accès éventuels. Certains attributs n'apparaissent pas directement, ils seront déduits à partir des relations entre classes.

Liste des méthodes de la classe. Quand la méthode accepte des paramètres alors ces paramètres et leurs types sont ajoutés entre les parenthèses.

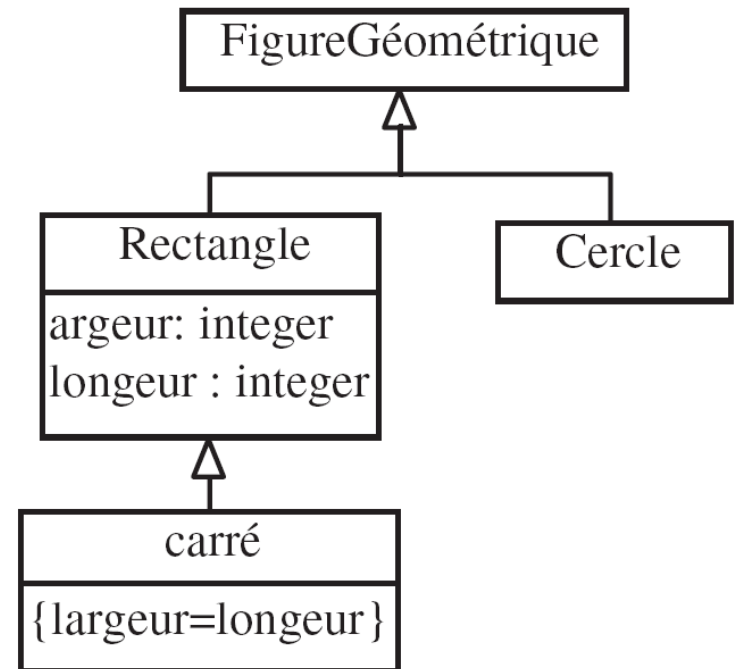
Relation d'héritage

- L'*héritage* une relation de spécialisation/*généralisation*.
 - Les éléments spécialisés héritent de la structure et du comportement des éléments plus généraux.
- La *spécialisation* de classes consiste à réduire le domaine de définition des objets
 - en ajoutant de nouveaux attributs
 - en réduisant le domaine de définition des attributs existants.



Héritage et attributs

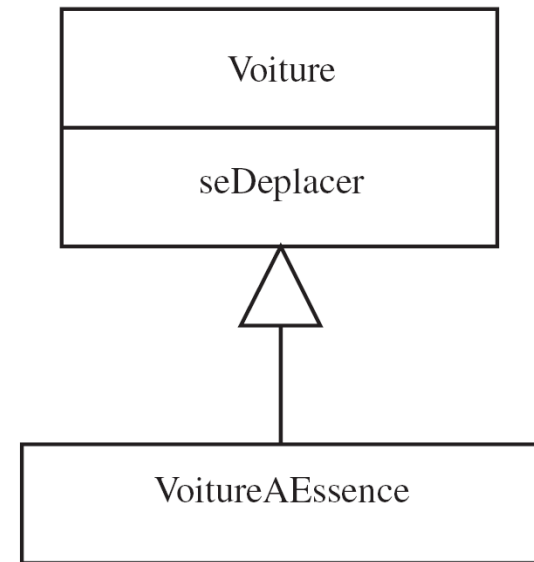
- La classe Rectangle spécialise FigureGéométrique en ajoutant des attributs
- La classe Carré spécialise la classe Rectangle en réduisant le domaine de définition des attributs existants
 - Contrainte {largeur = longueur}.



Implémentation de l'héritage

```
public class Voiture{  
    public void seDeplacer() {  
        // ...  
    }  
}
```

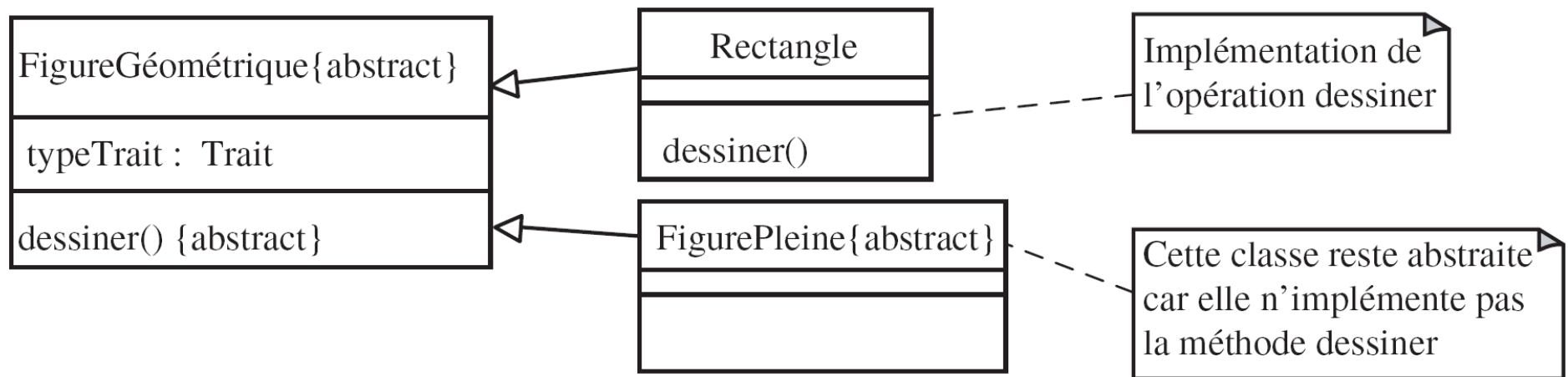
```
public class VoitureAEssence extends  
    Voiture{  
    // ...  
}
```



*Attention : le « extends » Java n'a rien à voir
avec le « extends » UML*

Classes abstraites

- Une méthode est dite *abstraite* lorsqu'on connaît son entête mais pas la manière dont elle peut être réalisée.
- Une classe est dite *abstraite* lorsqu'elle définit au moins une méthode abstraite ou lorsqu'une classe parent contient une méthode abstraite non encore réalisée.



Compartiment du nom de la classe

- Syntaxe de la déclaration d'un nom d'une classe :
[«<stéréotype>»] [<nomPaquetage1>::...::<nomPaquetageN>::]
<nomClasse> [{[abstract], [<auteur>], [<état>], ...}]
- Un nom de classe commence par une majuscule. Quand le nom est composé, chaque mot commence par une majuscule et les espaces sont éliminés.

NomSimple

nomPackage::NomSimple

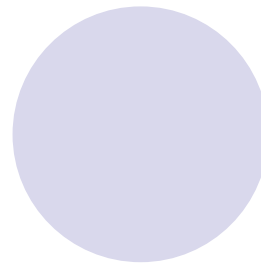
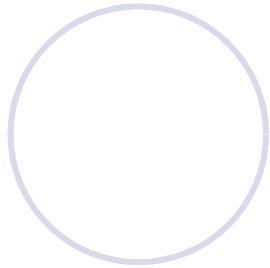
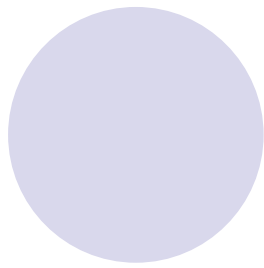
FigureGéométrique

{abstract,
auteur : osmani
état : validée}

« Contrôleur »
LeGardien

Personne

java::util::Vector

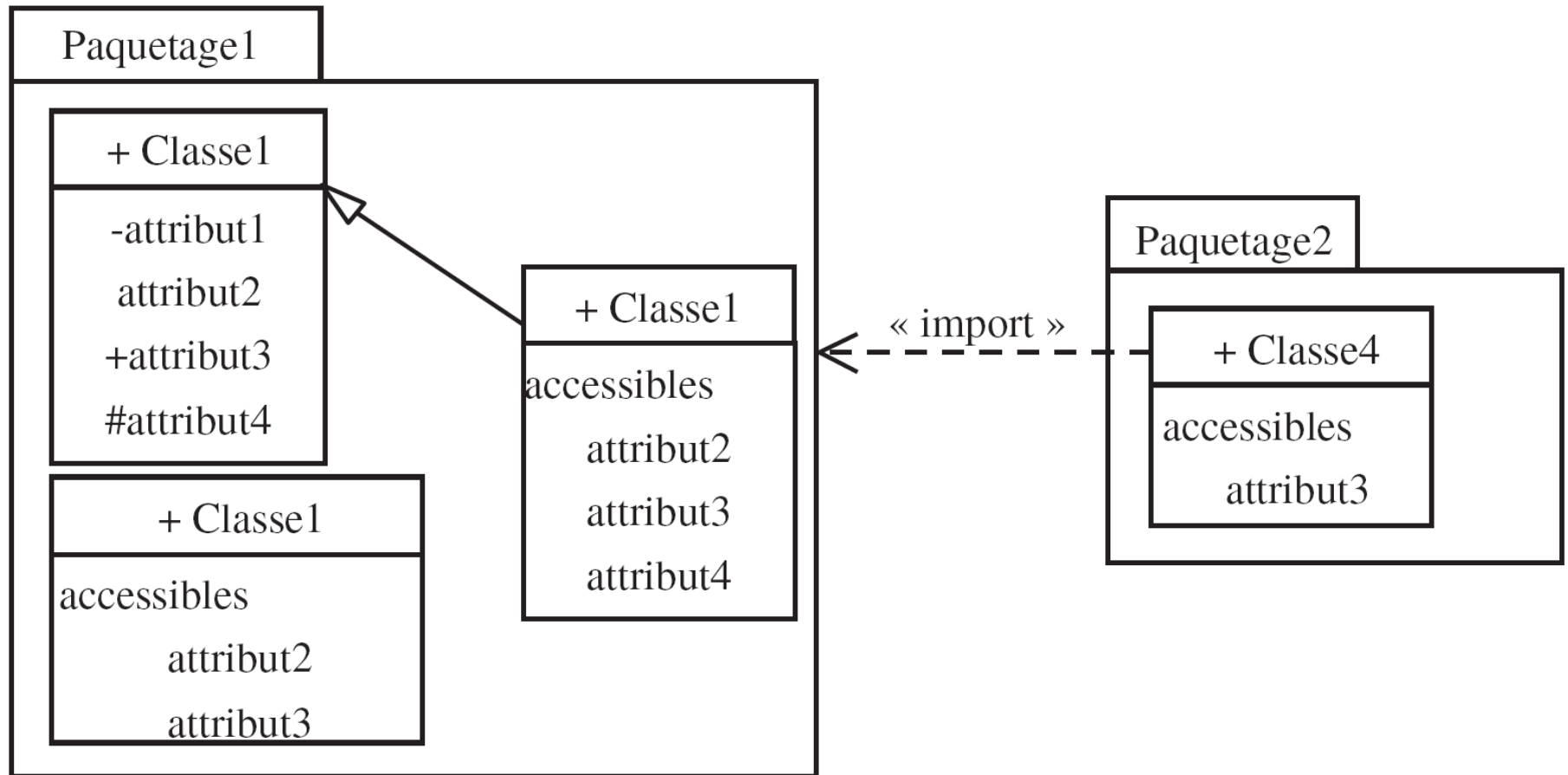


Encapsulation



- L'*encapsulation* est un principe de conception consistant à protéger le coeur d'un système des accès intempestifs venant de l'extérieur.
- En UML, utilisation de *modificateurs d'accès* sur les attributs ou les classes :
 - Public ou « + » : propriété ou classe visible partout
 - Protected ou « # » : propriété ou classe visible dans la classe et par tous ses descendants.
 - Private ou « - » : propriété ou classe visible uniquement dans la classe
 - Aucun caractère, ni mot-clé : propriété ou classe visible uniquement dans le paquetage

Exemple d'encapsulation



- Modificateurs également applicables aux opérations

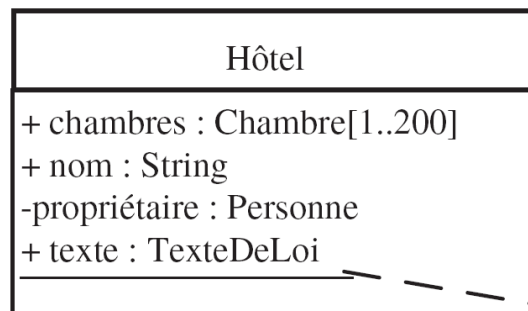
Compartiment des attributs

- Un attribut peut être initialisé et sa visibilité est définie lors de sa déclaration. La syntaxe de la déclaration d'un attribut est la suivante :

```
<modificateur d'accès>[/]
```

```
<nomAttribut>:<nomClasse>['['<multiplicité>']']
```

```
[=<valeurInitiale>]
```



Description des attributs en mettant en évidence la visibilité et la multiplicité. Dans le cas où la multiplicité est représentée par plusieurs intervalles, ils sont ordonnés par ordre croissant. Quand les intervalles sont continus, ils doivent être regroupés. Exemple : [5..6] doit être préféré à "5,6".

Le texte de loi qui régit la profession n'est pas propre à un hôtel. Elle existe en un seul exemplaire, partagée par toutes les instances d'hôtel. Il s'agit d'un attribut de classe (souligné).



Méthodes et Opérations

- Une *opération* est la spécification d'une *méthode* (sa signature) indépendamment de son implémentation
 - UML 2 autorise également la définition des opérations dans n'importe quel langage donné
- Méthodes pour l'opération **+fact(n:integer):integer**

```
{ int resultat =1 ;  
  for (int i = n ; i>0 ; i--) resultat*=i ;  
  return resultat ;  
}
```

```
  { if (n==0 || n==1) return (1) ;  
    return (n * fact(n-1) ; *= i ;  
  }
```



Compartiment des opérations

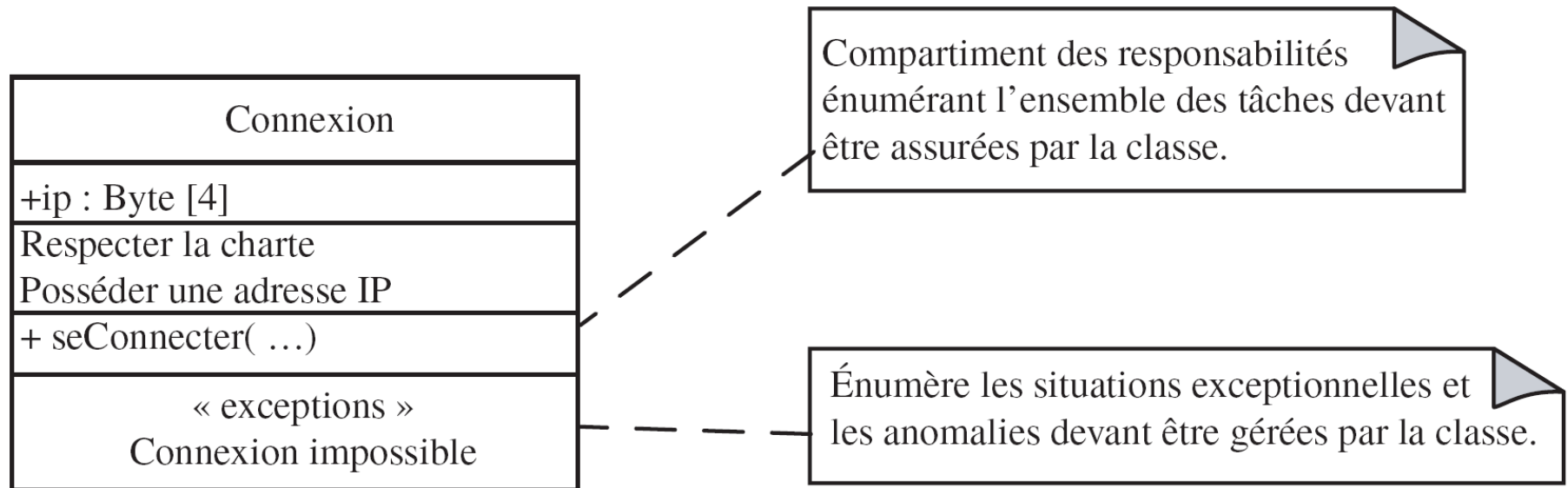
- Une opération décrit les types des paramètres et le type de la valeur de retour. La syntaxe de la déclaration est la suivante :

```
<modificateurAccès><nomMéthode  
([<paramètres>])>:[<valeurRenvoyée>]{<propriété>}
```

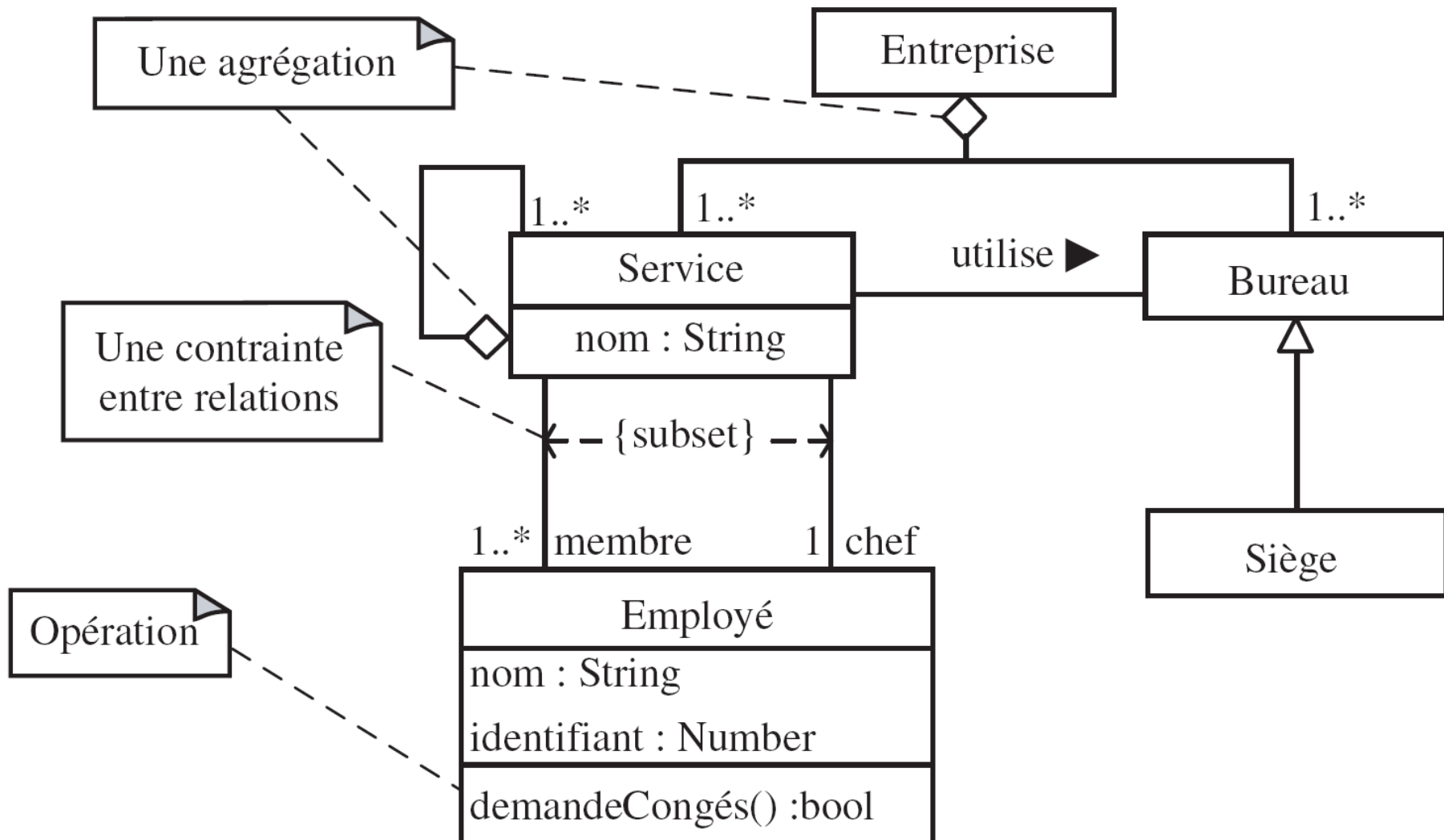
- La syntaxe de la liste des paramètres est la suivante :

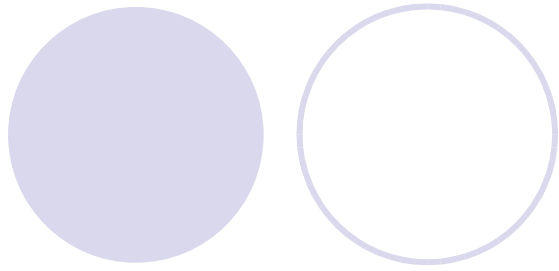
```
<nomClasse1><nomVariable1>,  
..., <nomClasseN><nomVariableN>
```

Compartiments supplémentaires

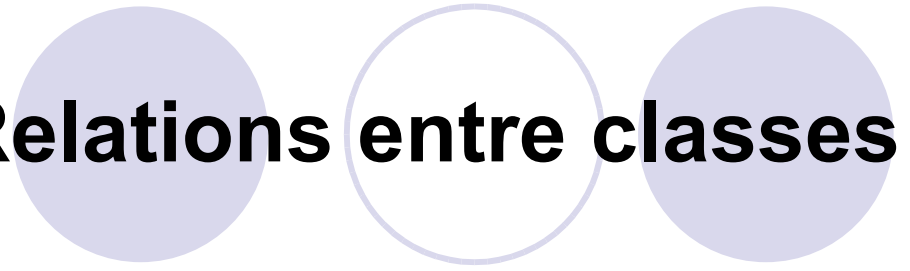


Exemple de diagramme des classes





Relations entre classes



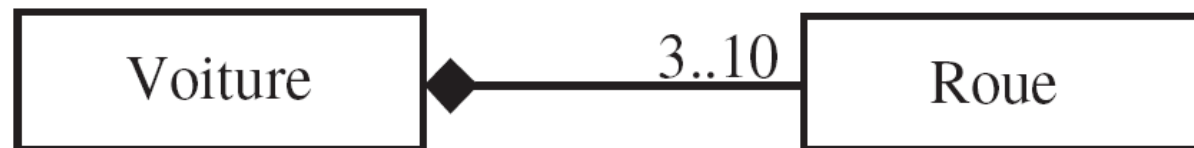
- Une relation d'*héritage* est une relation de généralisation/spécialisation permettant l'abstraction
- Une *dépendance* est une relation unidirectionnelle exprimant une dépendance sémantique entre les éléments du modèle.



- Une *association* représente une relation sémantique entre les objets d'une classe.
- Une relation d'*agrégation* décrit une relation de contenance ou de composition.

Multiplicités

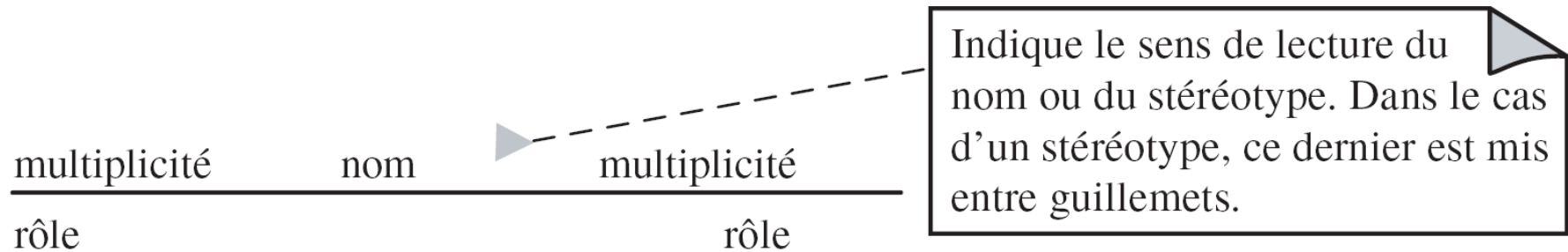
- Les relations expriment souvent les liens entre les objets.
- La notion de *multiplicité* permet le contrôle du nombre d'objets intervenant dans chaque instance de la relation.
 - *Exemple* : un objet de la classe Voiture est composé d'au moins trois roues et d'au plus dix roues.



- La syntaxe est `MultMin..MultMax`.
 - « * » à la place de `MultMax` signifie « plusieurs » sans préciser de nombre.
 - « n..n » se note aussi « n », et « 0..* » se note « * »

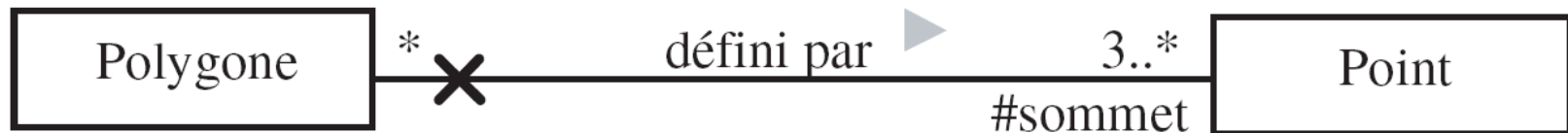
Association

- Une *association* est une relation structurelle entre objets.
 - Une association est souvent utilisée pour agréger les liens.
 - Elle est représentée par un trait entre classes.

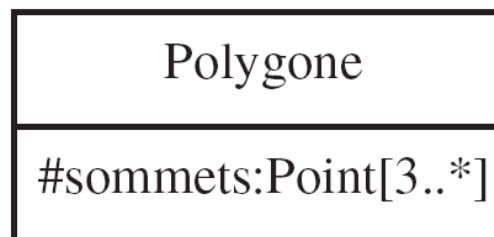


Navigabilité d'une association

- La *navigabilité* permet de spécifier dans quel(s) sens il est possible de traverser l'association à l'exécution.
- On interdit la navigation par une croix (X) du côté qui n'est pas atteignable

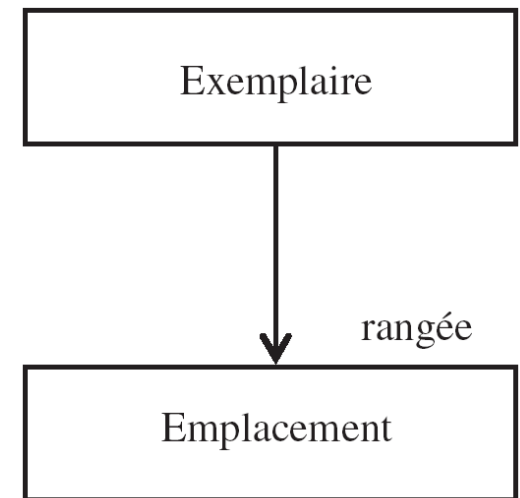


- On peut représenter les associations navigables dans un seul sens par des attributs. Les attributs sont toujours navigables.



Implémentation d'une association unidirectionnelle de 1 vers 1

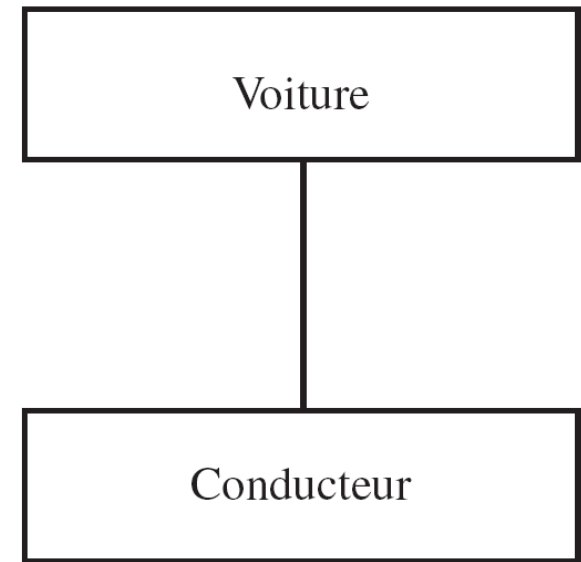
```
public class Emplacement{  
  
public class Exemplaire{  
    private Emplacement rangée;  
    public void setEmplacement(Emplacement  
        emplacement){this.rangée = emplacement;}  
    public Emplacement getEmplacement(){  
        return rangée;  
    }  
    public static void main( String [] argv ){  
        Emplacement emplacement = new Emplacement();  
        Exemplaire exemplaire = new Exemplaire();  
        exemplaire.setEmplacement( emplacement );  
        Emplacement place=exemplaire.getEmplacement();  
    }  
}
```



Implémentation d'une association bidirectionnelle de 1 vers 1

```
public class Conducteur{
    Voiture voiture;
    public void addVoiture( Voiture voiture ){
        if( voiture != null ){
            this.voiture = voiture;
            voiture.conducteur = this;
        }
    }
    public static void main( String [] argv ){
        Voiture voiture = new Voiture();
        Conducteur conducteur = new Conducteur();
        conducteur. addVoiture ( voiture );
    }
}

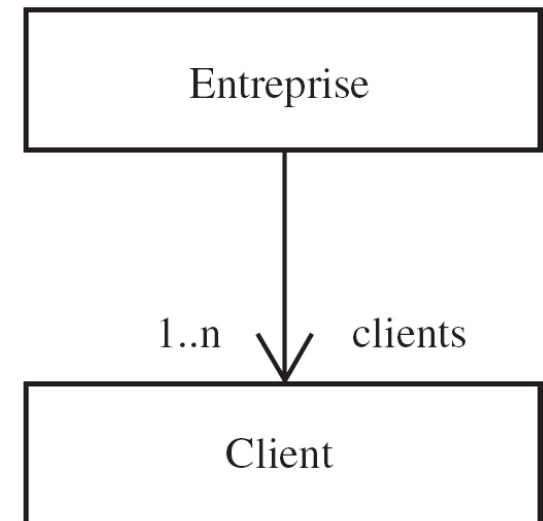
public class Voiture{
    Conducteur conducteur;
    public void addConducteur( Conducteur conducteur ){
        this.conducteur = conducteur;
        conducteur.voiture = this;
    }
}
```



Implémentation d'une association unidirectionnelle de 1 vers *

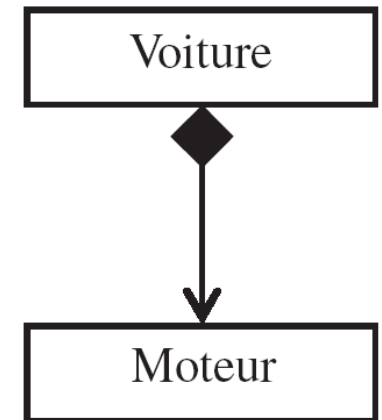
```
public class Client{

import java.util.Vector;
public class Entreprise{
    private Vector clients = new Vector();
    public void addClient( Client client ){
        clients.addElement( client );
    }
    public void removeClient( Client client ){
        clients.removeElement( client );
    }
    public static void main( String [] argv ){
        Entreprise monEntreprise = new Entreprise();
        Client client = new Client();
        monEntreprise.addClient( client );
        monEntreprise.removeClient( client );
    }
}
```



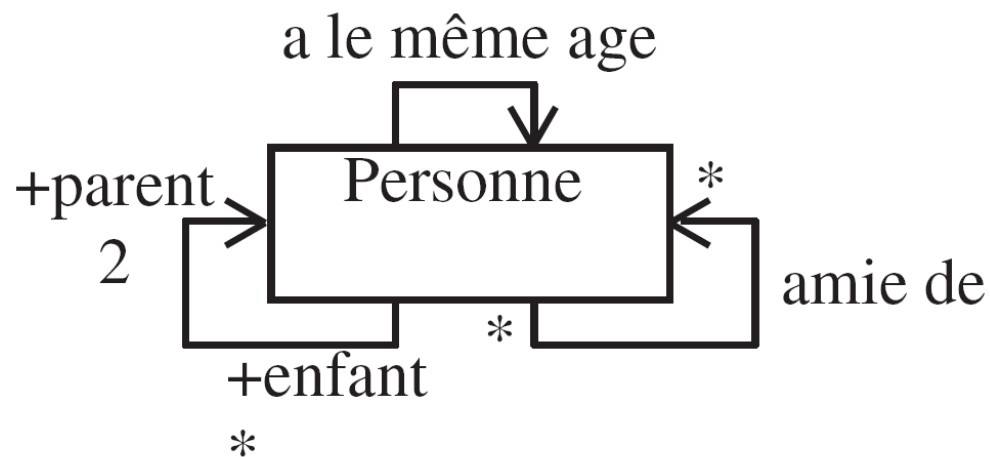
Implémentation d'une association bidirectionnelle de * vers 1

```
public class Voiture{  
    private class Moteur{}  
    private Moteur moteur;  
}
```



Associations réflexives

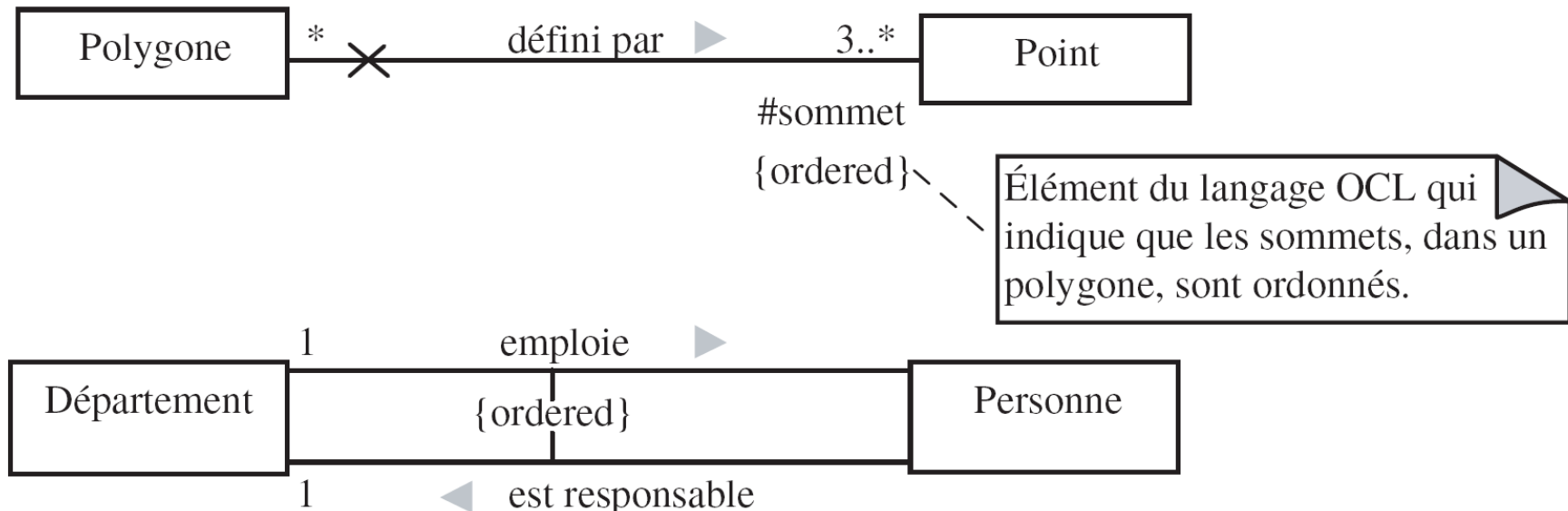
- L'association la plus utilisée est l'association binaire (reliant deux classeurs).
- Parfois, les deux extrémités de l'association pointent vers le même classeur. Dans ce cas, l'association est dite « *réflexive* ».



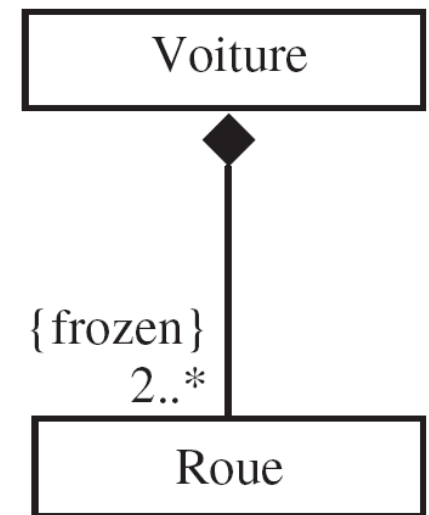
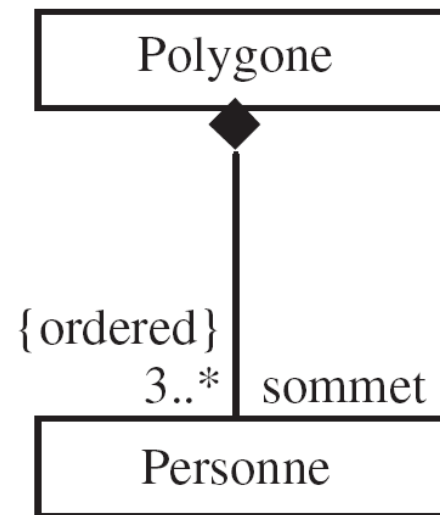
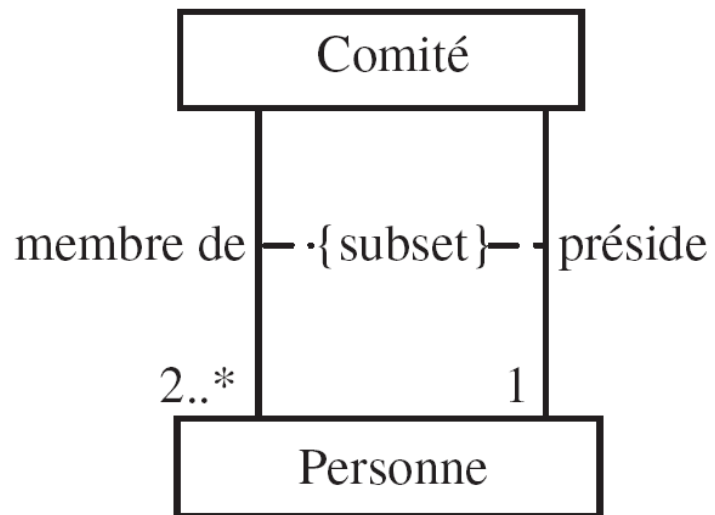
- Note : les flèches représentent des navigabilités permises

Association avec contraintes

- L'ajout de contraintes à une association ou bien entre associations apporte plus d'informations car cela permet de mieux préciser la portée et le sens de l'association.
- Les contraintes sont mises entre accolades et peuvent être exprimées dans n'importe quel langage mais on préférera OCL.

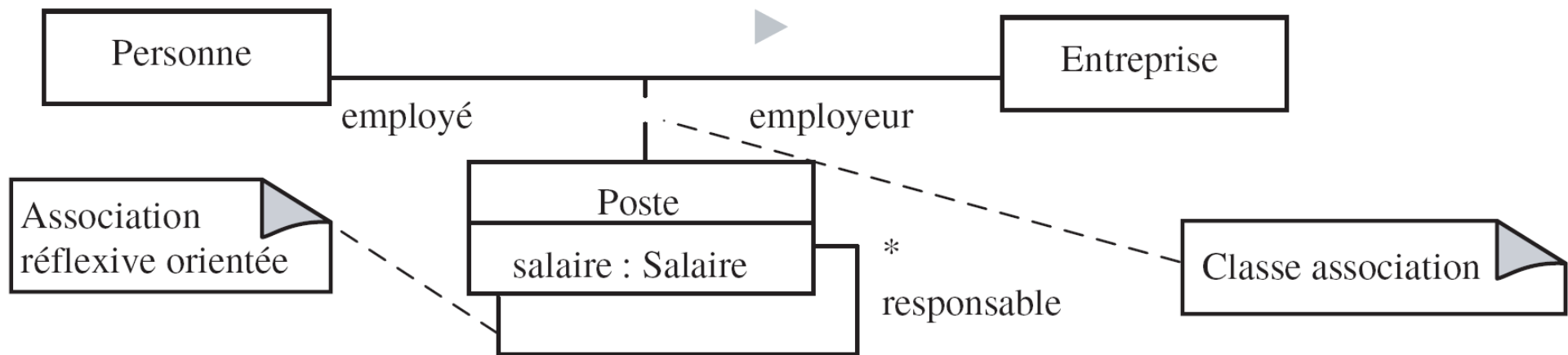


Associations avec contraintes



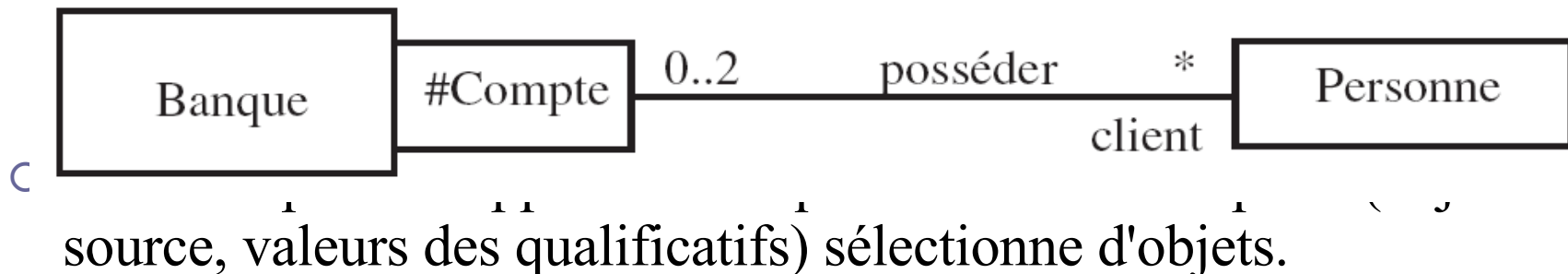
Classe-association

- Une association peut être raffinée et avoir ses propres attributs, qui ne sont disponibles dans aucune des classes qu'elle lie.
- Comme, dans le modèle objet, seules les classes peuvent avoir des attributs, cette association devient alors une classe appelée « *classe-association* ».



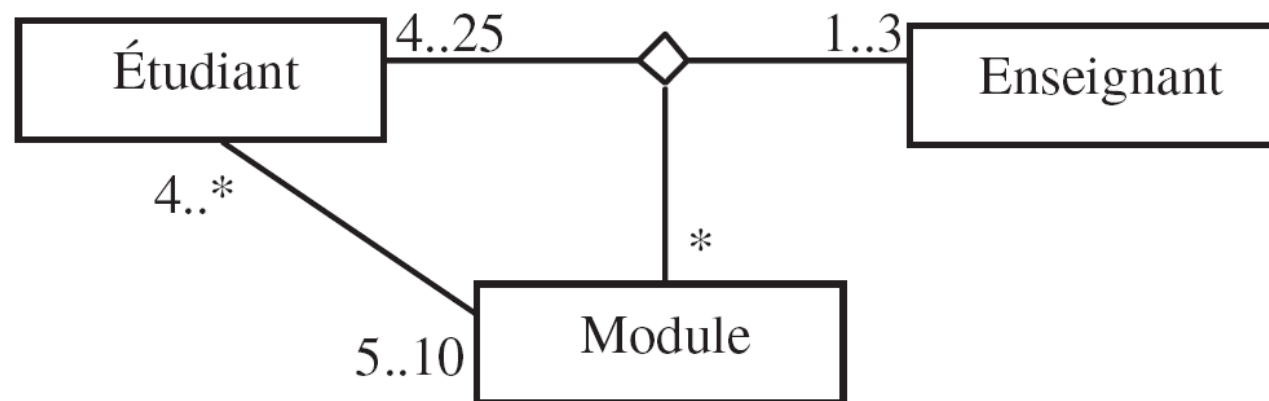
Qualificatif d'association

- Un *qualificatif d'association* est une liste d'attributs dans une association binaire où les valeurs des attributs sélectionnent un ou plusieurs objets liés.
 - Permet de limiter l'impact de l'*association qualifiée* sur les classes associées



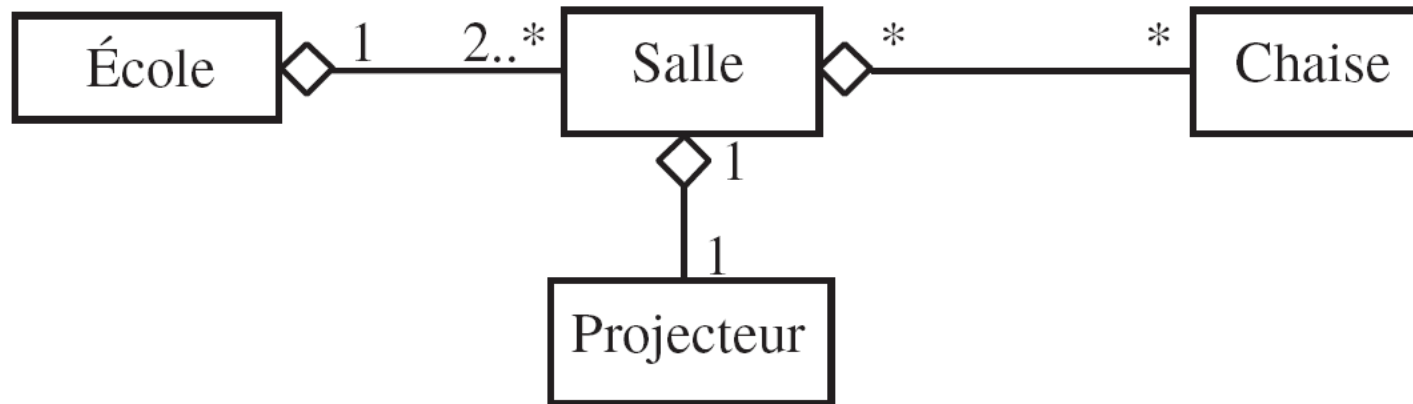
Associations n-aires

- Une association *n-aire* lie plus de deux classes.
 - Notation avec un losange central pouvant éventuellement accueillir une classe-association.
 - La multiplicité de chaque classe s'applique sur une instance du losange. Une instance du losange par rapport à une classe contient une unique instance de chacune des classes hormis la classe-association et la classe considérée.



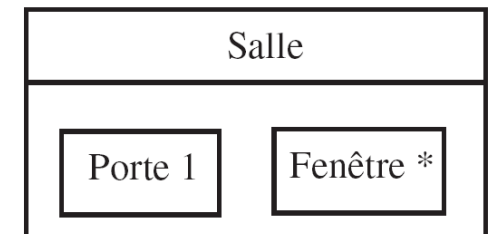
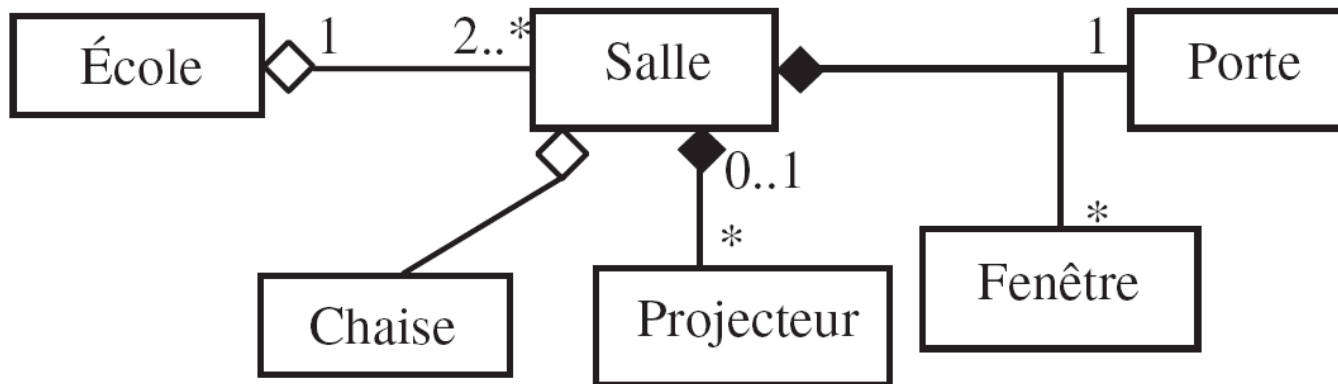
Relation d'agrégation

- Une *agrégation* est une forme particulière d'association. Elle représente la relation d'*inclusion* structurelle ou comportementale d'un élément dans un ensemble.
- Une agrégation se distingue d'une association par l'ajout d'un losange vide du côté de l'agrégat.



Relation de composition

- La relation de *composition* décrit une *contenance* structurelle entre instances. On utilise un losange plein.
 - La destruction ou la copie de l'objet composite implique respectivement la destruction ou la copie de ses composants.
 - Une instance de la partie appartient toujours à au plus une instance de l'élément composite.





Composition et agrégation

- Dès lors que l'on a une relation du tout à sa partie, on a une relation d'agrégation. La composition est aussi dite « *agrégation forte* ».
- Pour décider de mettre une composition plutôt qu'une agrégation, on doit se poser les questions suivantes :
 - Est-ce que la destruction de l'objet composite (du tout) implique nécessairement la destruction des objets composants (les parties) ? C'est le cas si les composants n'ont pas de raison d'être autonomes vis à vis des composites.
 - Lorsque l'on copie le composite, doit-on aussi copier les composants, ou est-ce qu'on peut les « réutiliser », auquel cas un composant peut faire partie de plusieurs composites ?
- Si on répond par l'affirmative, on doit considérer une composition.



Relation d'héritage, propriétés

- La classe enfant possède toutes les propriétés de ses classes parents. Mais elle n'a pas accès aux propriétés privées de celles-ci.
- Une classe enfant peut *redéfinir* (même signature) une ou plusieurs méthodes de la classe parent.
 - Sauf indications contraires, un objet utilise les opérations les plus spécialisées dans la hiérarchie des classes.
 - La *surcharge d'opérations* (même nom, mais signatures des opérations différentes) est possible dans toutes les classes.

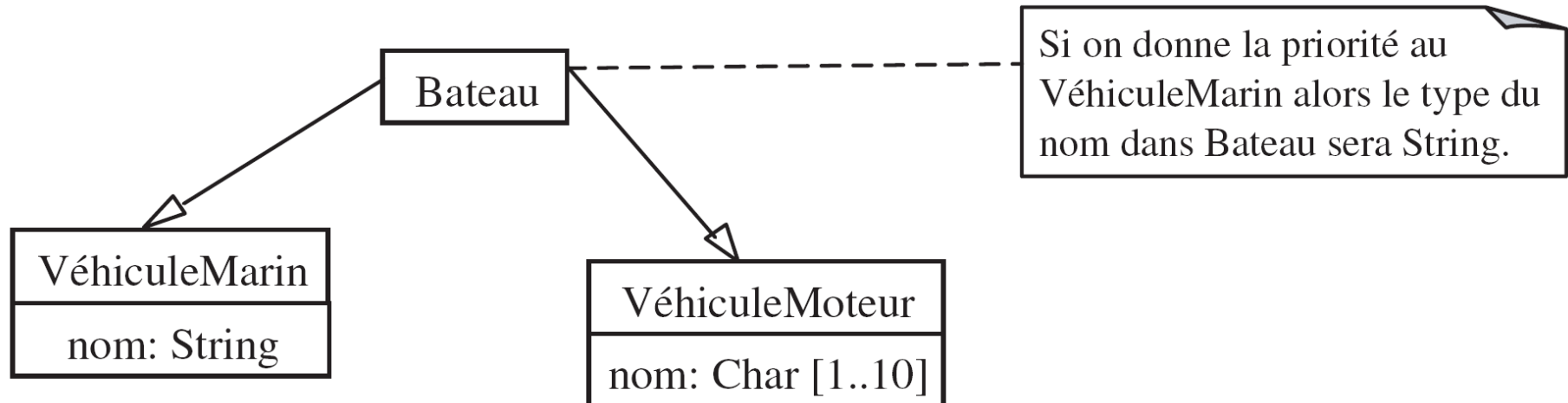


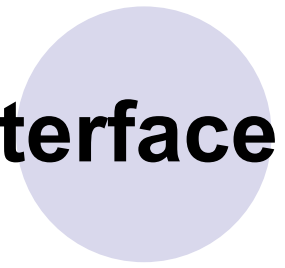
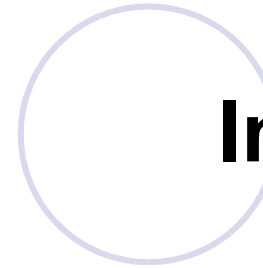
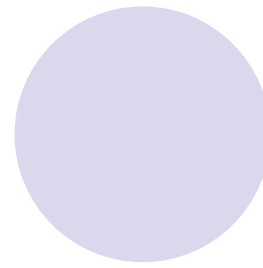
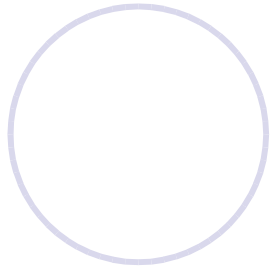
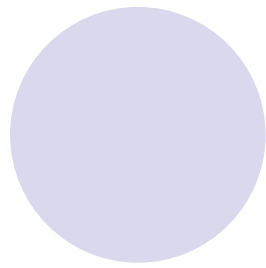
Relation d'héritage, propriétés

- Toutes les associations de la classe parent s'appliquent, par défaut, aux classes dérivées.
- Une instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue (principe de la *substitution*).
 - Par exemple, toute opération acceptant un objet d'une classe Animal doit accepter tout objet de la classe Chat (l'inverse n'est pas toujours vrai).

Héritage multiple

- Une classe peut avoir plusieurs classes parents. On parle alors d'héritage multiple.
 - Le langage C++ est un des langages objet permettant son implémentation effective.
 - Java ne le permet pas.

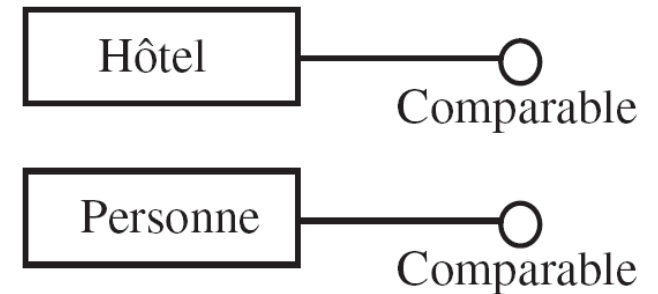
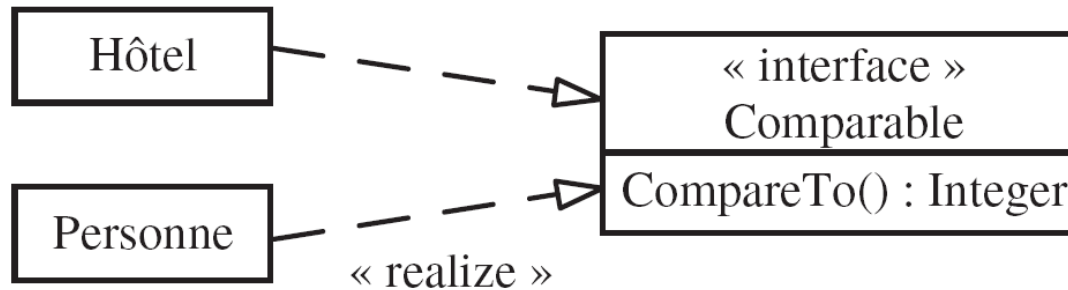
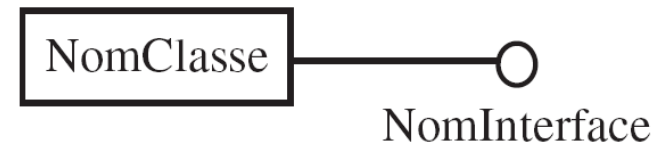
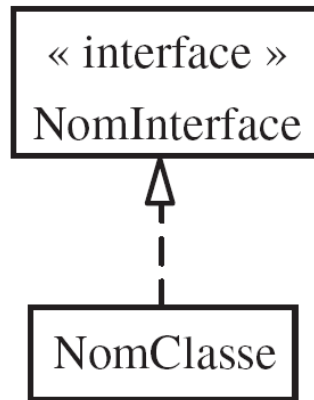




Interface

- Le rôle d'une *interface* est de regrouper un ensemble d'opérations assurant un service cohérent offert par un classeur et une classe en particulier.
- Une interface est définie comme une classe, avec les mêmes compartiments. Les principales différences sont
 - la non-utilisation du mot-clé « abstract », car l'interface et toutes ses méthodes sont, par définition, abstraites ;
 - l'ajout du stéréotype « interface » avant le nom de l'interface.

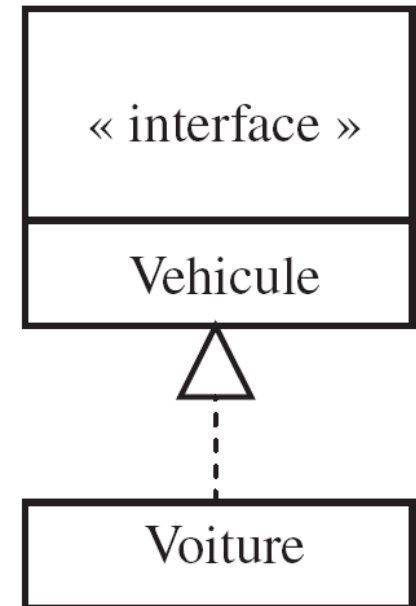
Exemples d'interfaces

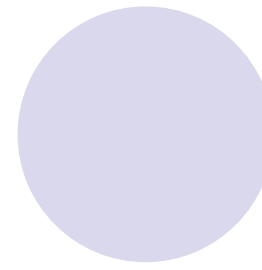
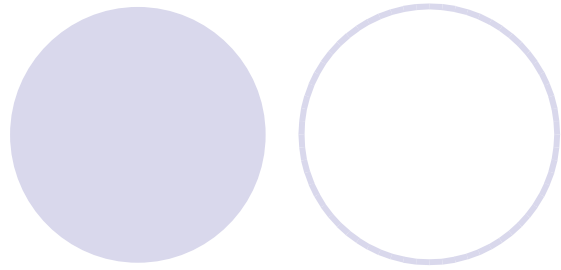


Implémentation d'une interface

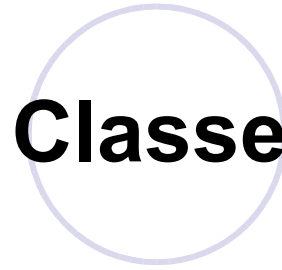
```
public interface Vehicule{  
    public void seDeplacer();  
}
```

```
public class Voiture implements Vehicule{  
    public void seDeplacer(){  
        // ...  
    }  
}
```

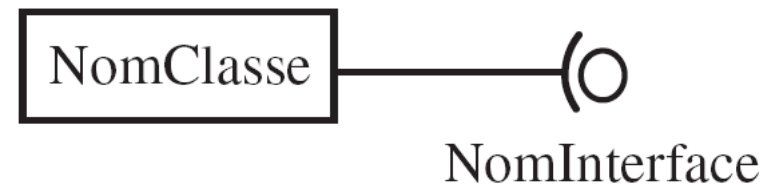
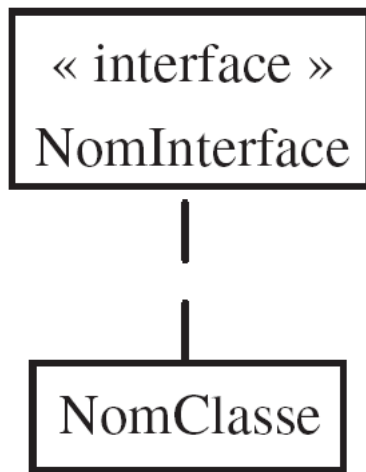


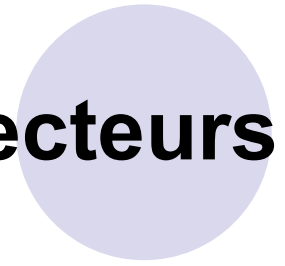
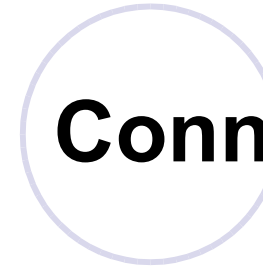
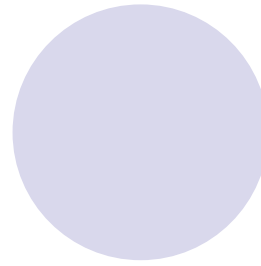
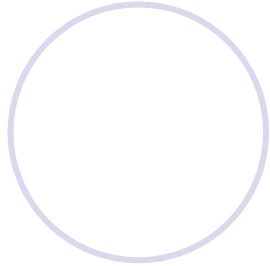
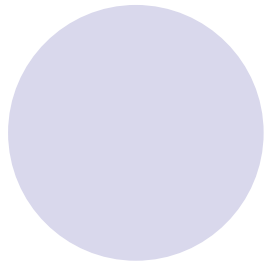


Classe cliente



- Quand une classe dépend d'une interface (*interface requise*) pour réaliser ses opérations, elle est dite « *classe cliente de l'interface* ».
- Graphiquement, cela est représenté par une relation de dépendance entre la classe et l'interface. Une notation *lollipop* équivalente est aussi possible.



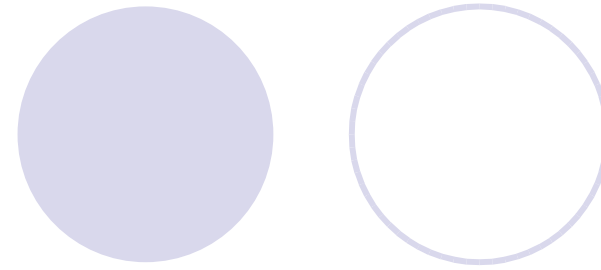
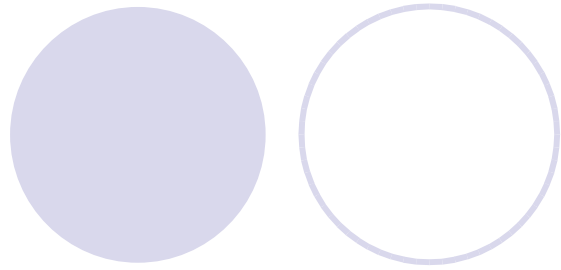


Connecteurs

- Les connecteurs spécifient les liens de communication.
 - Ces liens peuvent correspondre à
 - des instances d'association ou
 - à des communications via des passages de paramètres, des appels d'opération, etc.
- Une connexion peut être réalisée par un simple pointeur ou par un réseau de communication complexe.



connecteur



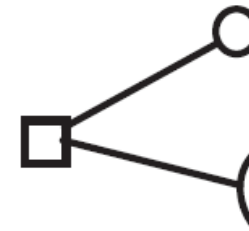
- Le *port* est une propriété d'un classeur lui permettant de préciser ses points d'interaction avec son environnement.
 - Les ports sont connectés via des *connecteurs*. Ils peuvent spécifier les services d'un classeur ou les services qu'il attend de son environnement. Deux types de ports existent :
 - le port protocole, qui décrit les connectiques internes et externes d'un classeur (souvent classe active),
 - le port comportemental, qui est directement associé à la machine d'états (automate) du classeur qui porte ce port.



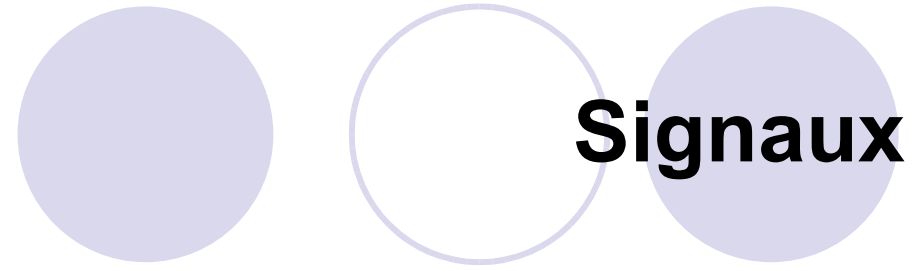
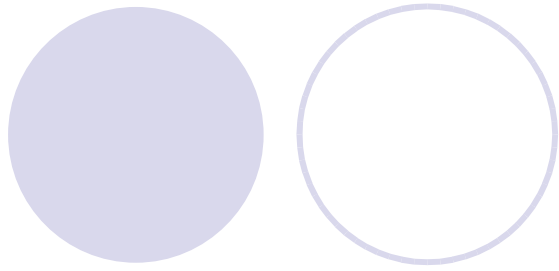
port



Port comportemental



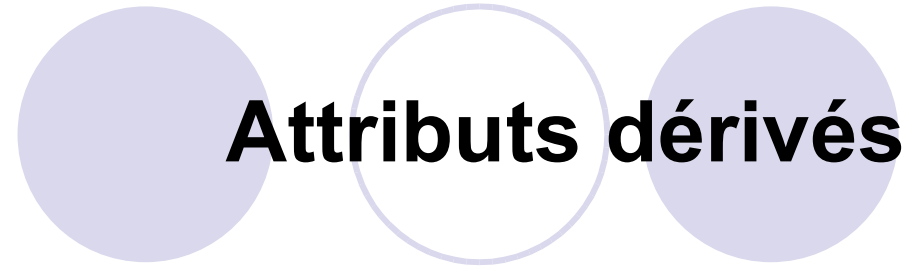
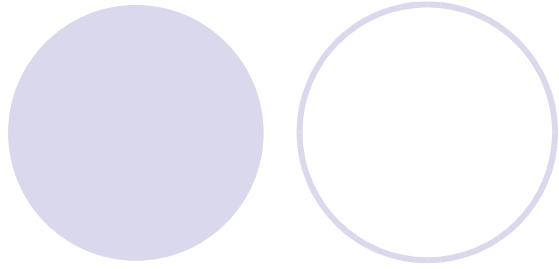
Port protocole



- Un *signal* modélise un *message asynchrone* échangé entre les instances.
- Il est caractérisé par un nom et un ensemble d'attributs qui correspondent aux données transportées par le signal.



Signal

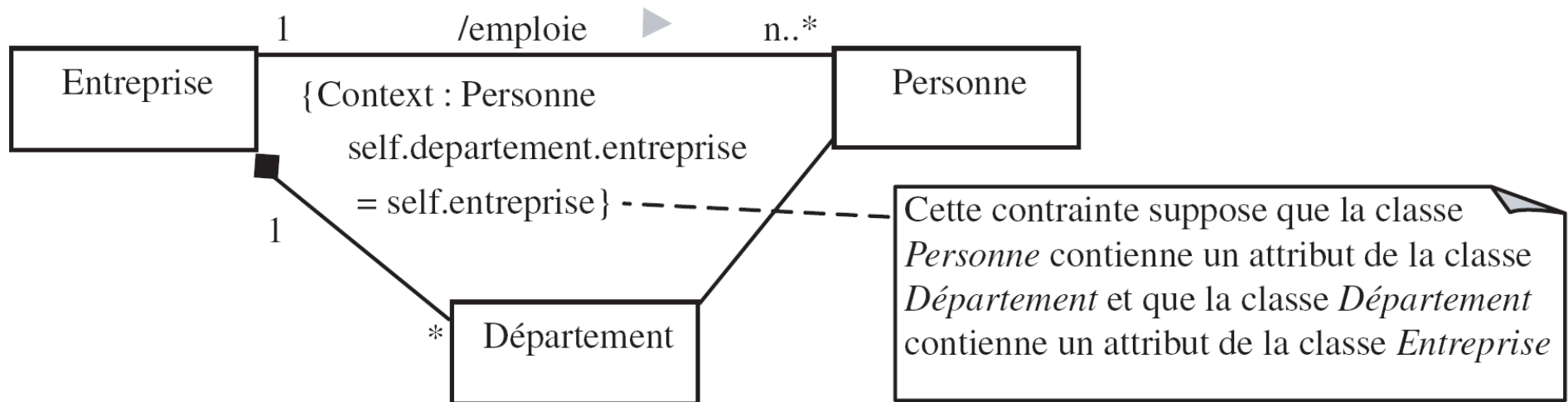


Attributs dérivés

- Les *attributs dérivés* peuvent être calculés à partir d'autres attributs et des formules de calcul.
- Les attributs dérivés sont symbolisés par l'ajout d'un « / » devant leur nom.
- Lors de la conception, un attribut dérivé peut être utilisé comme marqueur jusqu'à ce que vous puissiez déterminer les règles à lui appliquer.

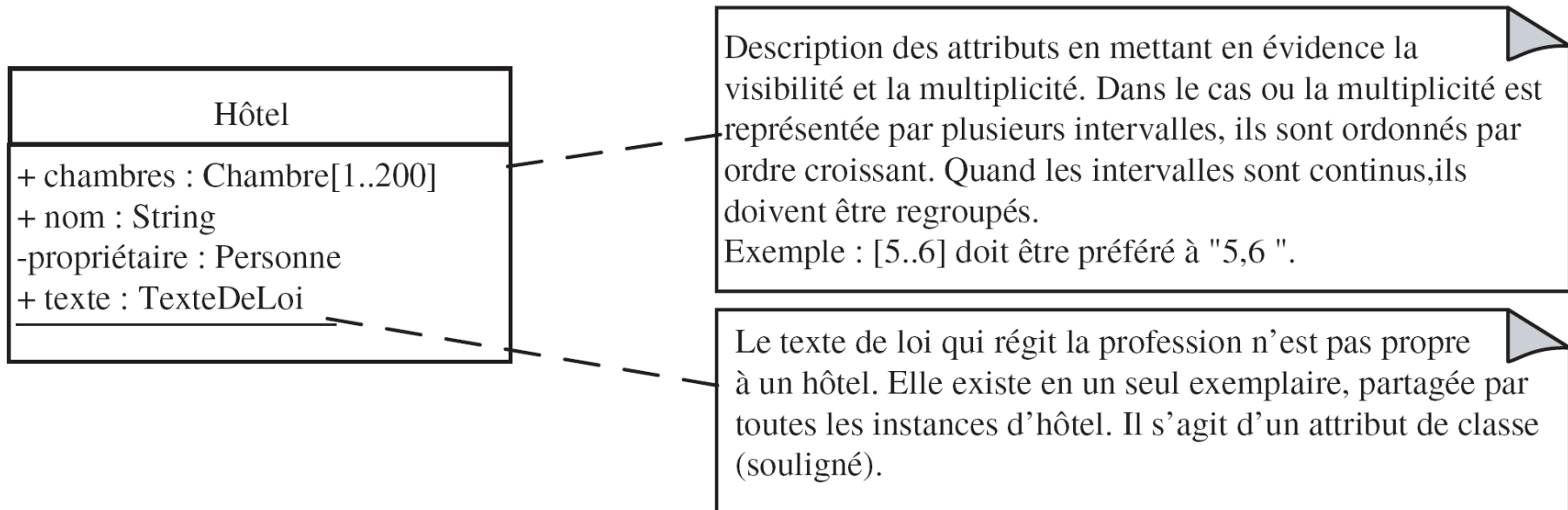
Association dérivée

- Une association *dérivée* est conditionnée ou peut être déduite à partir d'une autre association.
 - On utilise le symbole « / ».
 - Souvent un ensemble de contraintes, exprimées en OCL, est ajouté à une association dérivée pour définir les conditions de dérivation.



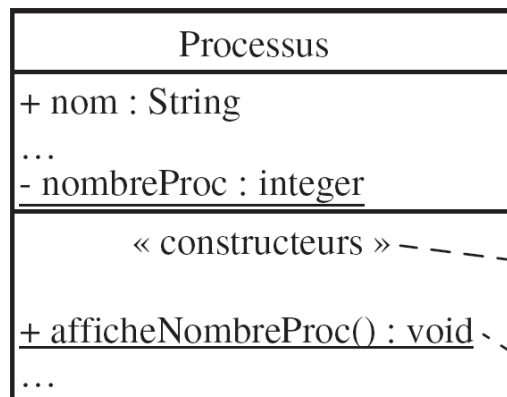
Attributs de classe

- Par défaut, les valeurs des attributs définis dans une classe diffèrent d'un objet à un autre. Parfois, il est nécessaire de définir un *attribut de classe* qui garde une valeur unique et partagée par toutes les instances.
- Graphiquement, un attribut de classe est souligné



Opérations de classe

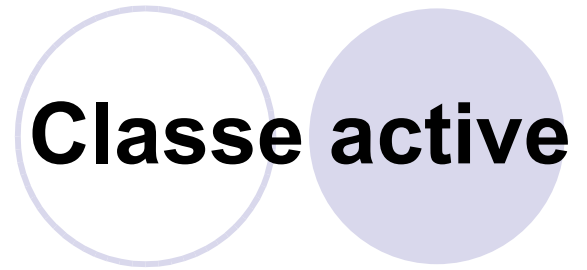
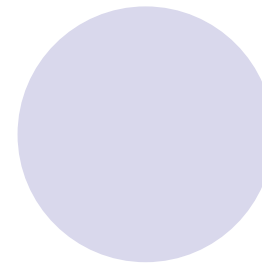
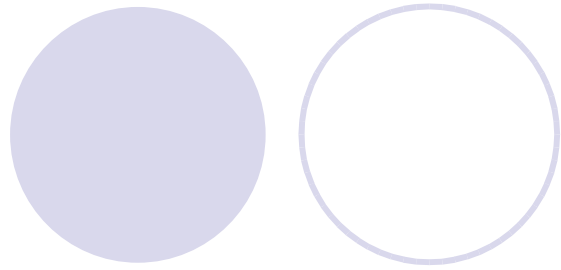
- Semblable aux attributs de classe
 - Une *opération de classe* est une propriété de la classe, et non de ses instances.
 - Elle n'a pas accès aux attributs des objets de la classe.



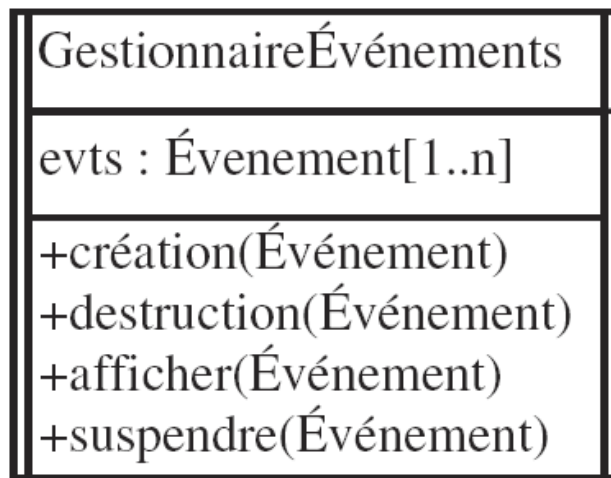
Le nombreProc est un attribut de classe privé. Contrairement au nom qui est associé à chaque processus, le nombreProc n'est disponible qu'en un seul exemplaire. Il est associé à la classe. Les « ... » indiquent que tous les attributs ne sont pas représentés.

Le stéréotype type de méthodes est utilisé pour simplifier la notation ou lorsque les opérations ou leurs signatures ne sont pas encore connues,

La méthode affiche() est une méthode de classe. Elle est publique et ne renvoie aucune valeur.



- Une classe passive (par défaut) sauvegarde les données et offre les services aux autres.
- Une *classe active* initie et contrôle le flux d'activités
 - Graphiquement, une classe active est représentée comme une classe standard, avec une épaisseur de trait plus prononcée.



L'épaisseur du trait indique qu'il s'agit d'une classe active.

Diagrammes de classes à différents niveaux

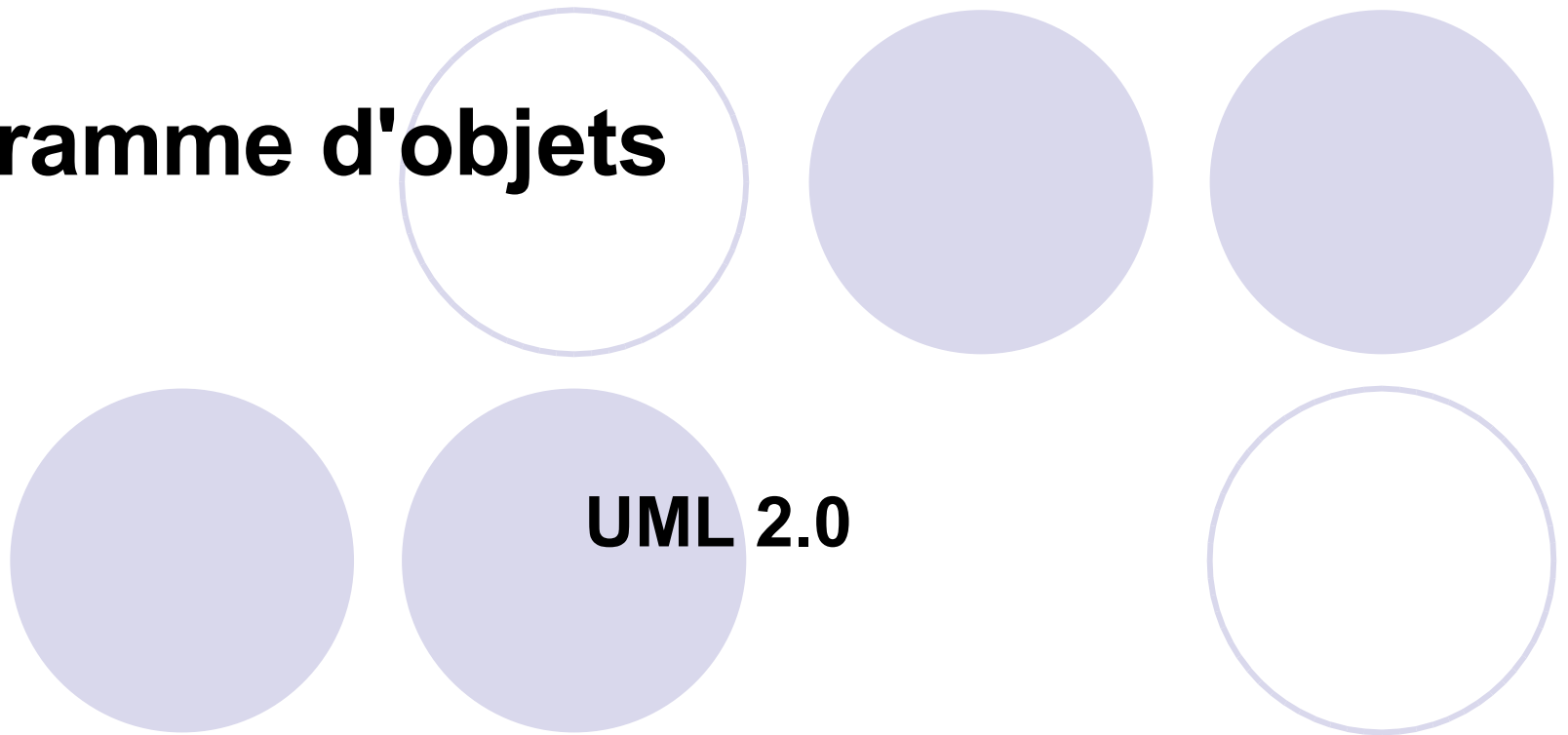
- On peut utiliser les diagrammes de classes pour représenter un système à différents niveaux d'abstraction :
 - Le point de vue *spécification* met l'accent sur les interfaces des classes plutôt que sur leurs contenus.
 - Le point de vue *conceptuel* capture les concepts du domaine et les liens qui les lient. Il s'intéresse peu ou prou à la manière éventuelle d'implémenter ces concepts et relations et aux langages d'implémentation.
 - Le point de vue *implémentation*, le plus courant, détaille le contenu et l'implémentation de chaque classe.
- On enrichit les modèles à mesure que l'on avance vers l'implémentation



Construction d'un diagramme de classes

1. Trouver les classes du domaine étudié.
 - Souvent, concepts et substantifs du domaine.
2. Trouver les associations entre classes.
 - Souvent, verbes mettant en relation plusieurs classes.
3. Trouver les attributs des classes.
 - Souvent, substantifs correspondant à un niveau de granularité plus fin que les classes. Les adjectifs et les valeurs correspondent souvent à des valeurs d'attributs.
4. Organiser et simplifier le modèle en utilisant l'héritage
5. Tester les chemins d'accès aux classées
6. Itérer et raffiner le modèle.

Diagramme d'objets

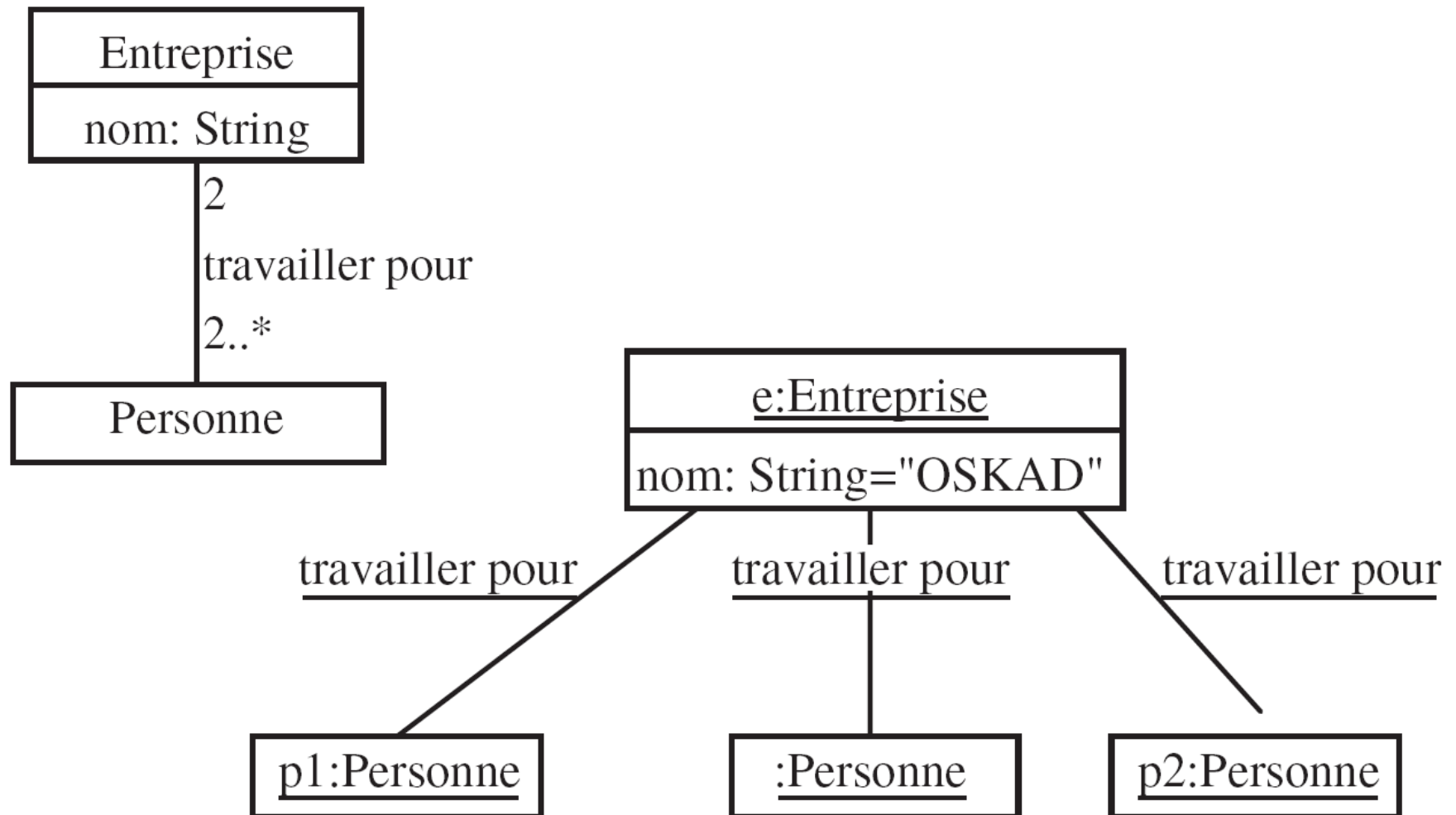




Objectif

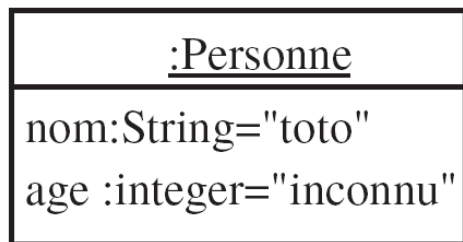
- Le *diagramme d'objets* représente les objets d'un système à un instant donné. Il permet :
 - d'illustrer le modèle de classes (en montrant un exemple qui explique le modèle)
 - de préciser certains aspects du système (en mettant en évidence des détails imperceptibles dans le diagramme de classes),
 - d'exprimer une exception (en modélisant des cas particuliers, des connaissances non généralisables...)
- Le diagramme de classes modélise les règles et le diagramme d'objets modélise des faits.

Diagramme de classes et diagramme d'objets



Représentation des objets

- Comme les classes, on utilise des cadres compartimentés.
- Les noms des objets sont par contre soulignés et on peut rajouter son identifiant devant le nom de sa classe.
- Les valeurs (a) ou l'état (f) d'un objet peuvent être spécifiées.
- Les instances peuvent être anonymes (a,c,d), nommées (b,f), orphelines (e), multiples (d) ou stéréotypées (g).



(a)



(b)



(c)



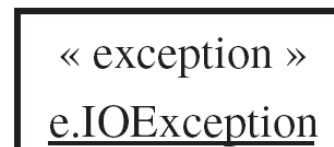
(d)



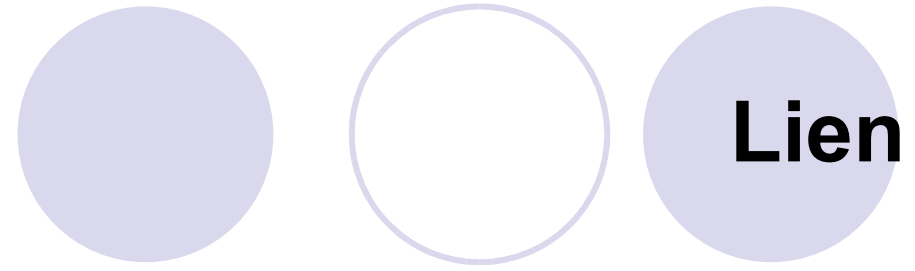
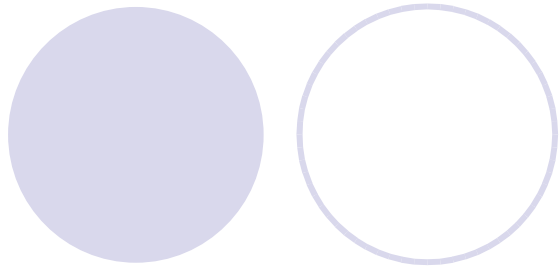
(e)



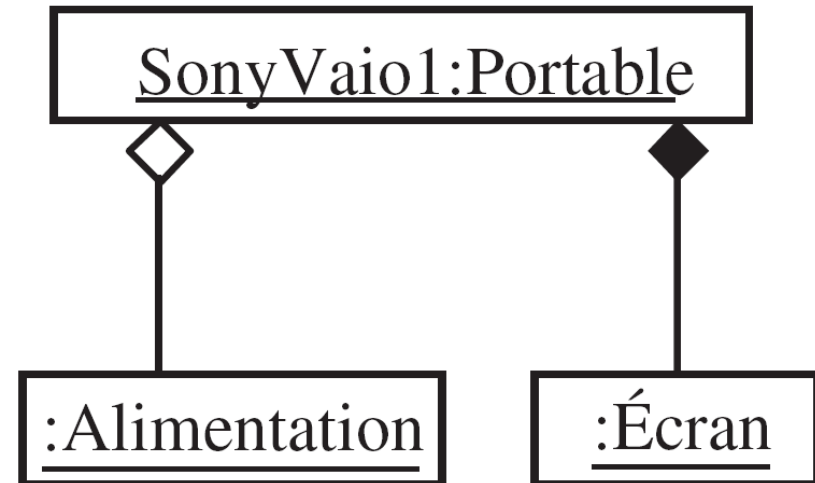
(f)



(g)

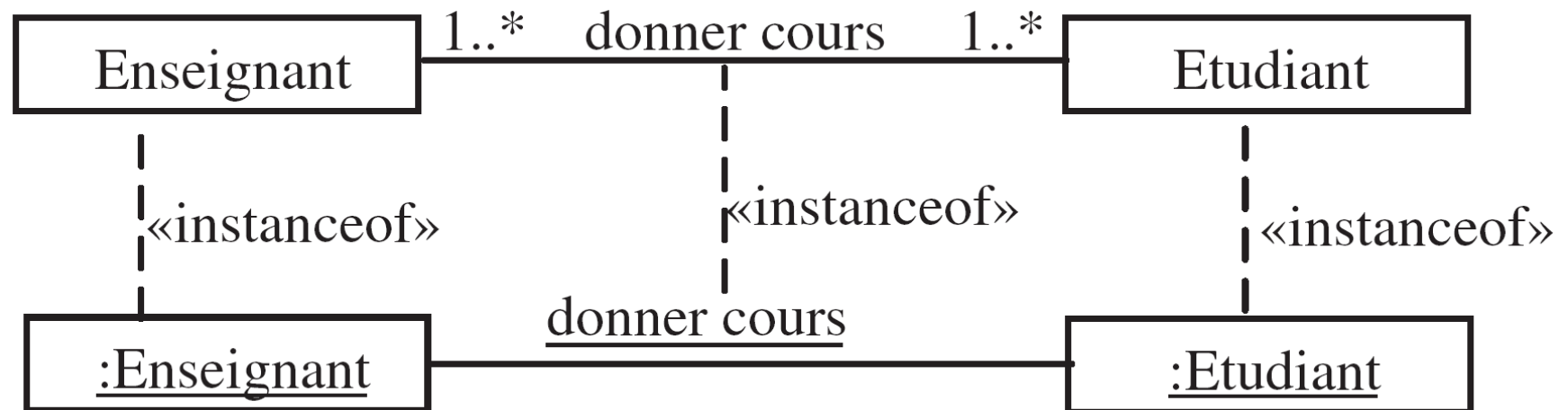


- Un *lien* est une instance d'une association
- Un lien se représente comme une association mais s'il a un nom, il est souligné.
- Naturellement, on ne représente pas les multiplicités



Relation de dépendance d'instanciation

- La relation de dépendance d'instanciation (stéréotypée) décrit la relation entre un classeur et ses instances.
- Elle relie, en particulier, les associations aux liens et les classes aux objets



Diagrammes d'interaction

Séquence et communication

The logo for UML 2.0 consists of five light purple circles arranged in a pentagonal pattern. The top circle is empty, while the other four are filled. The text 'UML 2.0' is centered within the bottom-most filled circle.

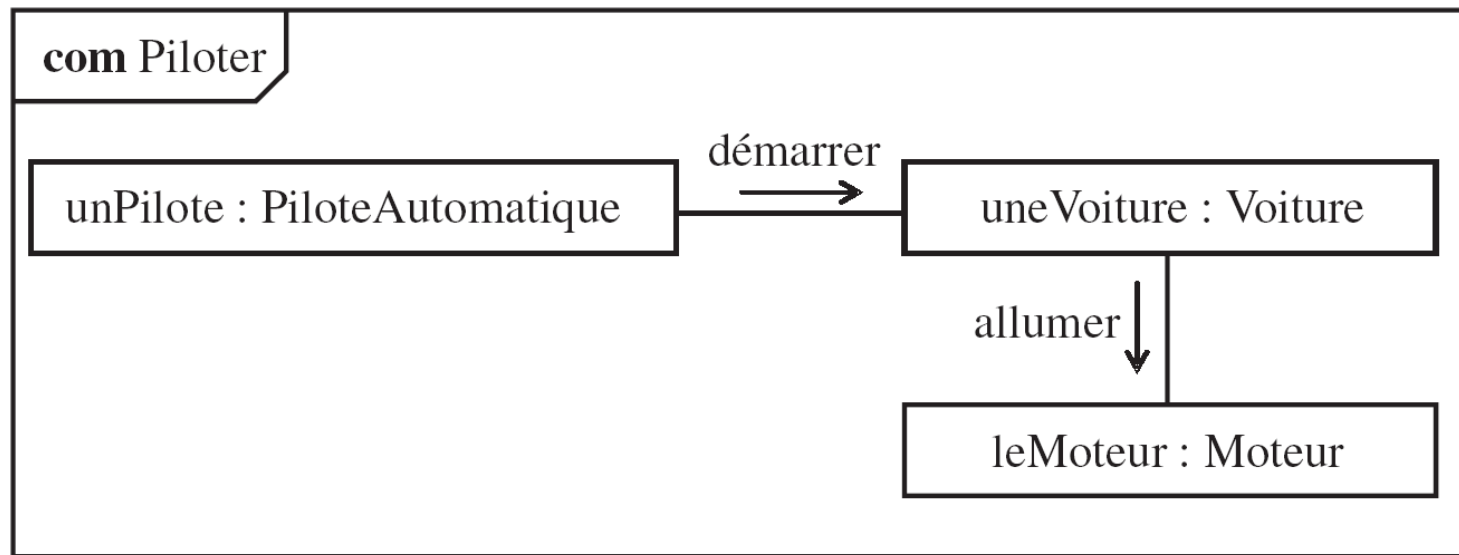
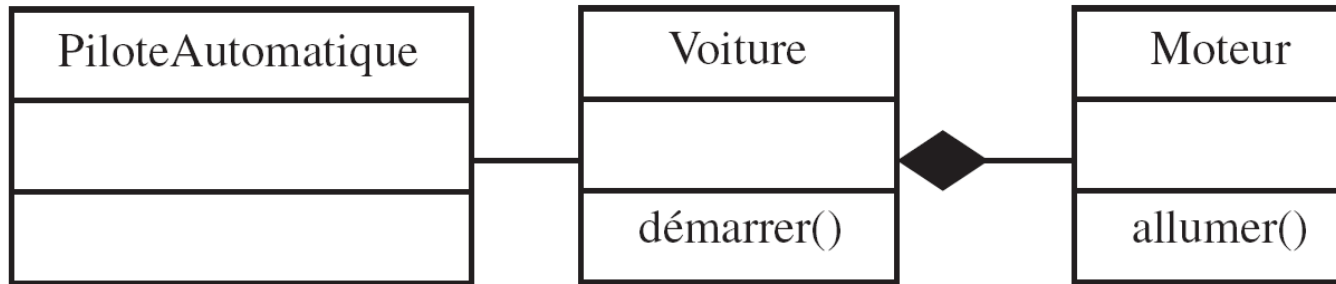
UML 2.0

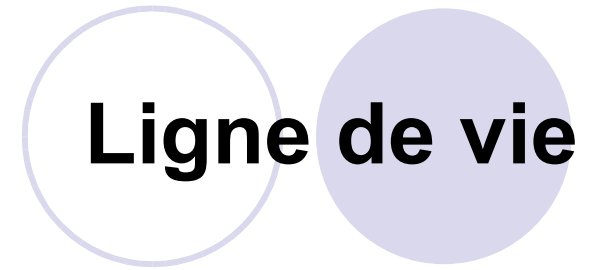
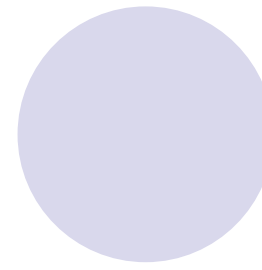
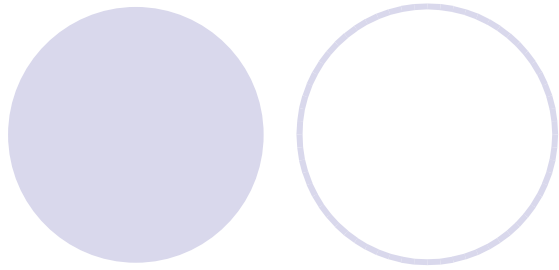


Objectif des diagrammes d'interaction

- Les diagrammes de cas d'utilisation listent des interactions avec des acteurs en spécifiant les grandes fonctions d'un système.
- Les diagrammes d'*interaction* permettent de décrire *comment* les éléments d'un système et les acteurs interagissent.
- Les interactions ne se limitent pas aux acteurs.
 - Par exemple, des objets au coeur d'un système, instances de classes, interagissent lorsqu'ils s'échangent des messages.
- Une *interaction* se focalise sur l'échange d'informations entre différents classeurs.
 - On peut utiliser ces diagrammes à différents niveaux d'abstraction.

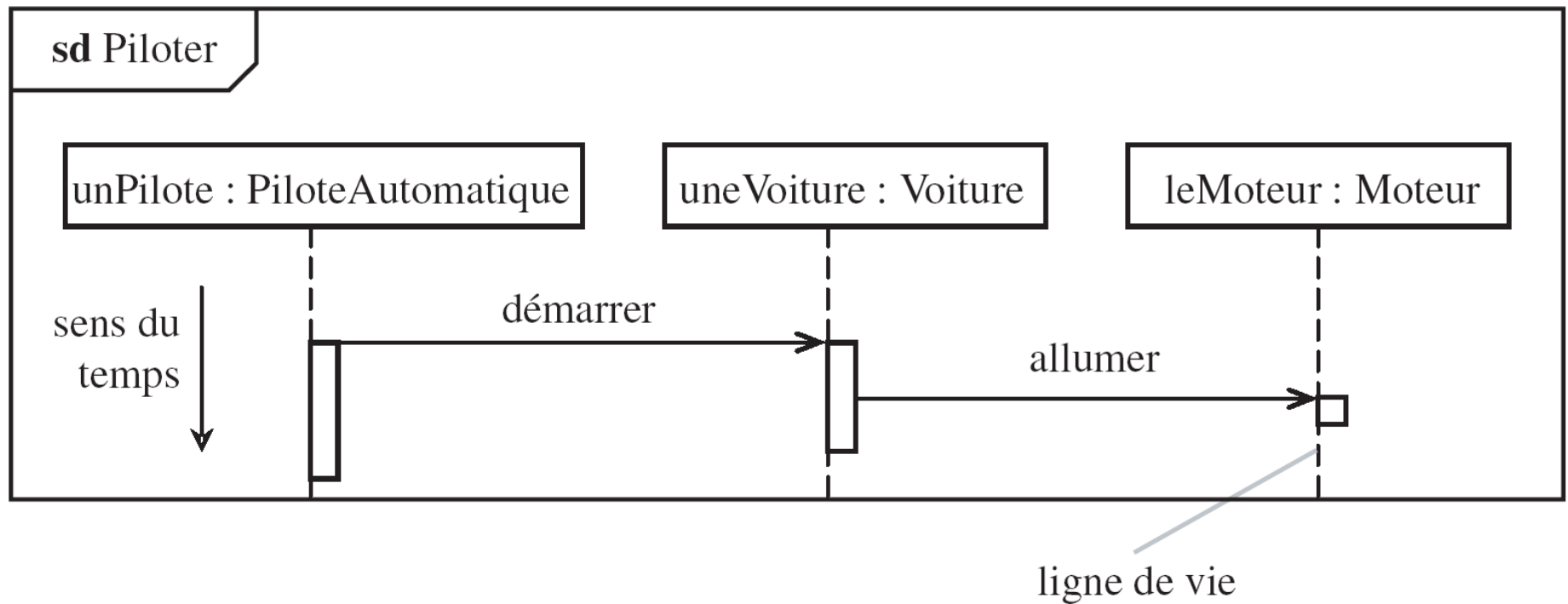
Exemple d'interaction





Ligne de vie

- Les participants à une interaction sont appelés « *lignes de vie* ».
 - Une ligne de vie représente un participant unique à une interaction.





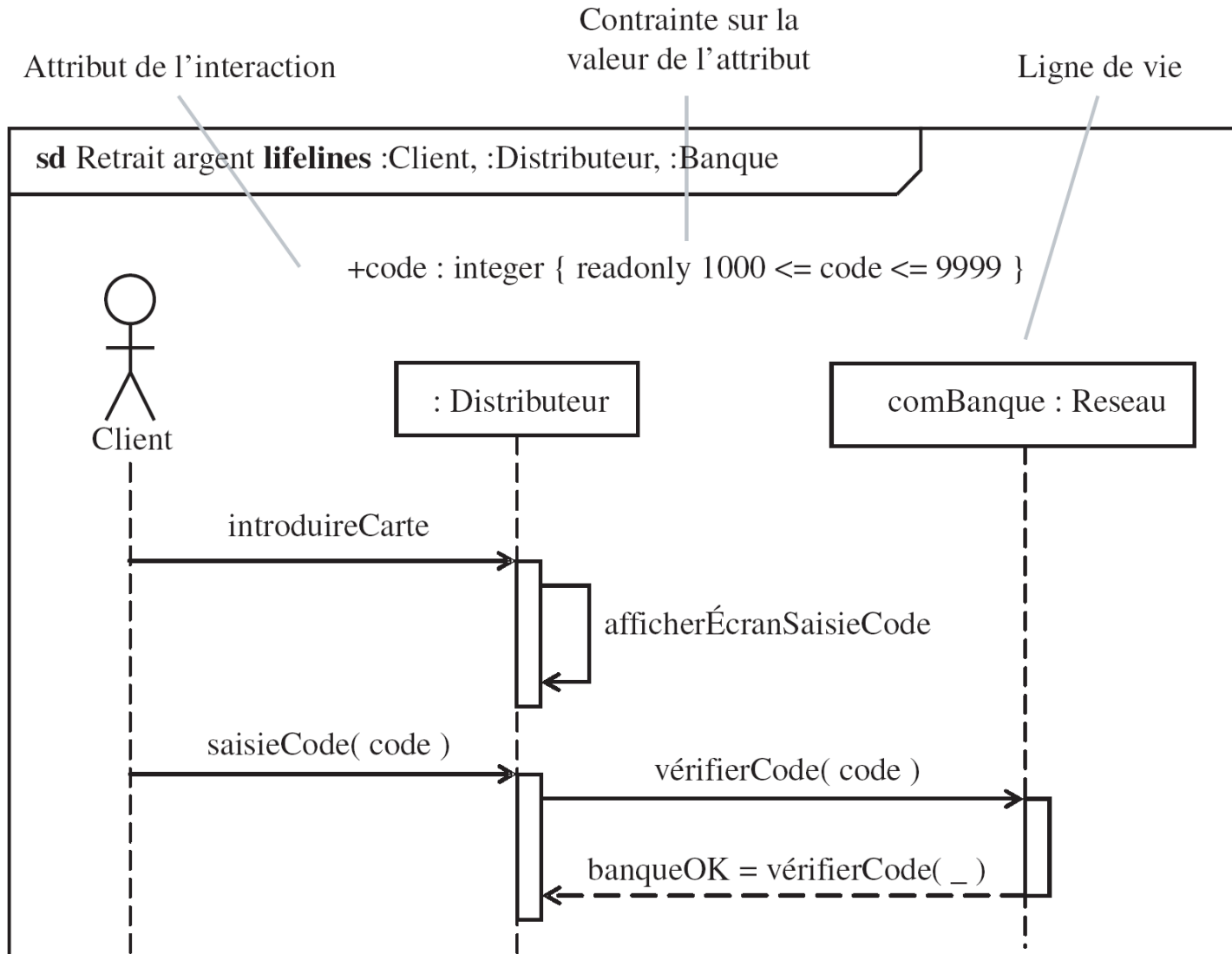
Diagrammes de séquence et de communication

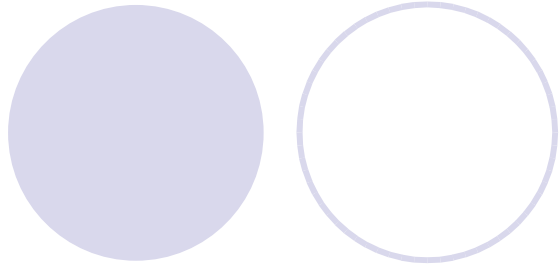
- Les diagrammes de communication et de séquence représentent des interactions entre des lignes de vie.
 - Un *diagramme de séquence* montre des interactions sous un angle temporel, en mettant l'emphasis sur le séquençement temporel de messages échangés entre des lignes de vie
 - Un *diagramme de communication* montre une représentation spatiale des lignes de vie.
- A ces diagrammes, UML 2.0 en ajoute un troisième : le *diagramme de timing*.
 - Son usage est limité à la modélisation des systèmes qui s'exécutent sous de fortes contraintes de temps, comme les systèmes temps réel.

Syntaxe des diagrammes d'interaction

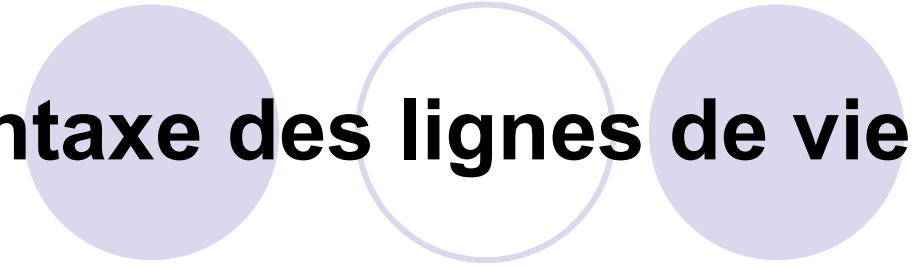
- Un diagramme d'interaction se représente par un rectangle contenant, dans le coin supérieur gauche, un pentagone accompagné du mot-clé
 - « `sd` » lorsqu'il s'agit d'un diagramme de séquence
 - « `com` » lorsqu'il s'agit d'un diagramme de communication.
- Le mot clé est suivi du *nom de l'interaction*
- Dans le pentagone, on peut aussi faire suivre le nom par la liste des *lignes de vie* impliquées, précédée par le mot clé « `lifelines` : »
- Des attributs peuvent être indiqués près du sommet du rectangle contenant le diagramme.

Exemple de diagramme d'interaction





Syntaxe des lignes de vie



- Une ligne de vie se représente par un rectangle auquel est accrochée une ligne verticale pointillée.
- Le rectangle contient un identifiant dont la syntaxe est :
`[<nomLigneDeVie>['['sélecteur']']] :<nomClasseur>`
- Le sélecteur permet de choisir un objet parmi n (par exemple objet[2]).

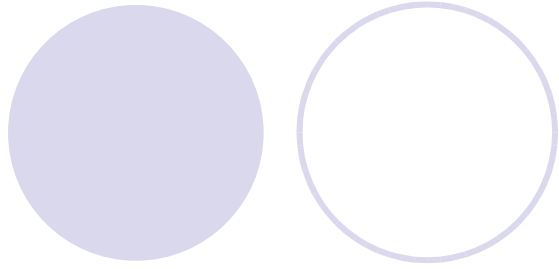


Diagramme de séquences



- Les principales informations contenues dans un diagramme de séquence sont les *messages* échangés entre les lignes de vie, présentés dans un ordre chronologique.
 - Un message définit une communication particulière entre des lignes de vie.
 - Plusieurs types de messages existent, dont les plus courants :
 - l'envoi d'un signal ;
 - l'invocation d'une opération ;
 - la création ou la destruction d'une instance.

Principaux types de messages

- Un message *synchrone* bloque l'expéditeur jusqu'à prise en compte du message par le destinataire. Le flot de contrôle passe de l'émetteur au récepteur.
 - Typiquement : envoi de signal

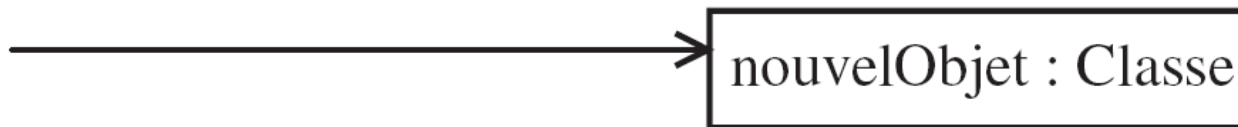


- Un message *asynchrone* n'interrompt pas l'exécution de l'expéditeur. Le message envoyé peut être pris en compte par le récepteur à tout moment ou ignoré.
 - Typiquement : appel de méthode



Création et destruction de lignes de vie

- La *création* d'un objet est matérialisée par une flèche qui pointe sur le sommet d'une ligne de vie.
 - On peut aussi utiliser un message normal portant le nom « create »

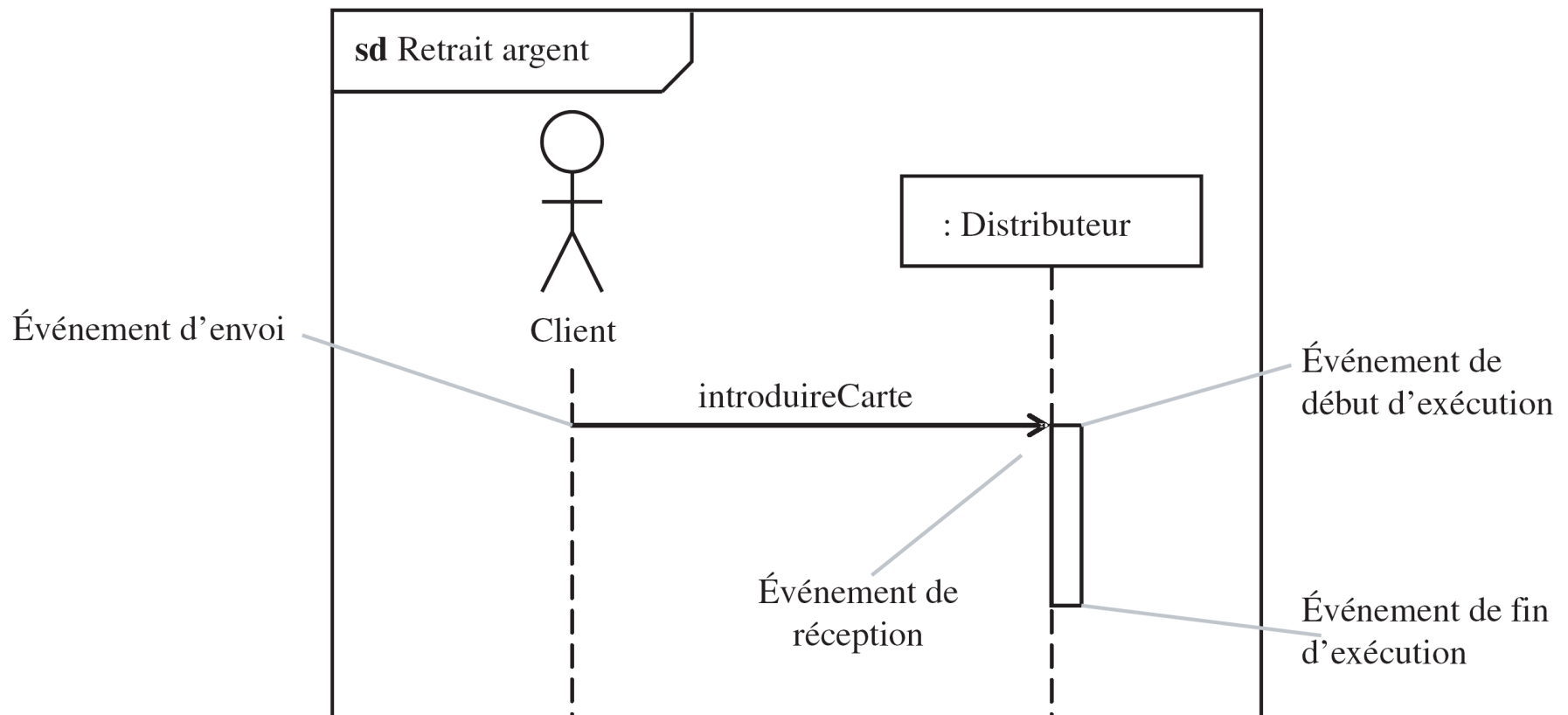


- La *destruction* d'un objet est matérialisée par une croix qui marque la fin de la ligne de vie de l'objet.



Messages et événements

- On distingue les *événements* d'*envoi* et de *réception* d'un message, ainsi que le début et la fin de l'exécution d'une éventuelle méthode appelée en réponse au message.



Messages complets, perdus et trouvés

- Un *message complet* est tel que les événements d'envoi et de réception sont connus.
 - Un message complet est représenté par une flèche partant d'une ligne de vie et arrivant à une ligne de vie
- Un *message perdu* est tel que l'événement d'envoi est connu, mais pas l'événement de réception.



- Un *message trouvé* est tel que l'événement de réception est connu, mais pas l'événement d'émission.





Syntaxe des messages

- La syntaxe des messages est :

`<nomSignalOuOpération>[' (' [<argument>']) ']`

où la syntaxe des arguments est la suivante :

`[<nomParamètre>'=']<valeurArgument>`

ou encore, si on veut un argument modifiable :

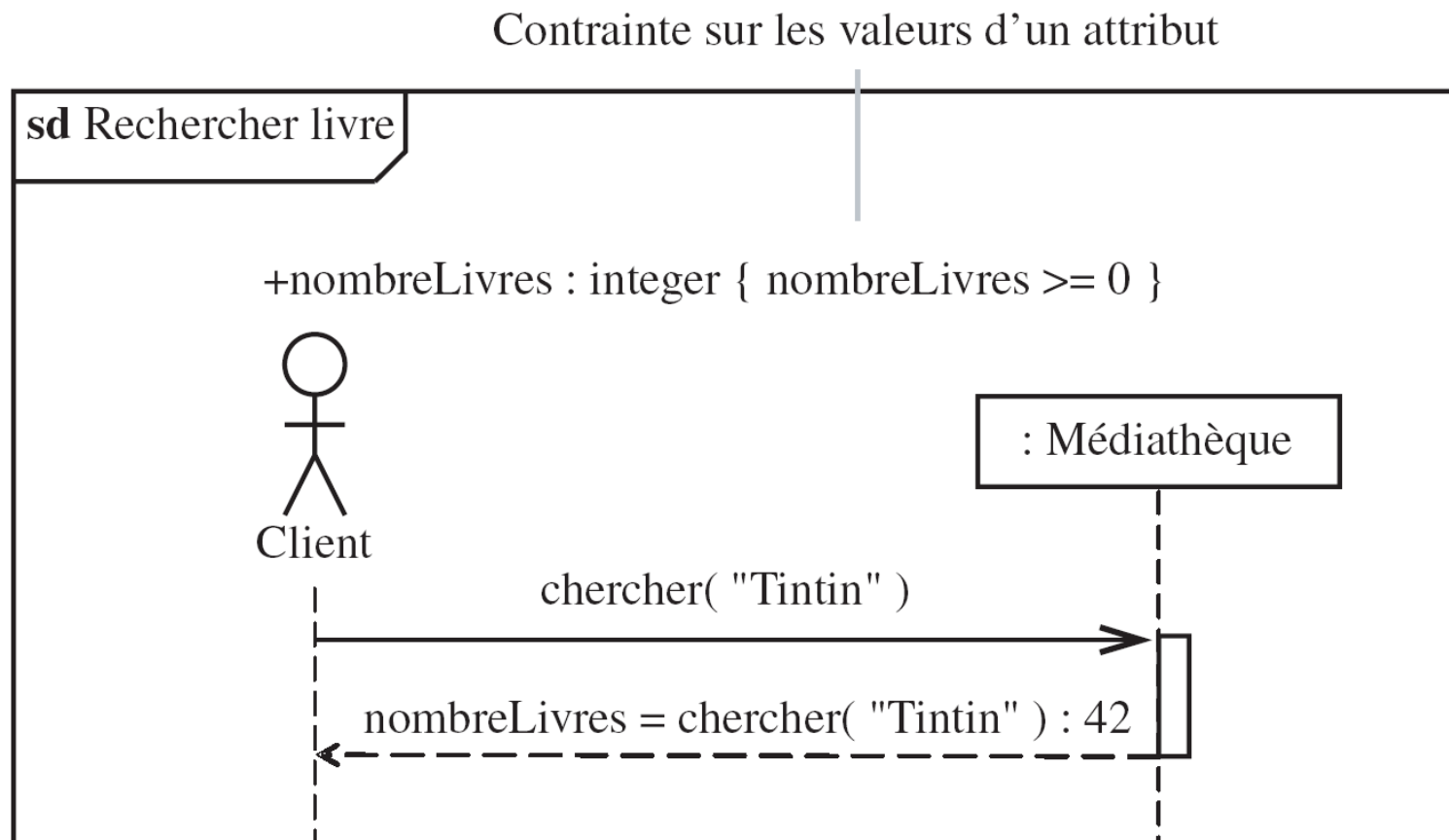
`[<nomParamètre>':']<valeurArgument>`

- Exemples :

- `appeler("Capitaine Hadock", 54214110)`
- `afficher(x, y)`
- `initialiser(x=100)`
- `f(x:12)`

Messages de retour

- Le récepteur du message peut répondre et transmettre un résultat via un message de retour.





Syntaxe des messages de retour

- La syntaxe des messages de retour est :

`<nomSignalOuOpération>['(' [<argument>', '...') ']`

où la syntaxe des arguments est la suivante :

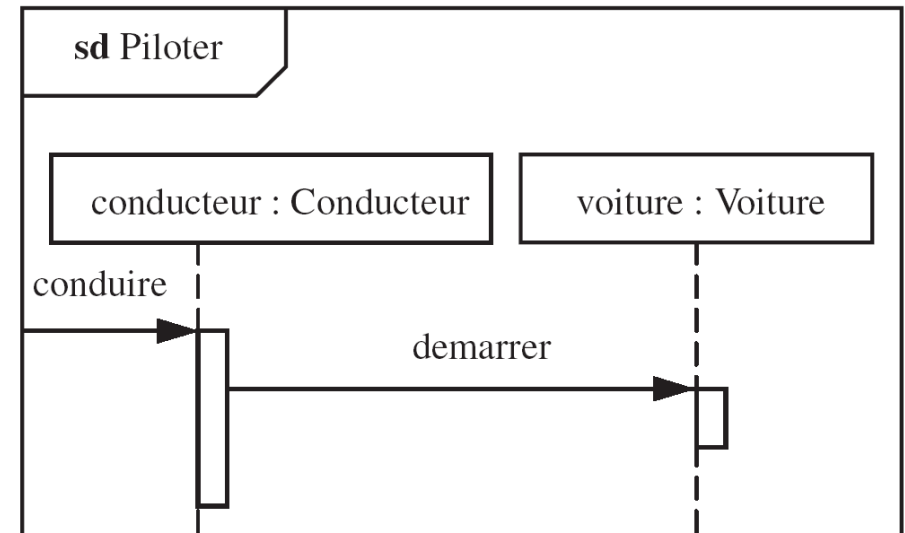
`[<nomParamètre>'=']<valeurArgument>`

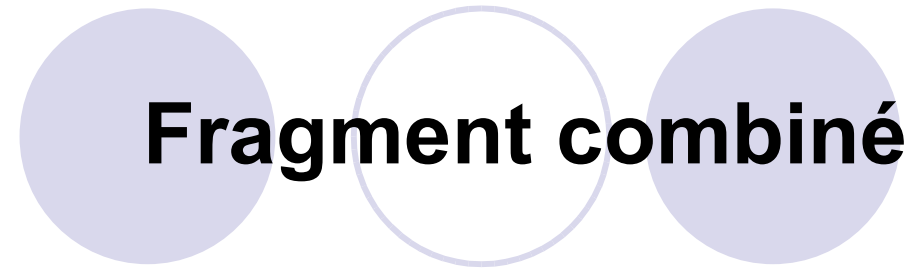
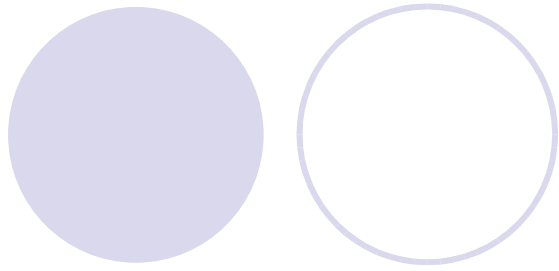
- Note : dans le cas de signaux, les arguments correspondent aux attributs du signal.
- Les messages de retour sont représentés en pointillés.

Implémentation des messages des diagrammes de séquence

```
public class Conducteur{
    private Voiture voiture;
    public void addVoiture( Voiture voiture ){
        if( voiture != null ){
            this.voiture = voiture;
        }
    }
    public void conduire(){
        voiture.demarrer();
    }
    public static void main( String [] argv ){
        // ...
        conducteur.conduire();
    }
}

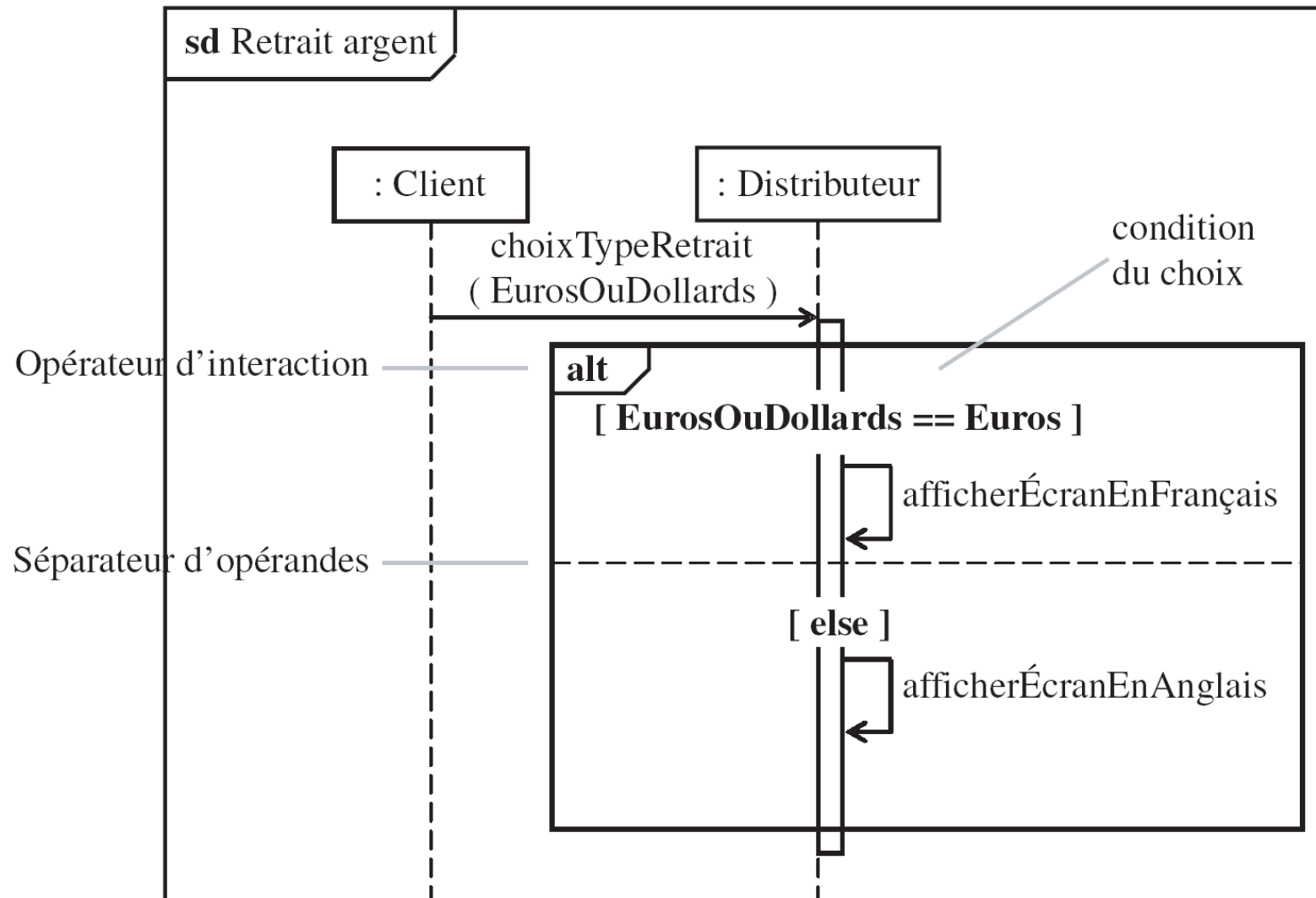
public class Voiture{
    public void demarrer(){
        // ...
    }
}
```





- Un *fragment combiné* permet de décomposer une interaction complexe en fragments suffisamment simples pour être compris.
 - Recombiner les fragments restitue la complexité
 - Syntaxe complète avec UML 2 : représentation complète de processus avec un langage simple (ex : processus parallèles)
- Un fragment combiné se représente de la même façon qu'une interaction. Il est représenté un rectangle dont le coin supérieur gauche contient un pentagone.
 - Dans le pentagone figure le type de la combinaison (appelé « opérateur d'interaction »).

Exemple de fragment avec l'opérateur conditionnel

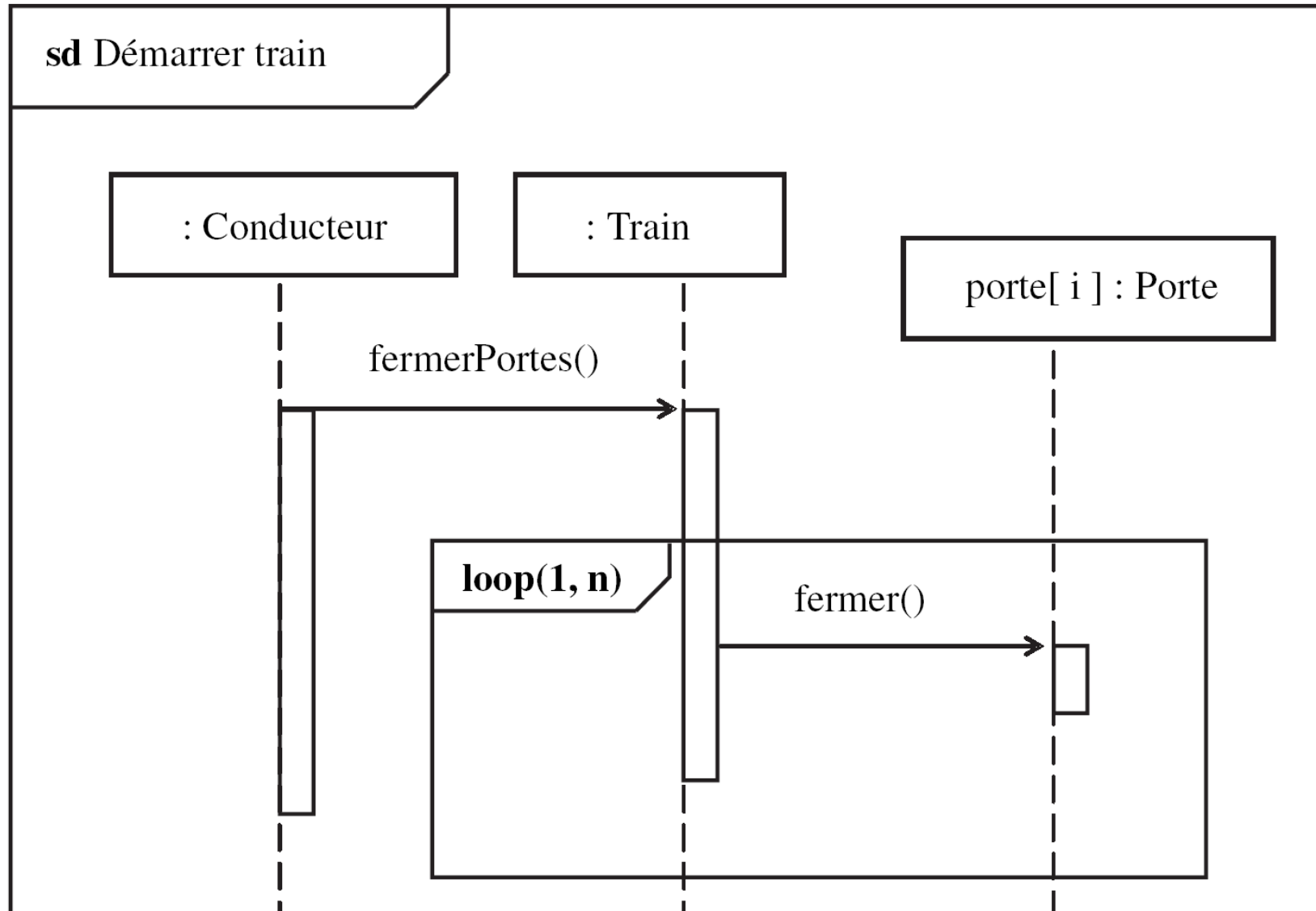




Type d'opérateurs d'interaction

- Opérateurs de **branchement** (*choix* et *boucles*) :
 - alternative, option, break et loop ;
- Opérateurs contrôlant l'envoi en *parallèle* de messages :
 - parallel et critical region ;
- Opérateurs contrôlant l'*envoi* de messages :
 - ignore, consider, assertion et negative ;
- Opérateurs fixant l'*ordre* d'envoi des messages :
 - weak sequencing, strict sequencing.

Opérateur de boucle





Syntaxe de l'opérateur loop

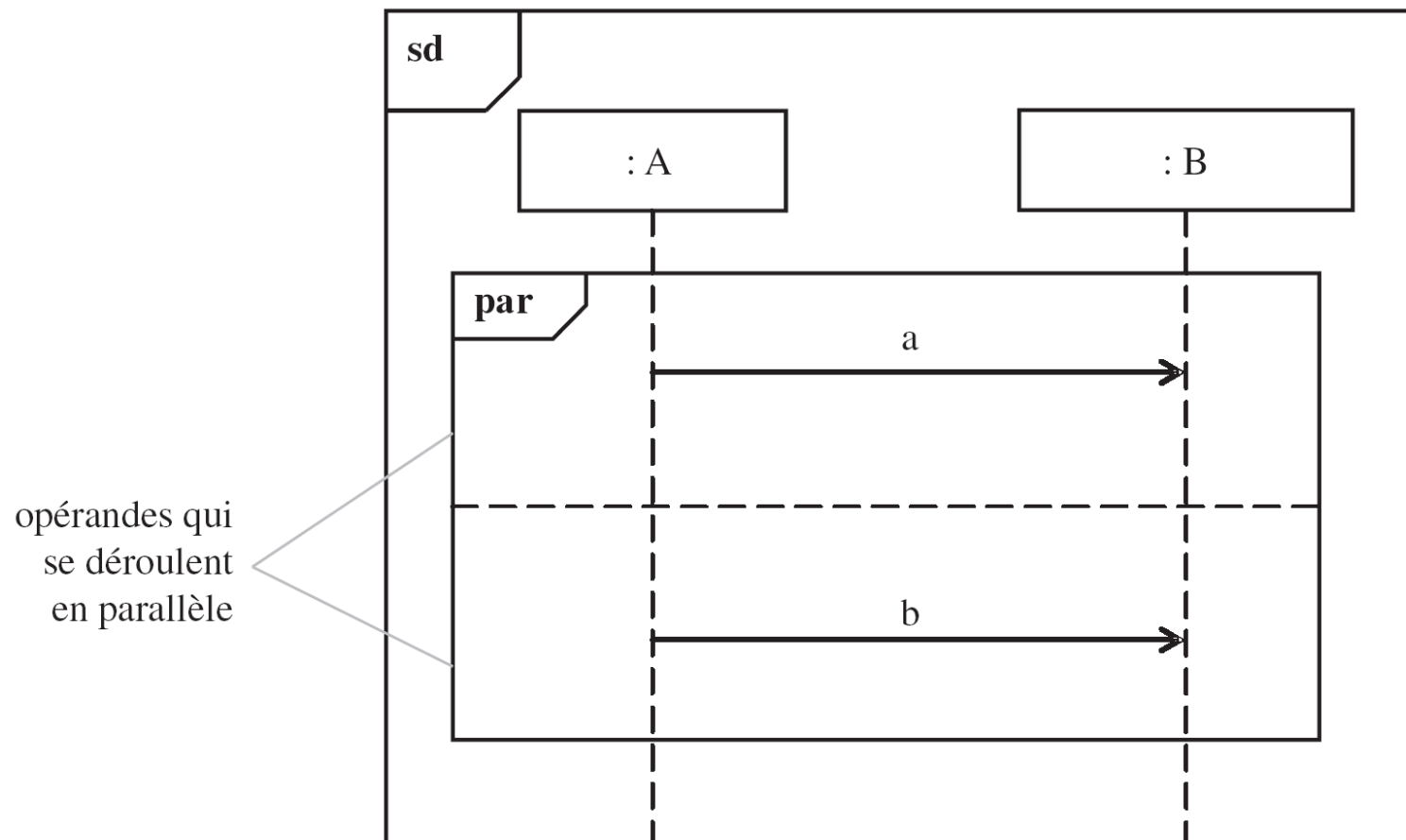
- La syntaxe d'une boucle est la suivante :

```
loop[ ' ( '<minInt>[ ', '<maxInt> ] ' ) ' ]
```

- La boucle est répétée au moins minInt fois avant qu'une éventuelle condition booléenne ne soit testée (la condition est placée entre crochets sur la ligne de vie)
- Tant que la condition est vraie, la boucle continue, au plus maxInt fois.
- Notations :
 - `loop(valeur)` est équivalent à `loop(valeur, valeur)`.
 - `loop` est équivalent à `loop(0, *)`, où `*` signifie « illimité ».
 - On peut adjoindre une condition booléenne entre crochets. Dans ce cas, les messages du fragment seront répétés *tant que* la condition est vraie.
 - Ex : `loop [i>0]` (tant que i est supérieur à 0, faire...)

Opérateur parallèle

- L'opérateur *par* permet d'envoyer des messages en parallèle.



Opérateurs de contrôle d'envoi des messages

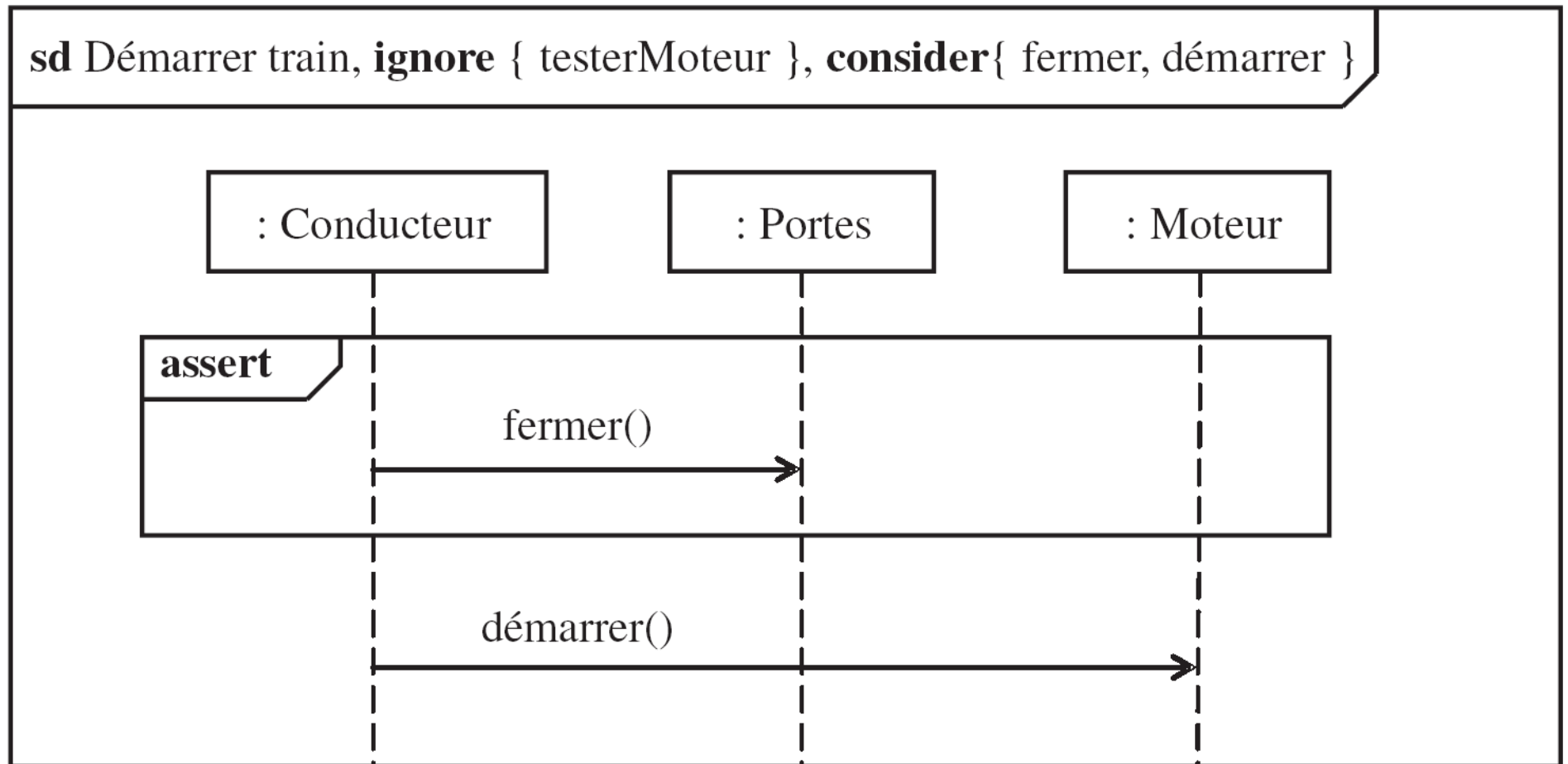
- Les opérateurs *ignore* et *consider* permettent de focaliser l'attention du modélisateur sur une partie des messages qui peuvent être envoyés durant une interaction.
 - L'opérateur *ignore* définit l'ensemble des messages à ignorer
 - L'opérateur *consider* définit l'ensemble de ceux qu'il faut prendre en compte.

```
ignore{listeNomsMessagesIgnorés}  
consider {listeNomsMessagesConsidérés}
```

(séparateur de liste : virgule « , »)

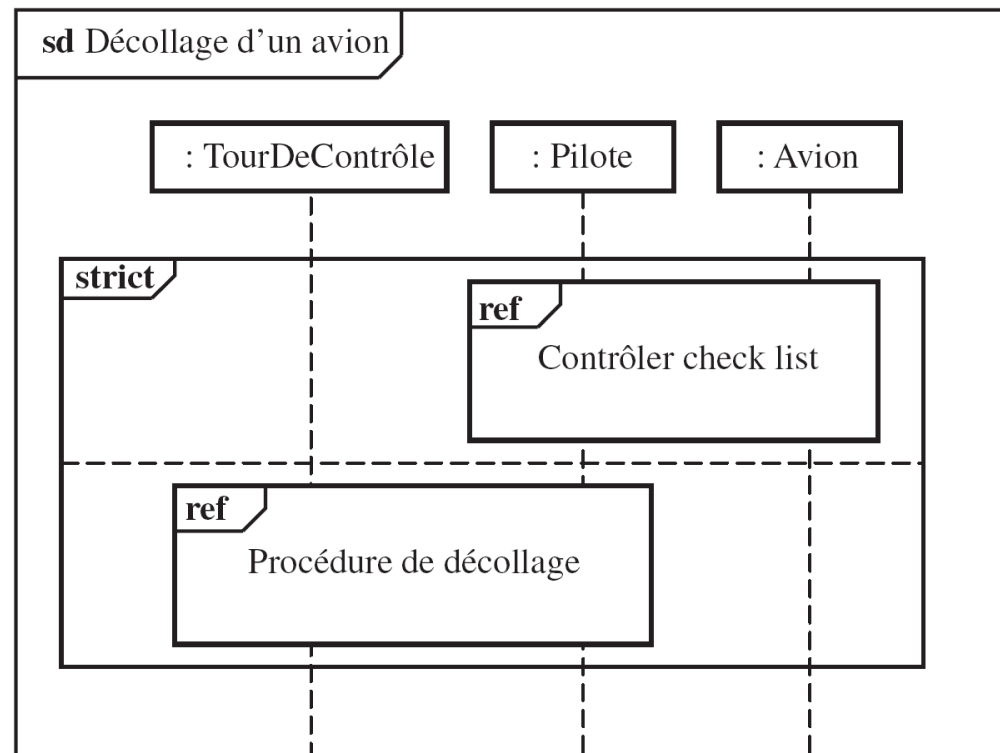
- L'opérateur *assert* permet de spécifier qu'un fragment doit nécessairement être effectuée.

Exemple de contrôle d'envoi des messages



Opérateur de séquençement strict

- L'opérateur *strict* contraint strictement l'ordre d'exécution de ses opérandes, du haut vers le bas.
 - Utile surtout lorsque deux parties d'un diagramme n'ont pas de ligne de vie en commun.





Réutilisation d'une itération

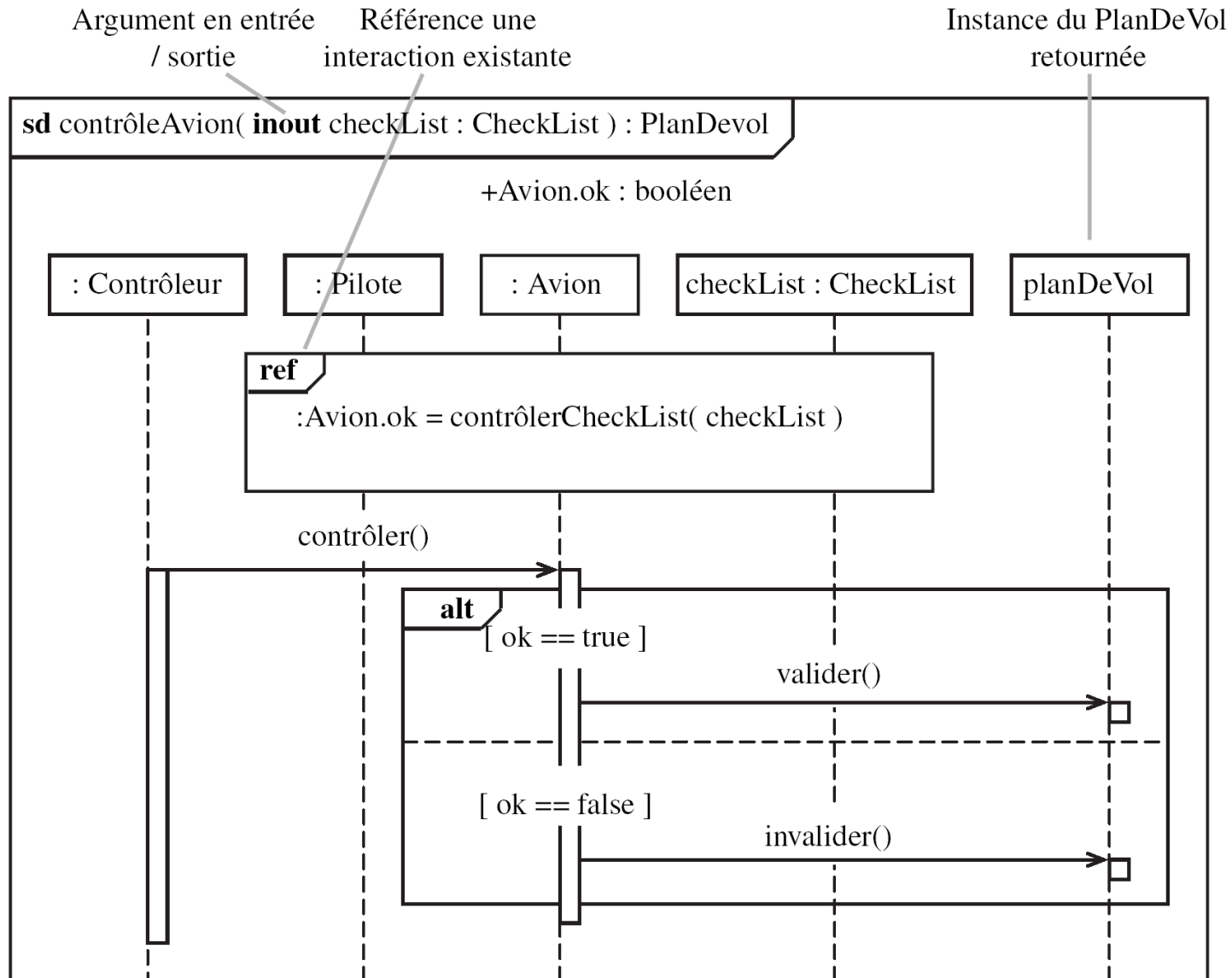
- *Réutiliser une interaction* consiste à placer un fragment portant la référence « *ref* » là où l'interaction est utile.
- La syntaxe complète pour spécifier l'interaction à réutiliser est la suivante :

```
[<nomAttributValeurRetour>'=']
```

```
<nomInteraction>
```

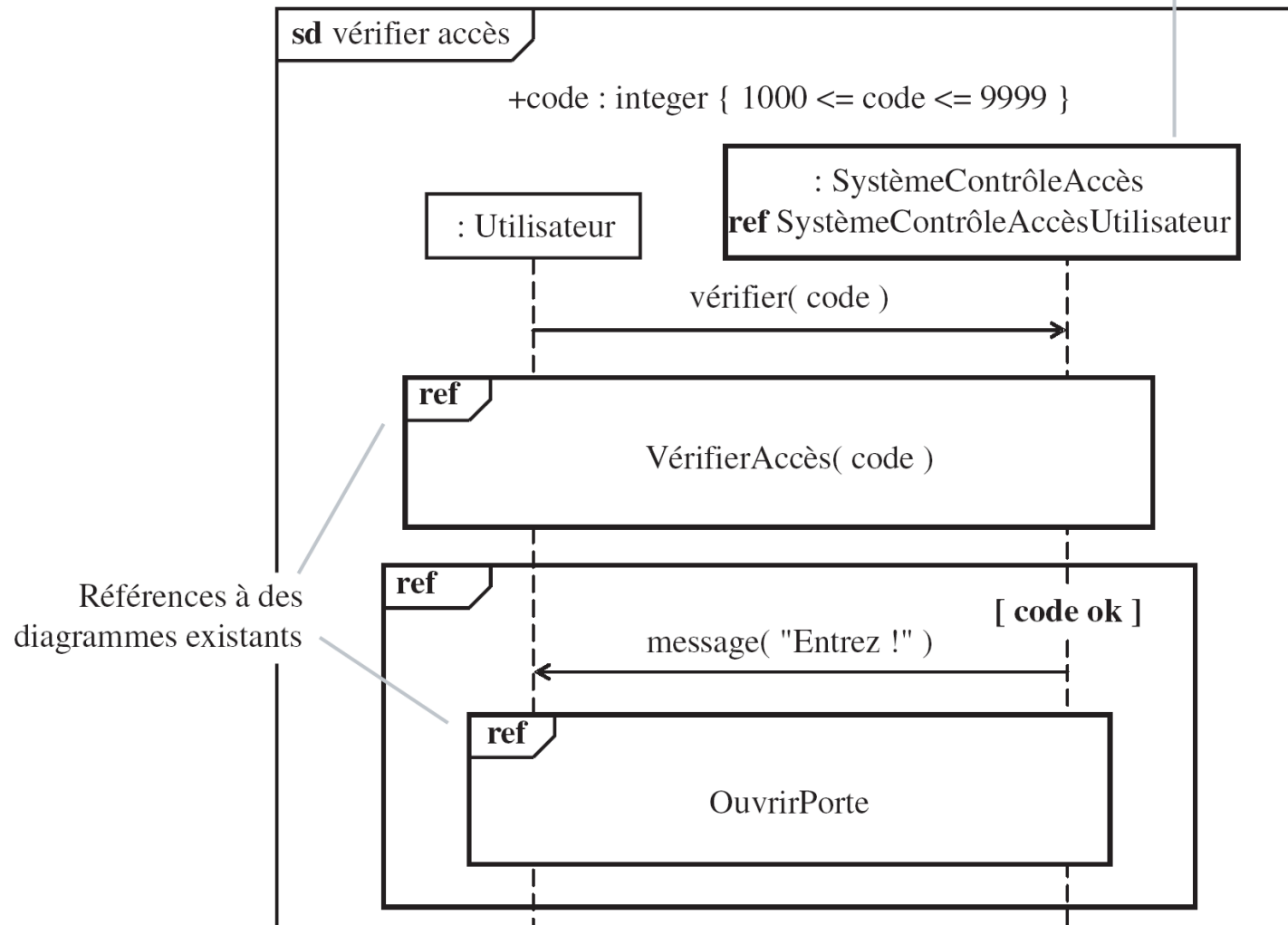
```
['(' [<arguments>] ')'] [':' <valeurRetour>]
```

Exemple de réutilisation



Décomposition d'une ligne de vie

Ligne de vie décomposée
sur la figure 3.26



Décomposition d'une ligne de vie

Autre notation
pour la décomposition

portes

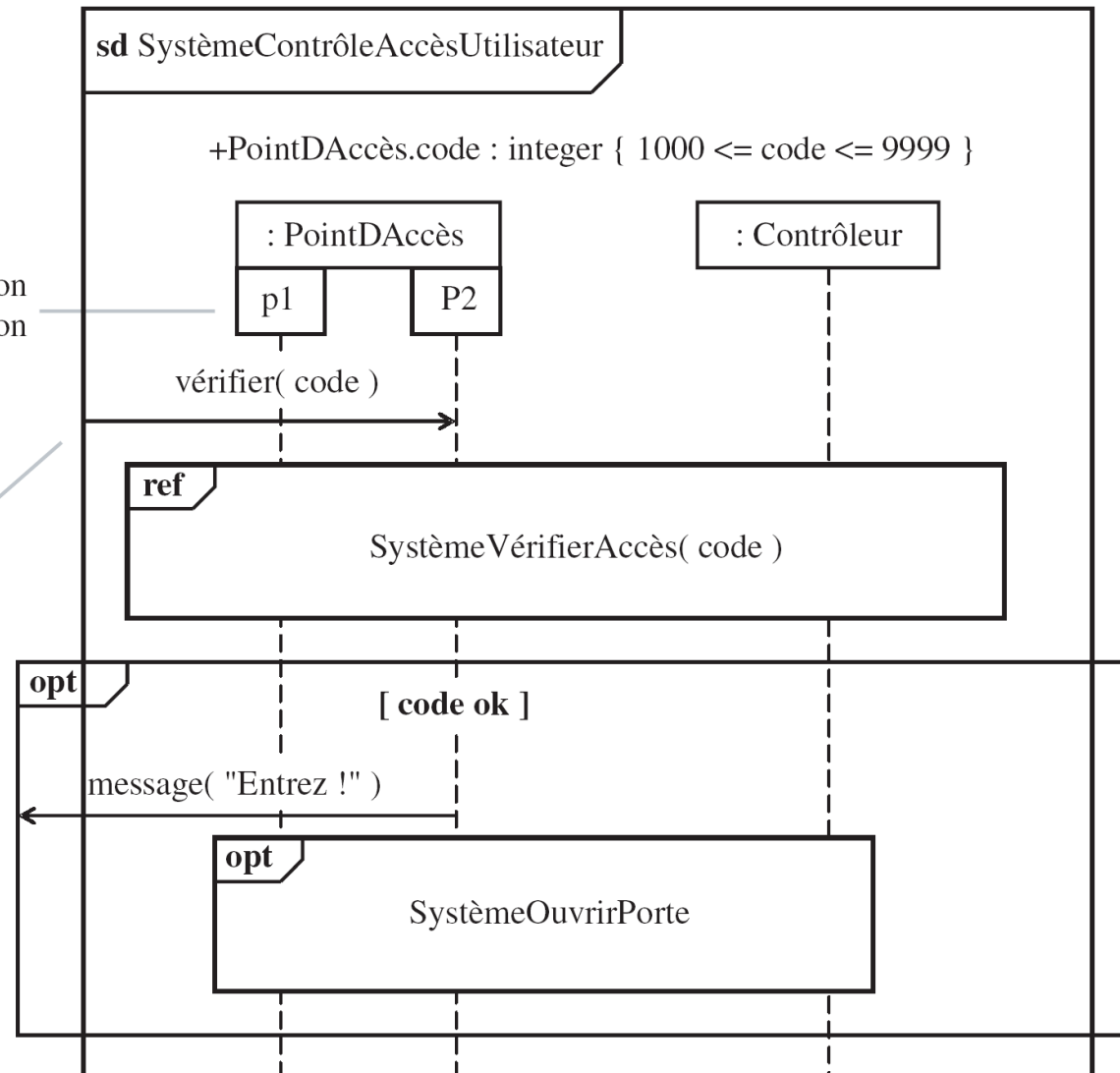
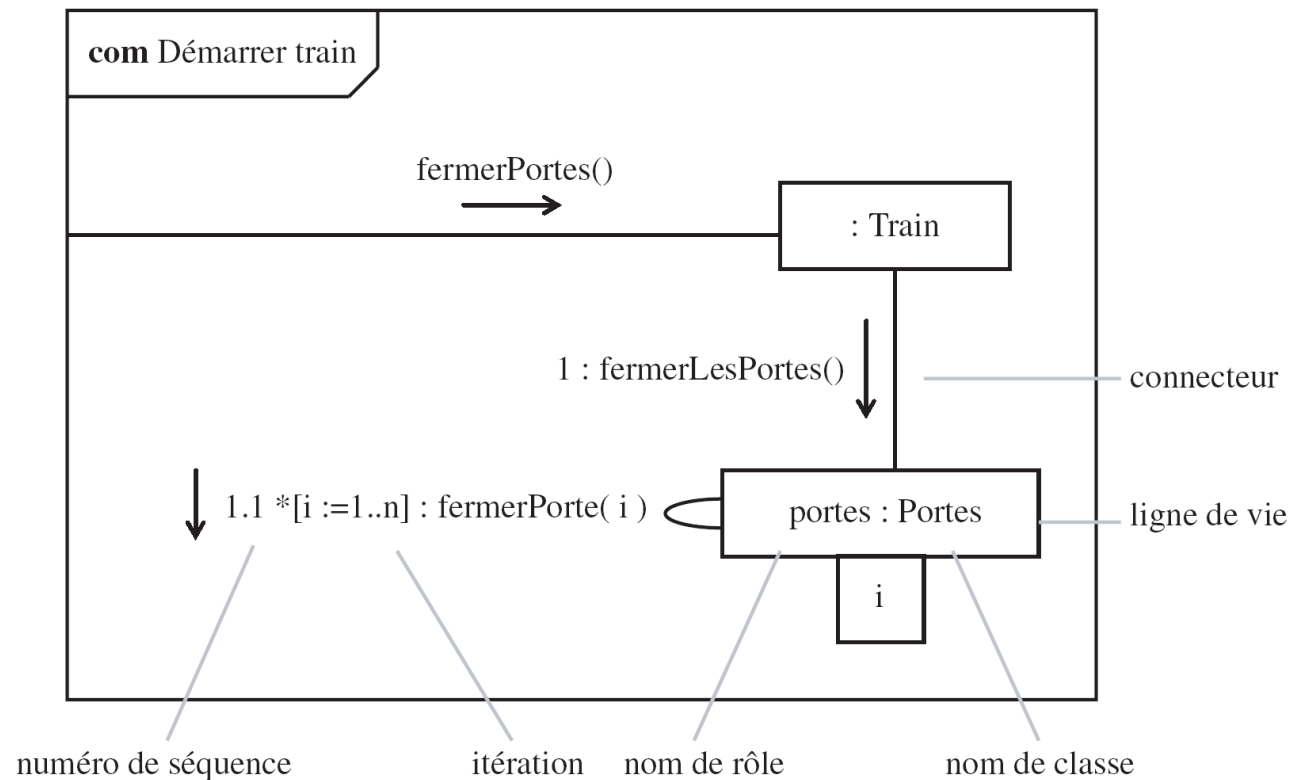


Diagramme de communication

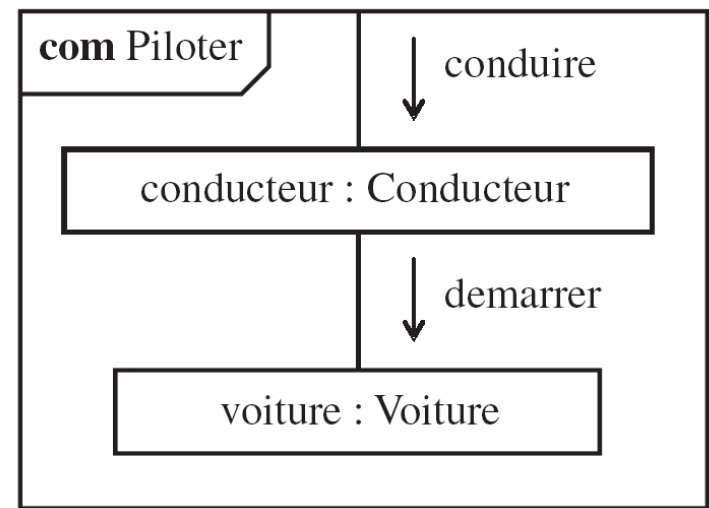
- Le diagramme de communication donne une représentation spatiale des interactions, plutôt que temporelle.
 - Relations entre les lignes de vie qui communiquent.

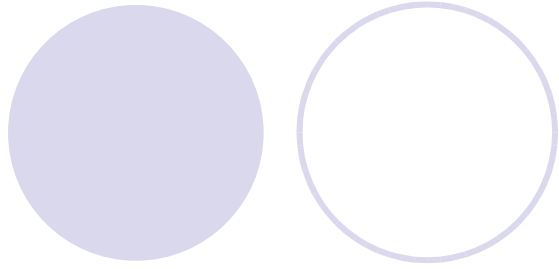


Implémentation des messages des diagrammes de communication

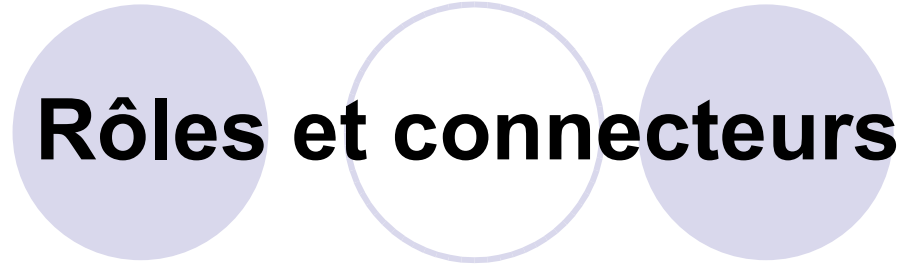
```
public class Conducteur{
    private Voiture voiture;
    public void addVoiture( Voiture voiture ){
        if( voiture != null ){
            this.voiture = voiture;
        }
    }
    public void conduire(){
        voiture.demarrer();
    }
    public static void main( String [] argv ){
        // ...
        conducteur.conduire();
    }
}

public class Voiture{
    public void demarrer(){
        // ...
    }
}
```





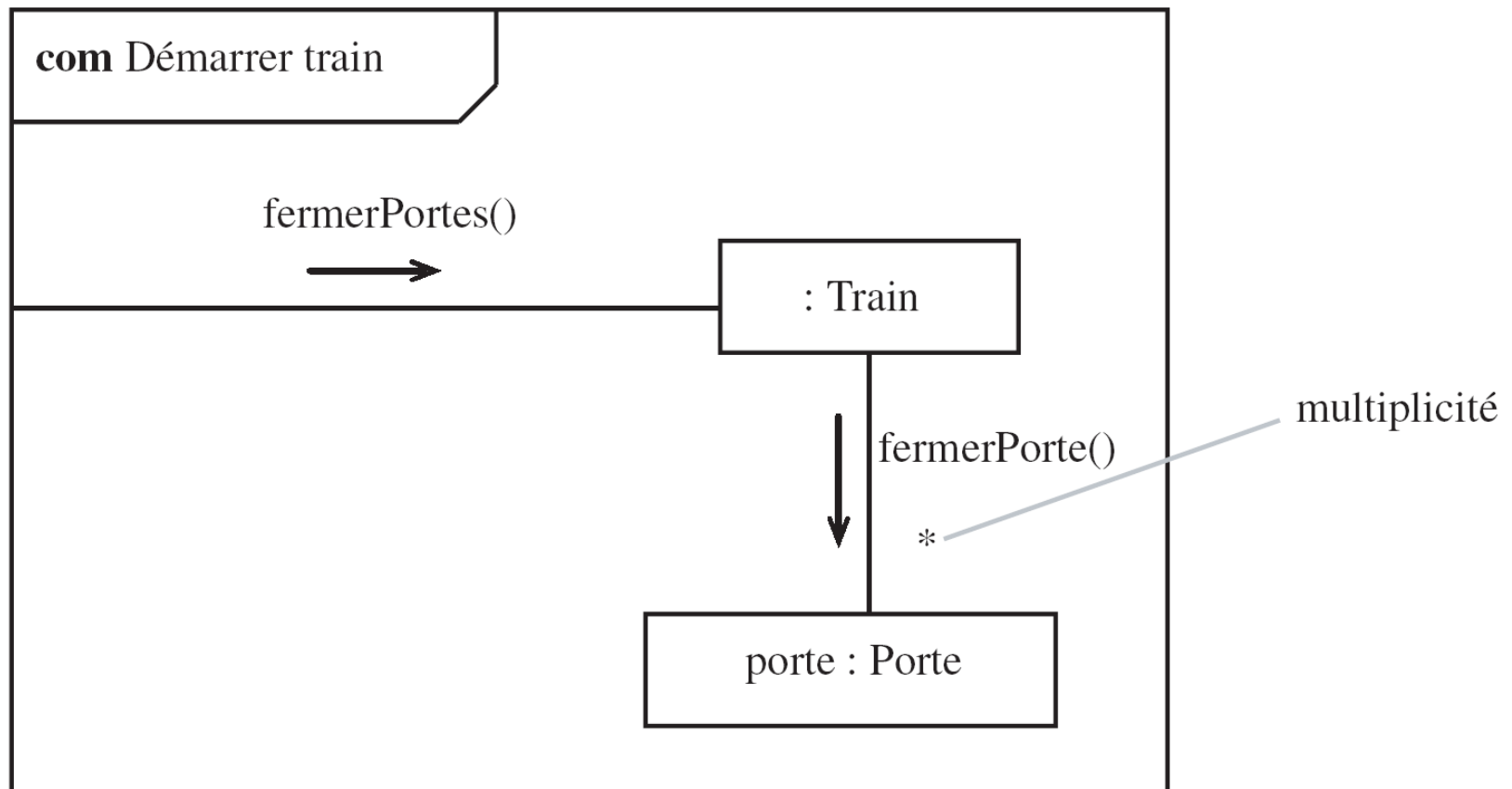
Rôles et connecteurs



- La syntaxe d'une ligne de vie est :
$$\langle \text{nomRôle} \rangle : \langle \text{nomType} \rangle$$
- Le *rôle* permet de définir le contexte d'utilisation de l'interaction.
 - Si la ligne de vie est un objet, celui-ci peut avoir plusieurs rôles au cours de sa vie.
- Les relations entre les lignes de vie sont appelées « *connecteurs* ».
 - Un connecteur se représente de la même façon qu'une association mais la sémantique est plus large : un connecteur est souvent une association transitoire.

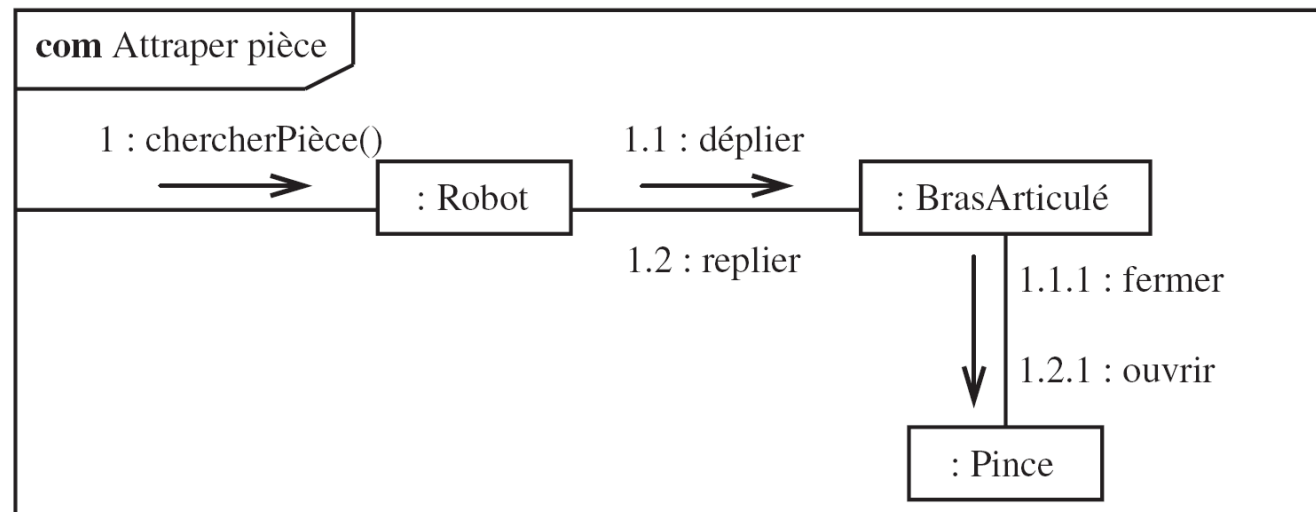
Multiplicité

- Les connecteurs permettent de rendre compte de la multiplicité.

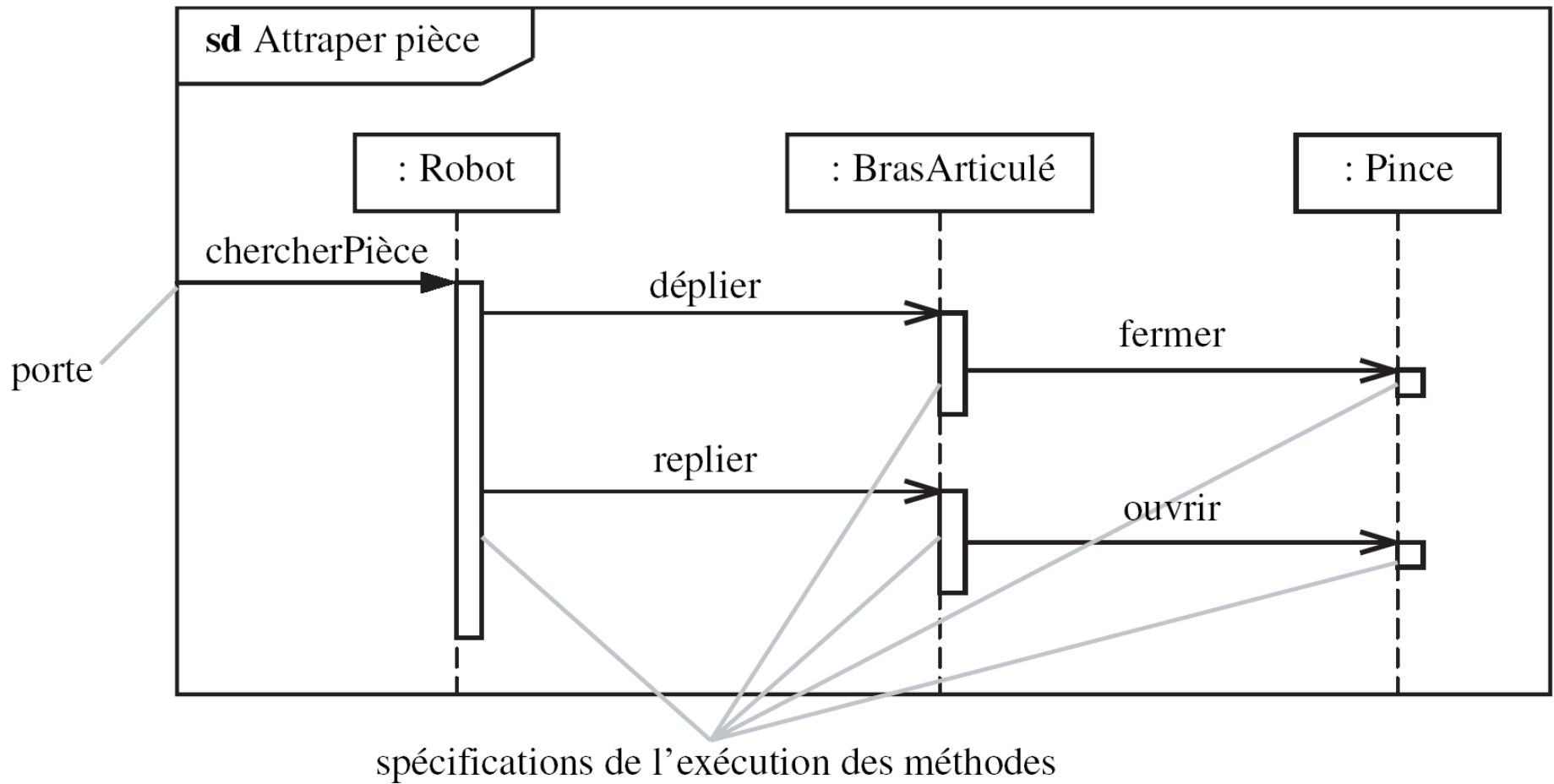


Numéros de séquence des messages

- Pour représenter les aspects temporels, on adjoint des *numéros de séquence* aux messages.
 - Des messages successifs sont ordonnés selon un numéro de séquence croissant.
 - Des messages envoyés en cascade (ex : appel de méthode à l'intérieur d'une méthode) portent un numéro d'emboîtement avec une notation pointée.
 - On utilise des lettres plutôt que des numéros pour représenter la simultanéité.

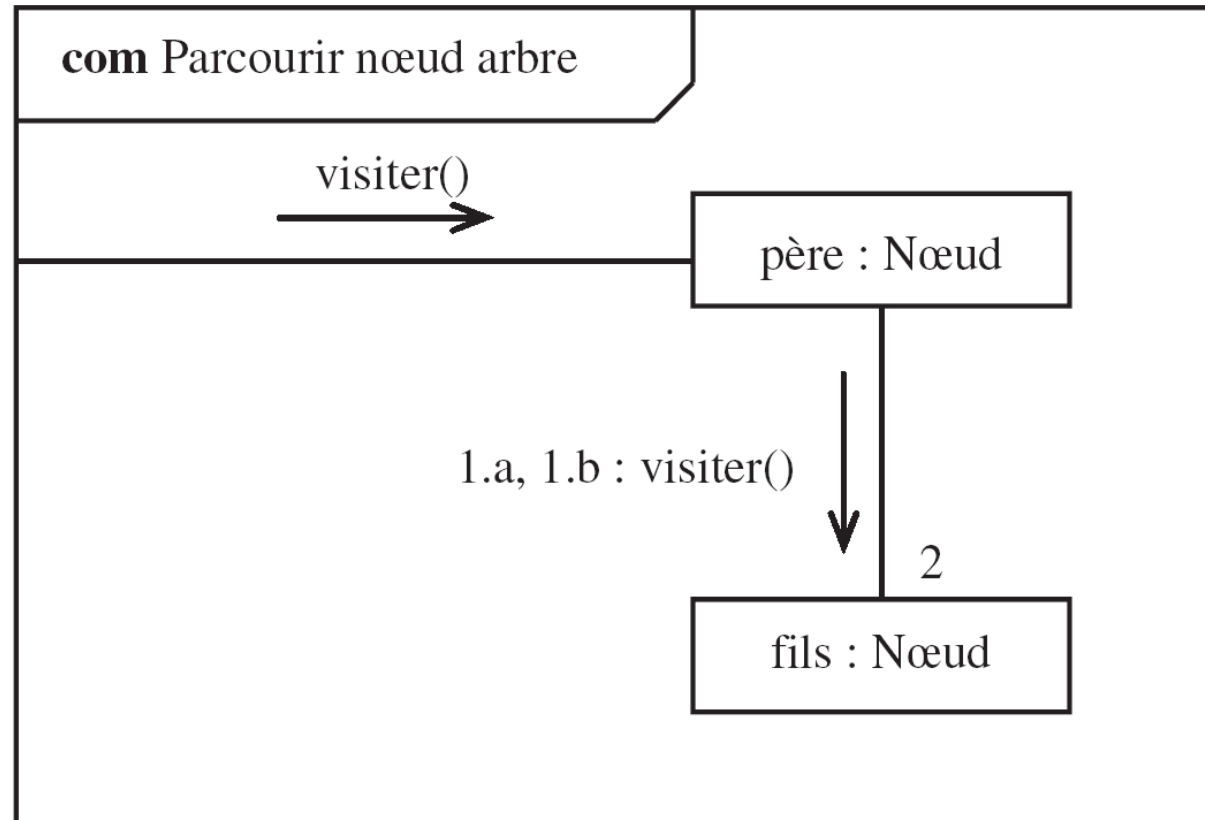


Equivalence avec un diagramme de séquence



Messages simultanés

- Lorsque des messages sont envoyés en parallèle, on numérote les messages avec des lettres





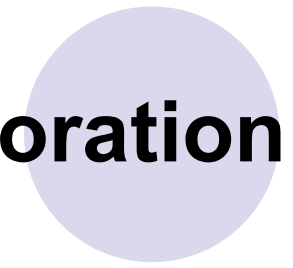
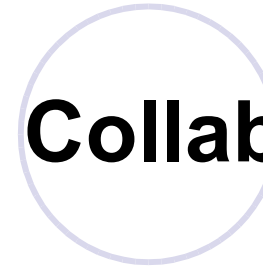
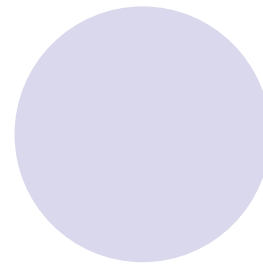
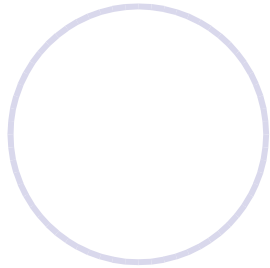
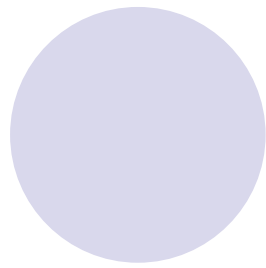
Opérateurs de choix et de boucles

- Pas d'opérateurs combinés dans les diagrammes de communication
 - * [`<clauseItération>`] représente une itération.
 - La clause d'itération peut être exprimée dans le format $i := 1 \dots n$.
 - [`<clauseCondition>`] représente un choix. La clause de condition est une condition booléenne.



Syntaxe des messages

- La syntaxe complète des messages est :
`[<numéroSéquence>] [<expression>] :<message>`
 - « message » a la même forme que dans les diagrammes de séquence, « numéroSéquence » représente le numéro de séquençement des messages et « expression » précise une itération ou un embranchement.
- Exemples :
 - `2 : affiche(x, y) : message simple.`
 - `1.3.1 : trouve("Hadock") : appel emboîté.`
 - `4 [x < 0] : inverse(x, couleur) : conditionnel.`
 - `3.1 *[i:=1..10] : recommencer() : itération.`



Collaboration

- Une *collaboration* montre des instances qui collaborent dans un contexte donné pour mettre en oeuvre une fonctionnalité d'un système.
- Une collaboration se représente par une ellipse en trait pointillé comprenant deux compartiments.
 - Le compartiment supérieur contient le nom de la collaboration ayant pour syntaxe :
`<nomDuRôle>[':' <NomDuType>] [multiplicité]`
 - Le compartiment inférieur montre les participants à la collaboration.

Exemple de collaboration

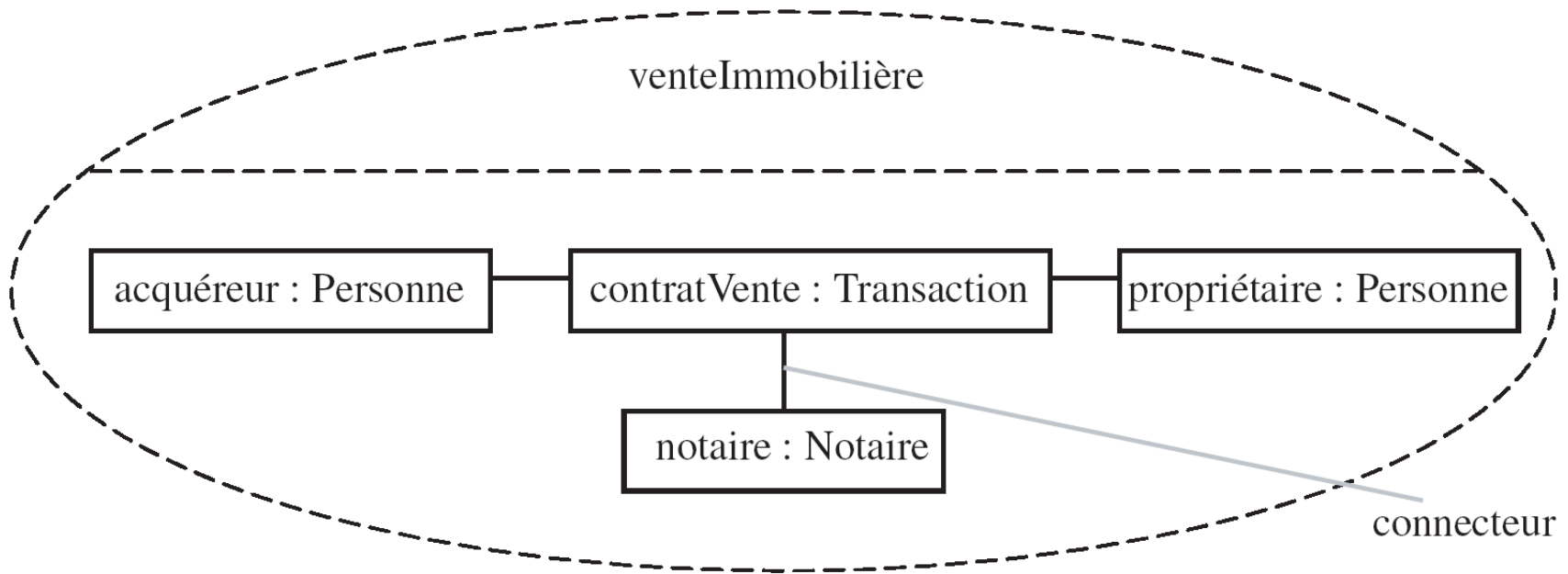
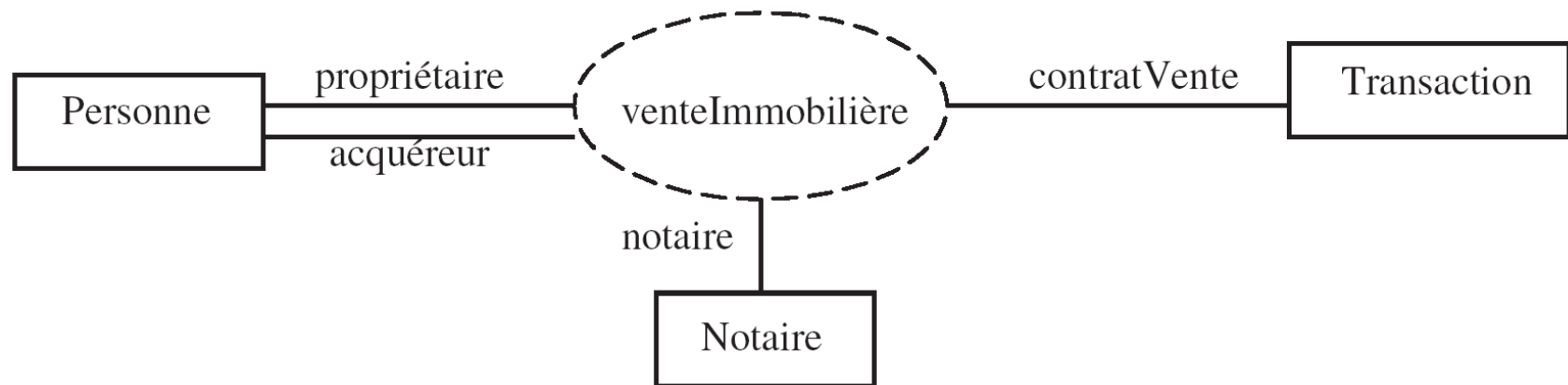


Diagramme de collaboration

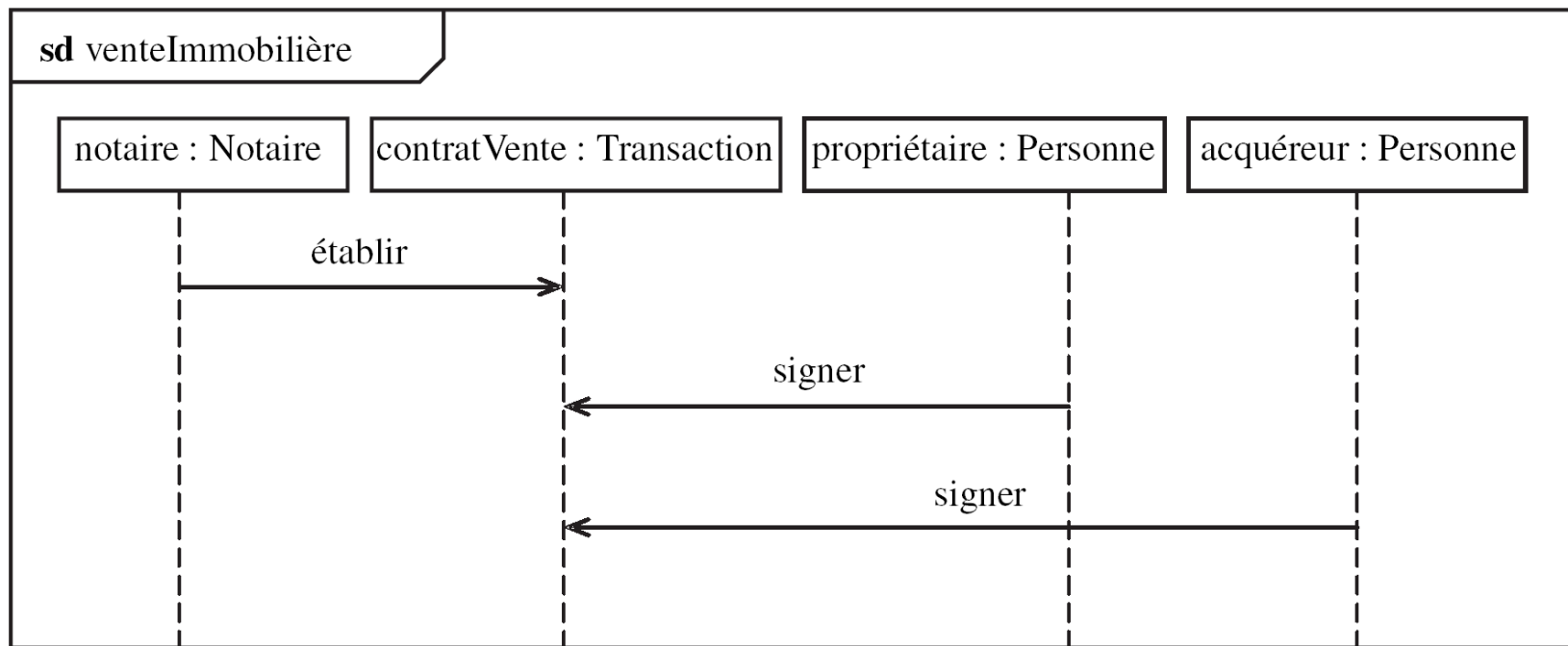
- Une collaboration peut aussi se représenter par une ellipse sans compartiment, portant le nom de la collaboration en son sein. Les instances sont reliées à l'ellipse par des lignes qui portent le nom du rôle de chaque instance.



• C

Collaborations et interactions

- Les collaborations donnent lieu à des interactions



- Les interactions documentent les collaborations
- Les collaborations organisent les interactions.

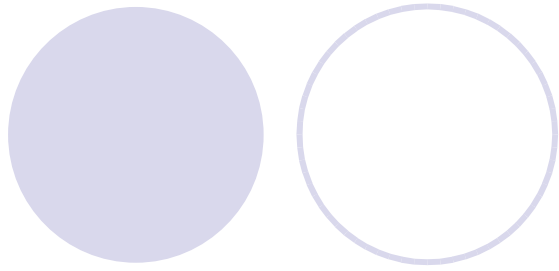
Diagramme d'états-transitions

UML 2.0



Automates

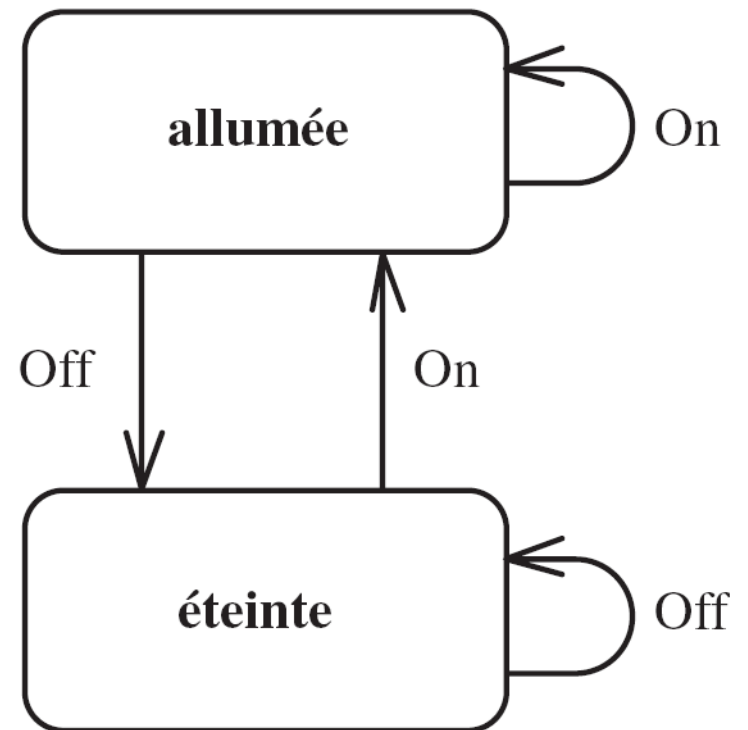
- Un *automate à états finis* est la spécification de la séquence d'états que subira un objet au cours de son cycle de vie.
- Un tel automate représente le comportement d'un classeur dont les sorties
 - ne dépendent pas seulement de ses entrées,
 - mais aussi d'un historique des sollicitations passées.
- Cet historique est caractérisé par un *état*.
- Les objets changent d'état en réponse à des *événements* extérieurs donnant lieu à des *transitions* entre états.
- Sauf cas particuliers, à chaque instant, chaque objet est dans un et un seul état.



- Les états sont représentés par des rectangles aux coins arrondis
- Les transitions sont représentées par des arcs orientés liant les états entre eux.
- Certains états, dits « composites », peuvent contenir des sous-diagrammes.



Etat et transition



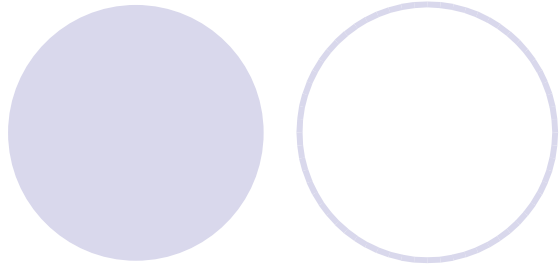
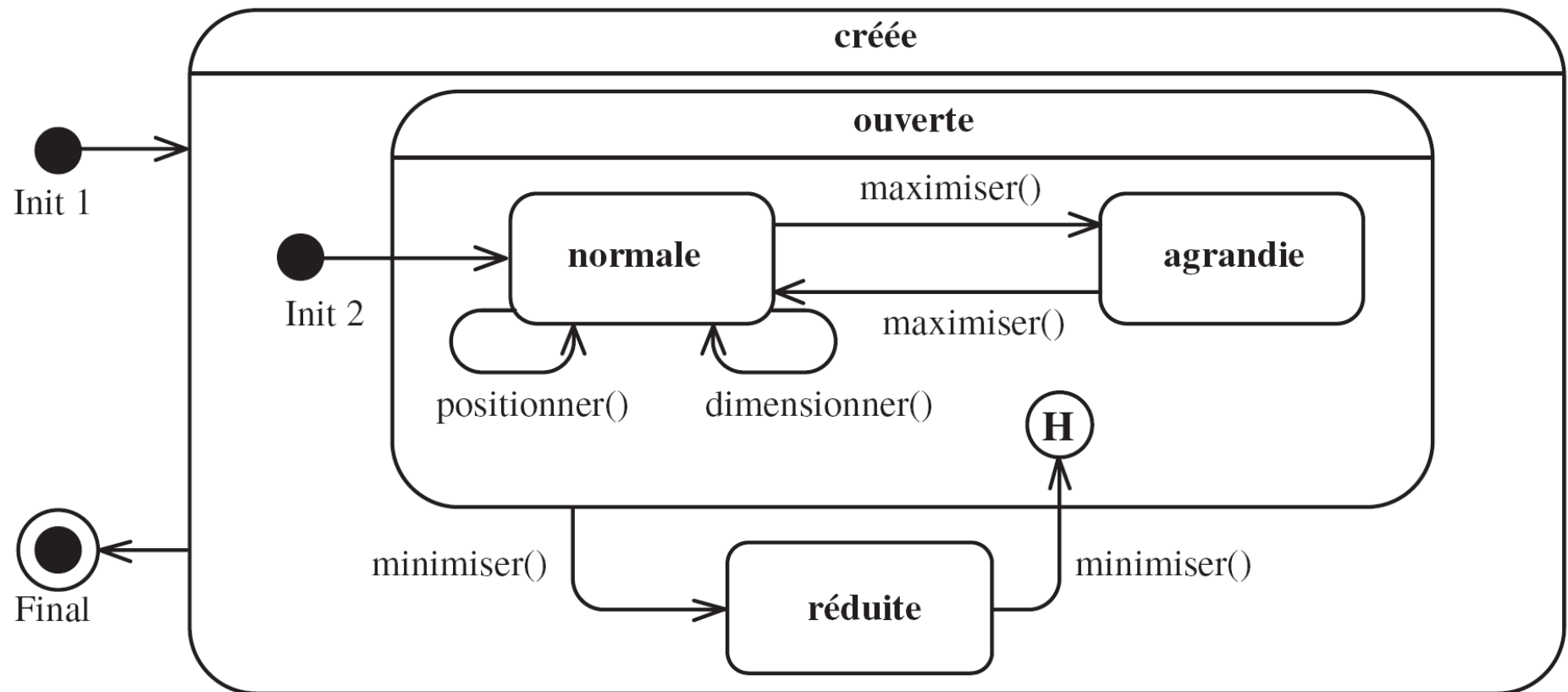


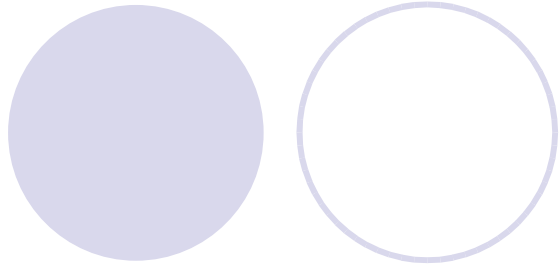
Diagramme d'état-transition



- L'organisation des états et des transitions pour un classeur donné est représentée dans un diagramme d'états.
- Le modèle dynamique comprend plusieurs diagrammes d'états.
 - On en construit un pour chaque classeur (classe le plus souvent) ayant un comportement dynamique important.
 - Chaque diagramme d'états est associé à un seul classeur.
- Chaque automate à états finis s'exécute concurremment et peut changer d'état de façon indépendante.

Exemple de diagramme d'états-transitions



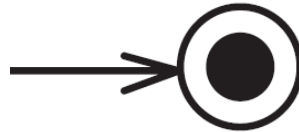


Etat initial et état final

- L'état initial est un pseudo-état qui définit le point de départ par défaut pour l'automate ou le sous-état.
 - Lorsqu'un objet est créé, il entre dans l'état initial.



- L'état final est un pseudo-état qui indique que l'exécution de l'automate ou du sous-état est terminée.





Événement déclencheur

- Les transitions d'un diagramme d'états-transitions sont donc déclenchées par des *événements déclencheurs*.
 - Un appel de méthode sur l'objet courant génère un événement de type *call*.
 - Le passage de faux à vrai de la valeur de vérité d'une condition booléenne génère implicitement un événement de type *change*.
 - La réception d'un signal asynchrone, explicitement émis par un autre objet, génère un événement de type *signal*.
 - L'écoulement d'une durée déterminée après un événement donné génère un événement de type *after*. Par défaut, le temps commence à s'écouler dès l'entrée dans l'état courant.



Événements call et signal

- Un événement de type *call* ou *signal* est déclaré ainsi :

`nomÉvénement '(' ListeParamètres ') '`

où chaque paramètre a la forme :

`nomParamètre ':' typeParamètre`

- Les événements de type *call* sont des méthodes déclarées au niveau du diagramme de classes.
- Les *signaux* sont déclarés par la définition d'une classe portant le stéréotype signal, ne fournissant pas d'opérations, et dont les attributs sont interprétés comme des arguments.



Evénements *change* et *after*

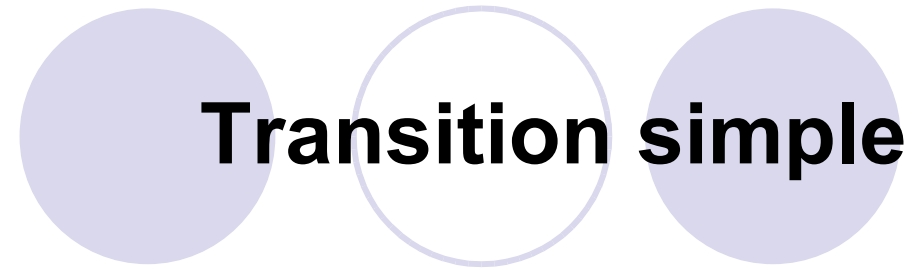
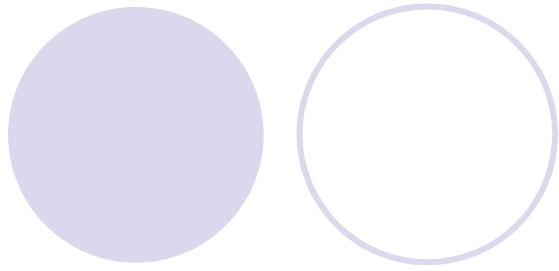
- Un événement de type *change* est introduit de la façon suivante :

`when '(' condition-bouléenne ')'`

- Il prend la forme d'un test continu et se déclenche potentiellement à chaque changement de valeurs des variables intervenant dans la condition.
- Un événement temporel de type *after* est spécifié par :

`after '(' paramètre ')'`

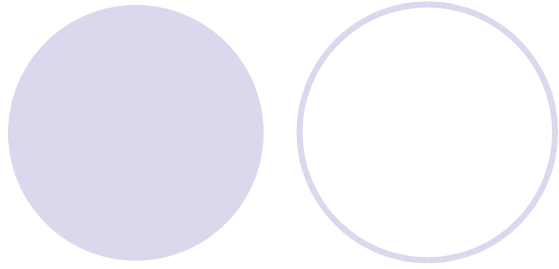
 - Le paramètre s'évalue comme une durée, par défaut écoulée depuis l'entrée dans l'état courant.
 - Par exemple : `after(10 secondes)`.



- Une transition entre deux états simples est représentée par un arc qui les lie l'un à l'autre.
 - Elle indique qu'une instance peut changer d'état et exécuter certaines activités, si un événement déclencheur se produit et que les conditions de garde sont vérifiées.
- Sa syntaxe est la suivante :

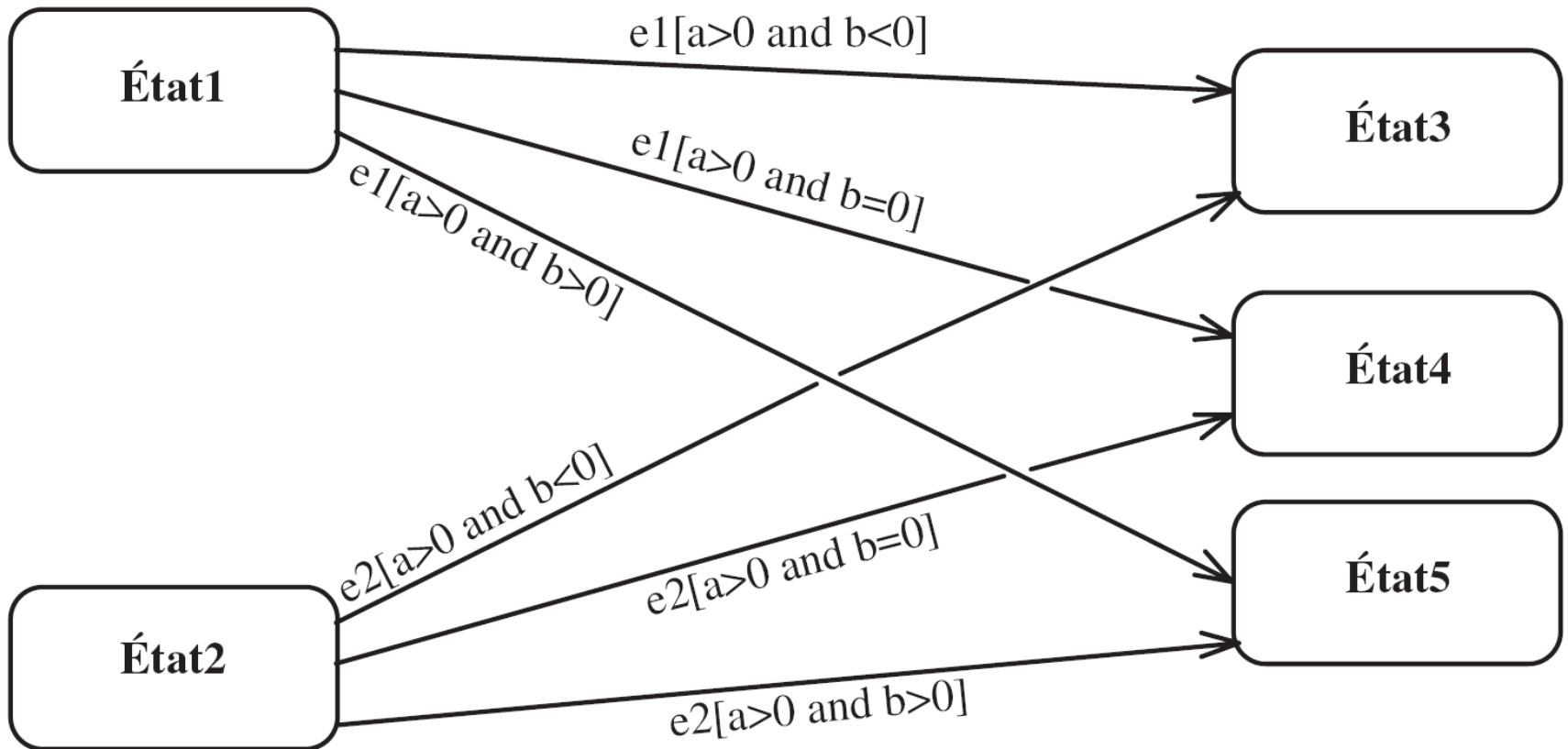
```
nomEvénement '(' listeParamEvénement ') '  
'[' garde ']' '/'activité
```

 - La garde désigne une condition qui doit être remplie pour pouvoir déclencher la transition,
 - L'activité désigne des instructions à effectuer au moment du tir.



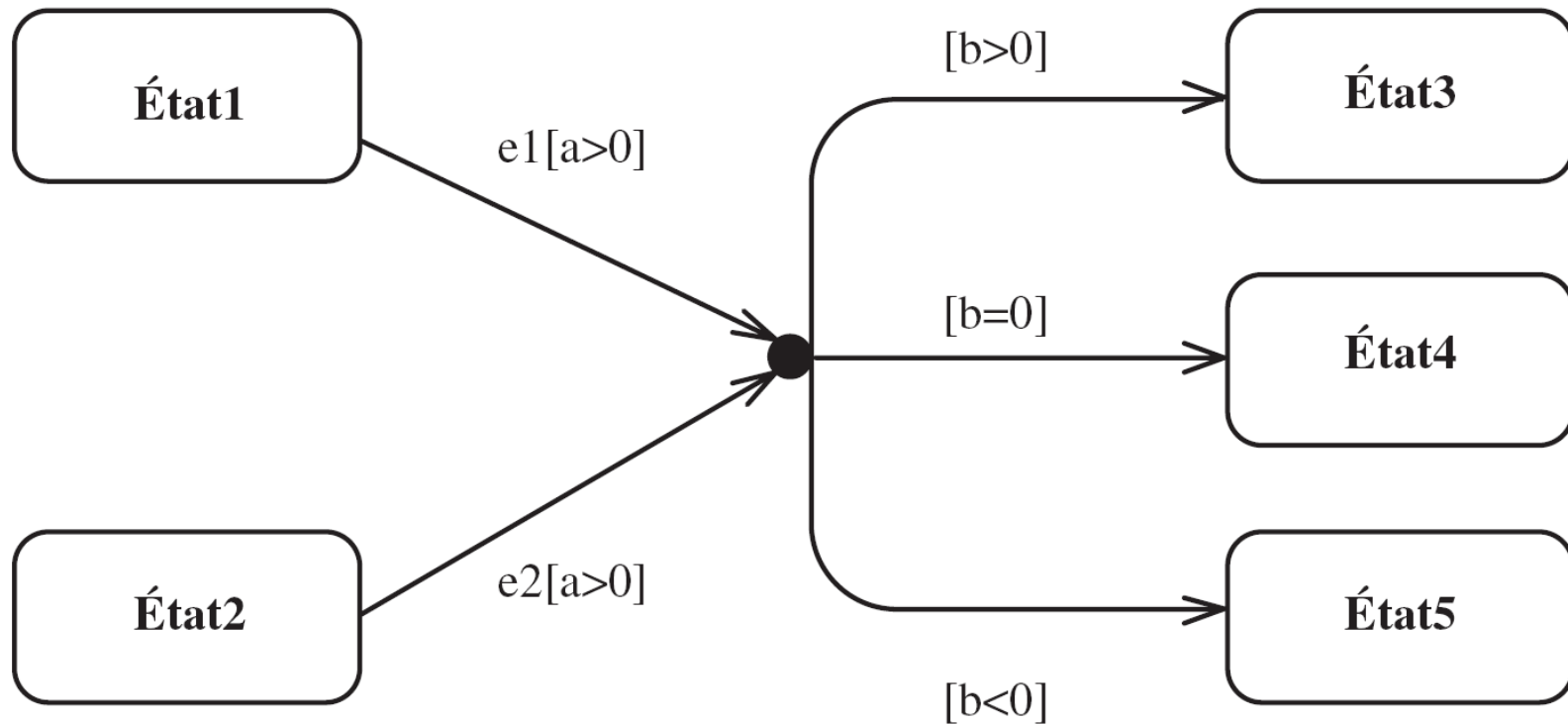
- On peut représenter des alternatives pour le franchissement d'une transition.
- On utilise pour cela des pseudo-états particuliers :
 - Les *points de jonction* (petit cercle plein) permettent de partager des segments de transition.
 - Ils ne sont qu'un raccourci d'écriture.
 - Ils permettent des représentations plus compactes.
 - Les *points de choix* (losange) sont plus que des raccourcis d'écriture.

Simplification avec les points de jonction



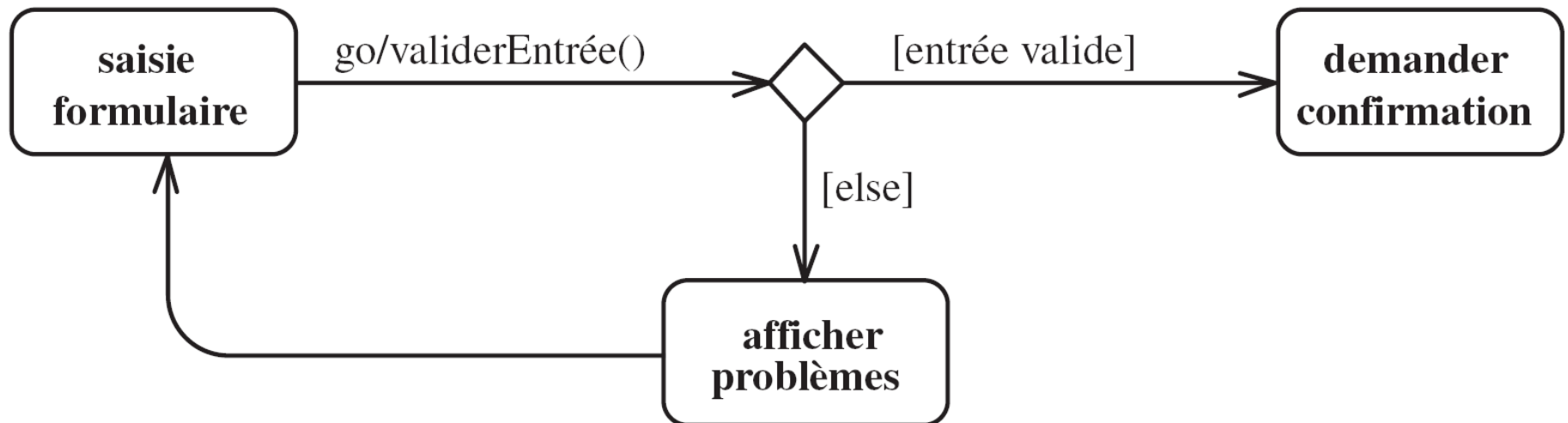
Simplification avec les points de jonction

- Pour emprunter un chemin, toutes les gardes le long de ce chemin doivent s'évaluer à vrai dès le franchissement du premier segment.

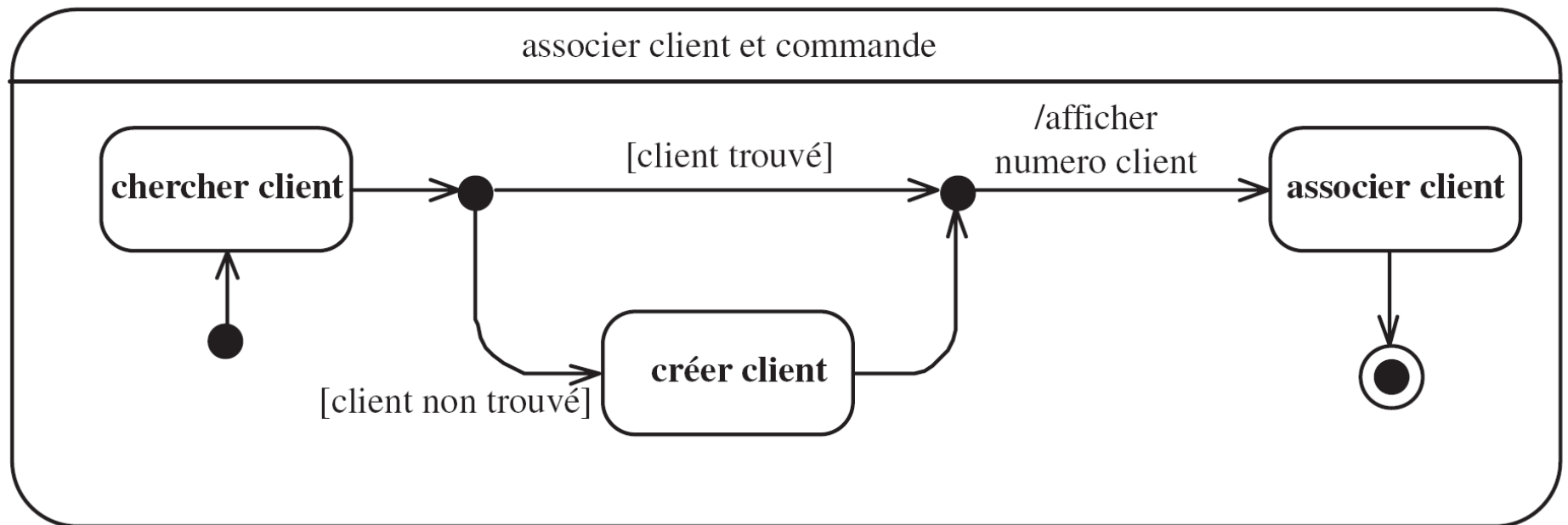


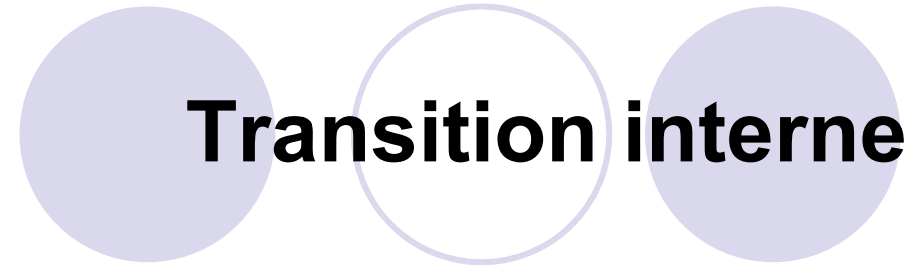
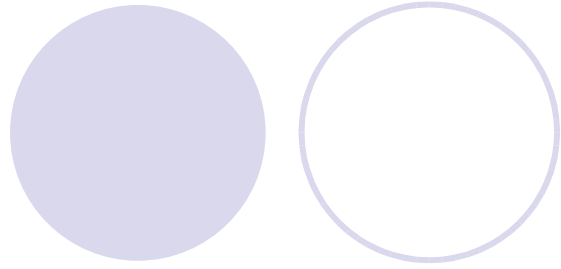
Utilisation des points de choix

- Les gardes après le point de choix sont évaluées au moment où il est atteint.
 - Cela permet de baser le choix sur des résultats obtenus en franchissant le segment avant le point de choix.
 - Si, quand le point de choix est atteint, aucun segment en aval n'est franchissable, le modèle est *mal formé*.

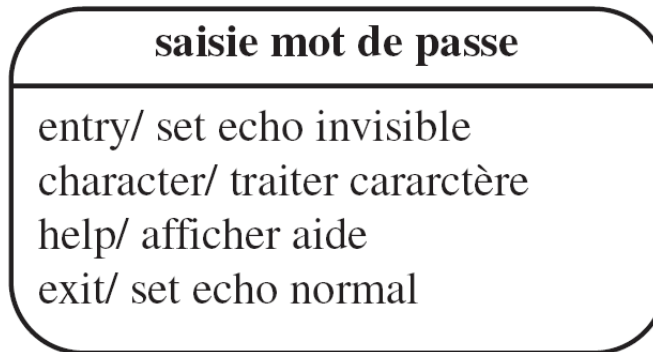


Représentation d'alternatives avec les points de jonction





- Un objet reste dans un état durant une certaine durée et des transitions internes peuvent intervenir.
- Une *transition interne* ne modifie pas l'état courant, mais suit globalement les règles d'une transition simple entre deux états.
- Trois déclencheurs particuliers sont introduits permettant le tir de transitions internes : *entry/*, *do/*, et *exit/*.





Déclencheurs de transitions internes prédéfinis

- « entry » définit une activité à effectuer à chaque fois que l'on rentre dans l'état considéré.
- « exit » définit une activité à effectuer quand on quitte l'état.
- « do » définit une activité continue qui est réalisée tant que l'on se trouve dans l'état, ou jusqu'à ce que le calcul associé soit terminé.
 - On pourra dans ce dernier cas gérer l'événement correspondant à la fin de cette activité (completion event).
- « include » permet d'invoquer un sous-diagramme d'états-transitions.

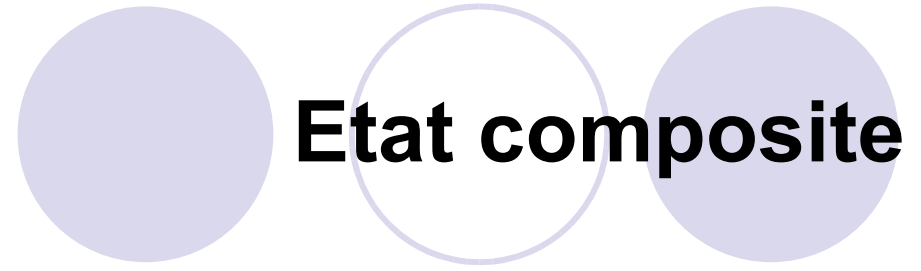
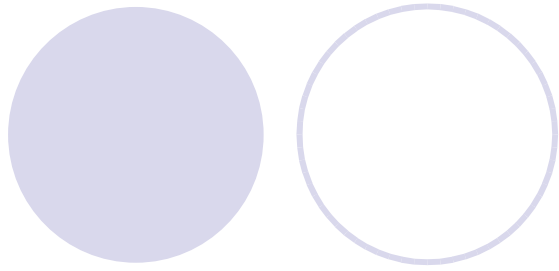
Représentation des transitions internes

- Les transitions internes sont spécifiées dans le compartiment inférieur de l'état, sous le compartiment du nom.
- Chaque transition interne est décrite selon la syntaxe suivante :

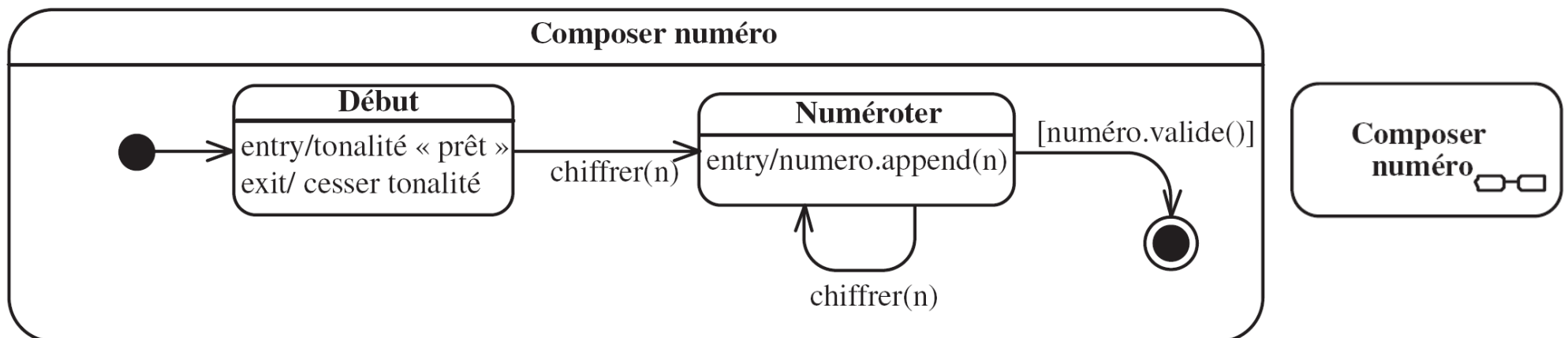
NomÉvénement ' ('liste-paramètres') '
' ['garde'] ' '/' activitéARéaliser

saisie mot de passe

entry/ set echo invisible
character/ traiter caractère
help/ afficher aide
exit/ set echo normal



- Un *état composite*, par opposition à un état dit « simple », est décomposé en deux ou plusieurs sous-états.
 - Tout état ou sous-état peut ainsi être décomposé en sous-états enchaînés sans limite a priori de profondeur.
 - Un état composite est représenté par les deux compartiments de nom et d'actions internes habituelles, et par un compartiment contenant le sous-diagramme.



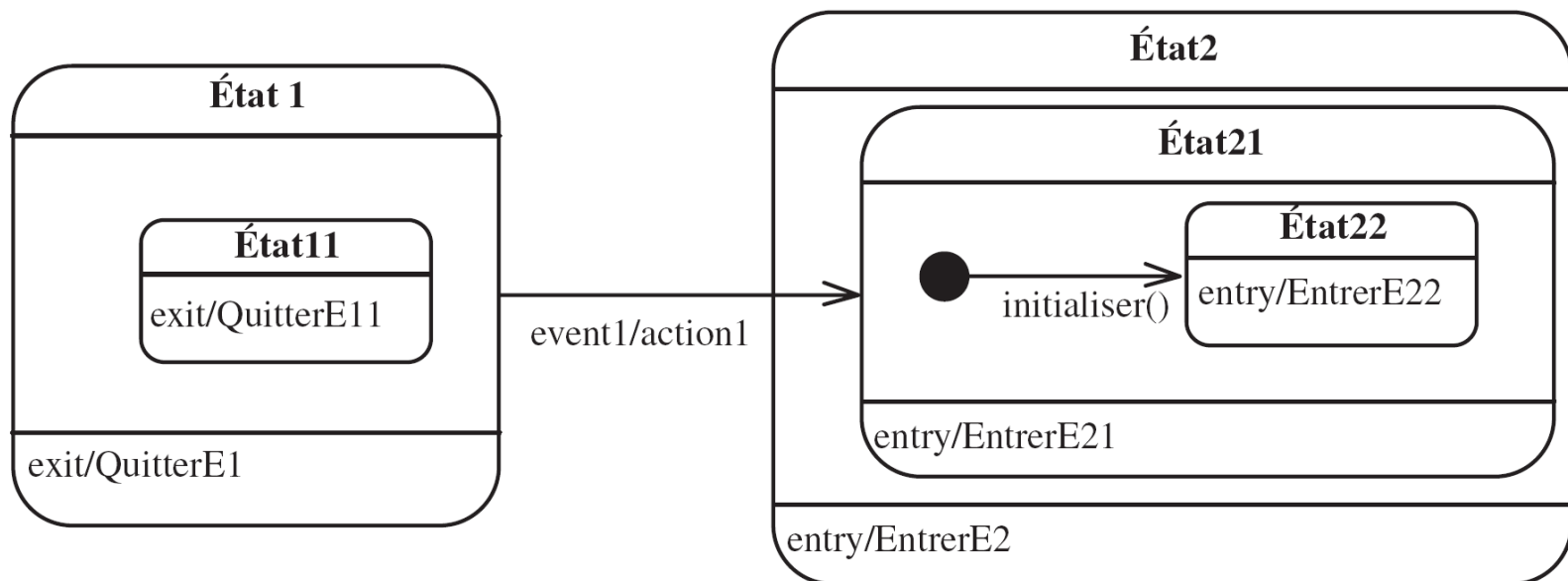


Etats composites et états initiaux/finaux

- Les transitions peuvent avoir pour cible la frontière d'un état composite. Elle sont alors équivalentes à une transition ayant pour cible l'état initial de l'état composite.
- Une transition ayant pour source la frontière d'un état composite est équivalente à une transition qui s'applique à tout sous-état de l'état composite source.
 - Cette relation est transitive et peut « traverser » plusieurs niveaux d'imbrication.

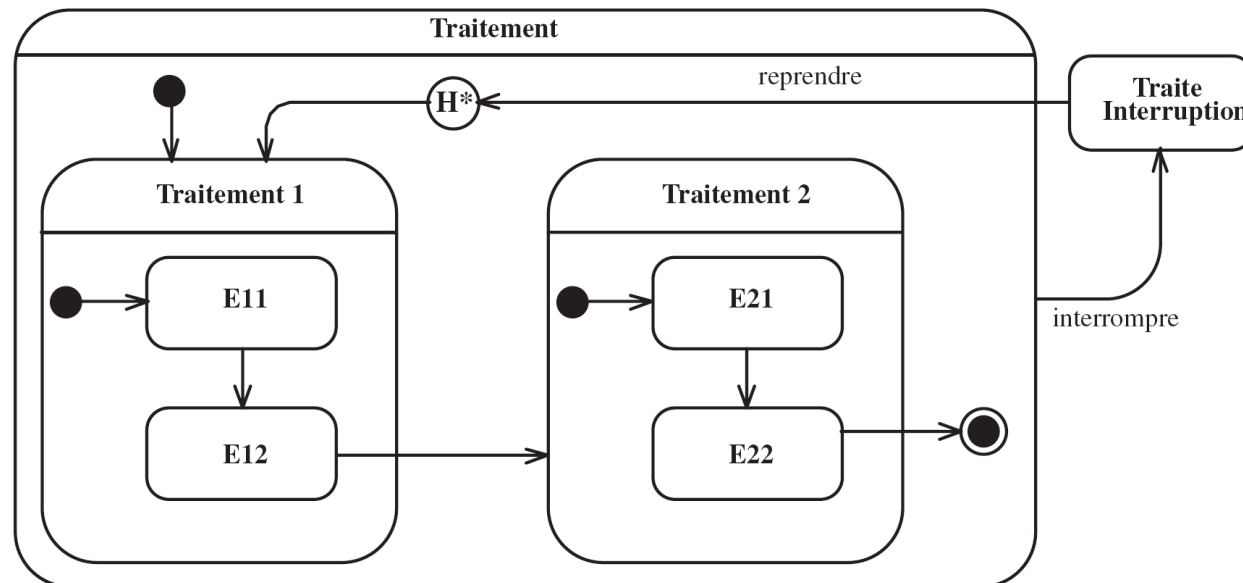
Etats composites et transitions internes

- Depuis Etat1 1, quand event survient
 - On produit la séquence d'activités QuitterE1, EntrerE2, EntrerE21, initialiser, EntrerE22 ;
 - L'objet est dans Etat22.



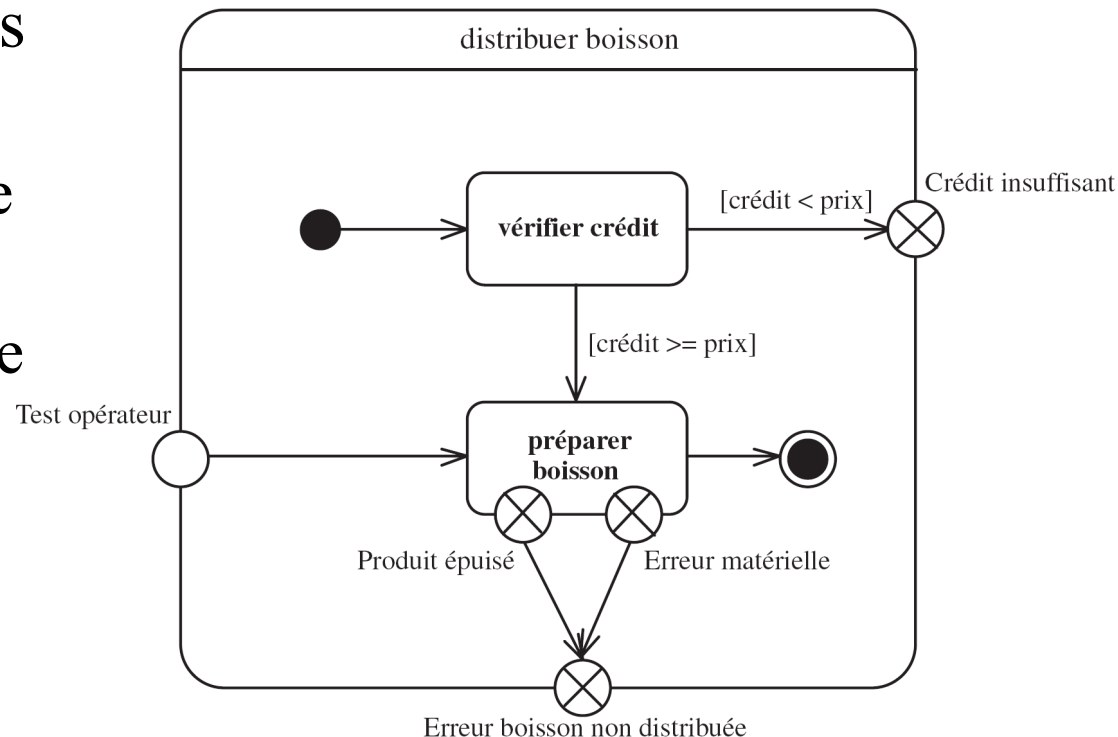
Historique

- Un pseudo-état historique est noté par un H cerclé.
- Une transition ayant pour cible le pseudo-état historique est équivalente à une transition qui a pour cible le dernier état visité dans la région contenant le H.
- H^* désigne un historique profond, cad un historique pour tous les niveaux d'imbrication.



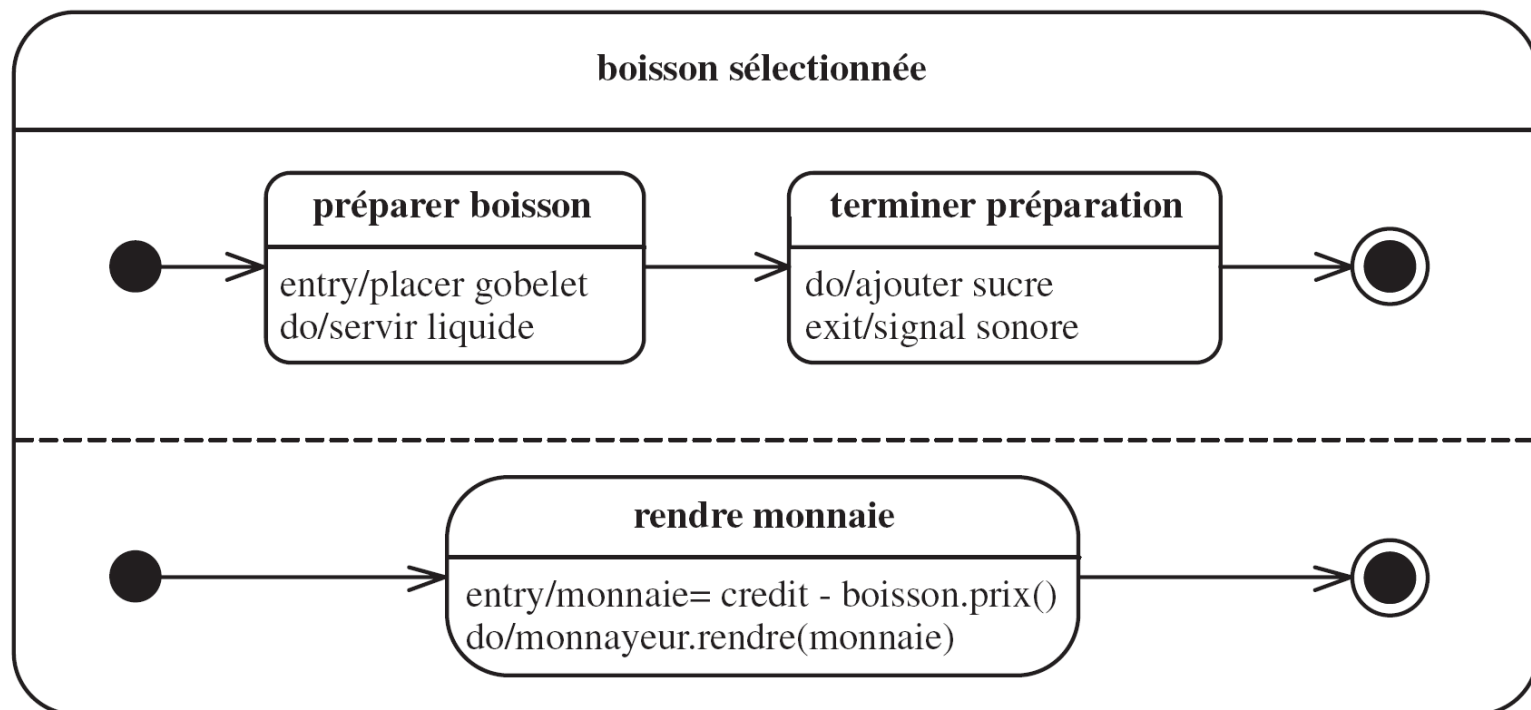
Interface des états composites

- Pour pouvoir représenter un sous état indépendamment d'un macro-état, on a recours à des *points de connexion*.
 - Avec un X pour les points de sortie
 - Vides pour les points d'entrée
- Ces interfaces permettent d'abstraire les sous-états des macro-états (réutilisabilité).



Etat concurrent

- Avec un séparateur en pointillés, on peut représenter plusieurs automates s'exécutant indépendamment.
- Un objet peut alors être simultanément dans plusieurs *états concurrents*.



Transition concurrente

- Une transition *fork* correspond à la création de deux états concurrentes.
- Une transition *join* correspond à une barrière de synchronisation qui supprime la concurrence.
 - Pour pouvoir continuer leur exécution, toutes les tâches concurrentes doivent préalablement être prêtes à franchir la transition.

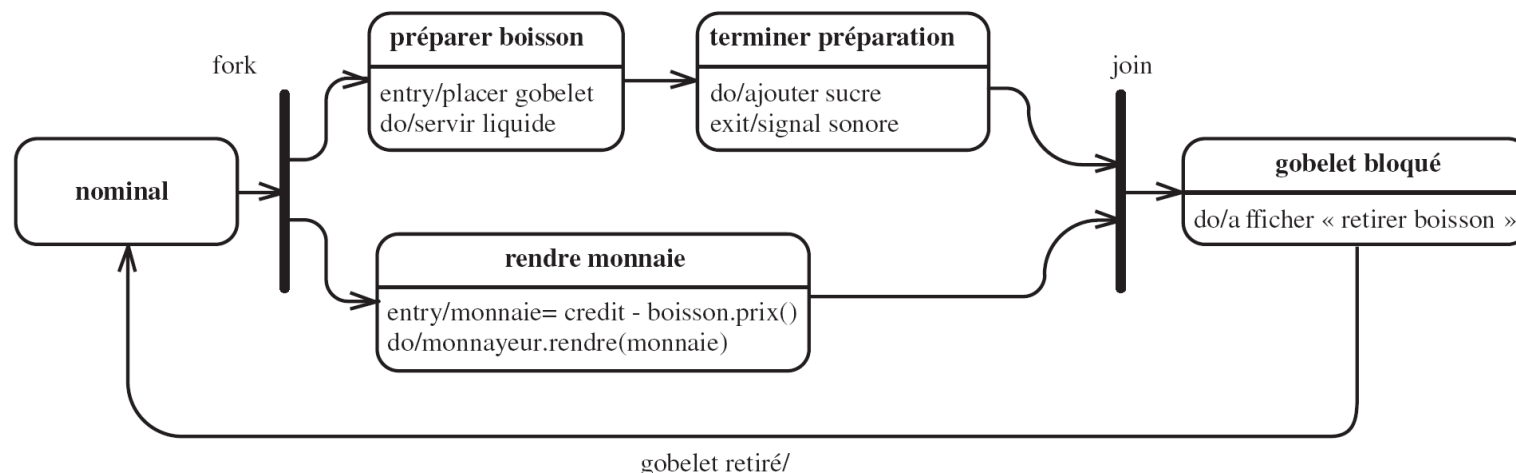
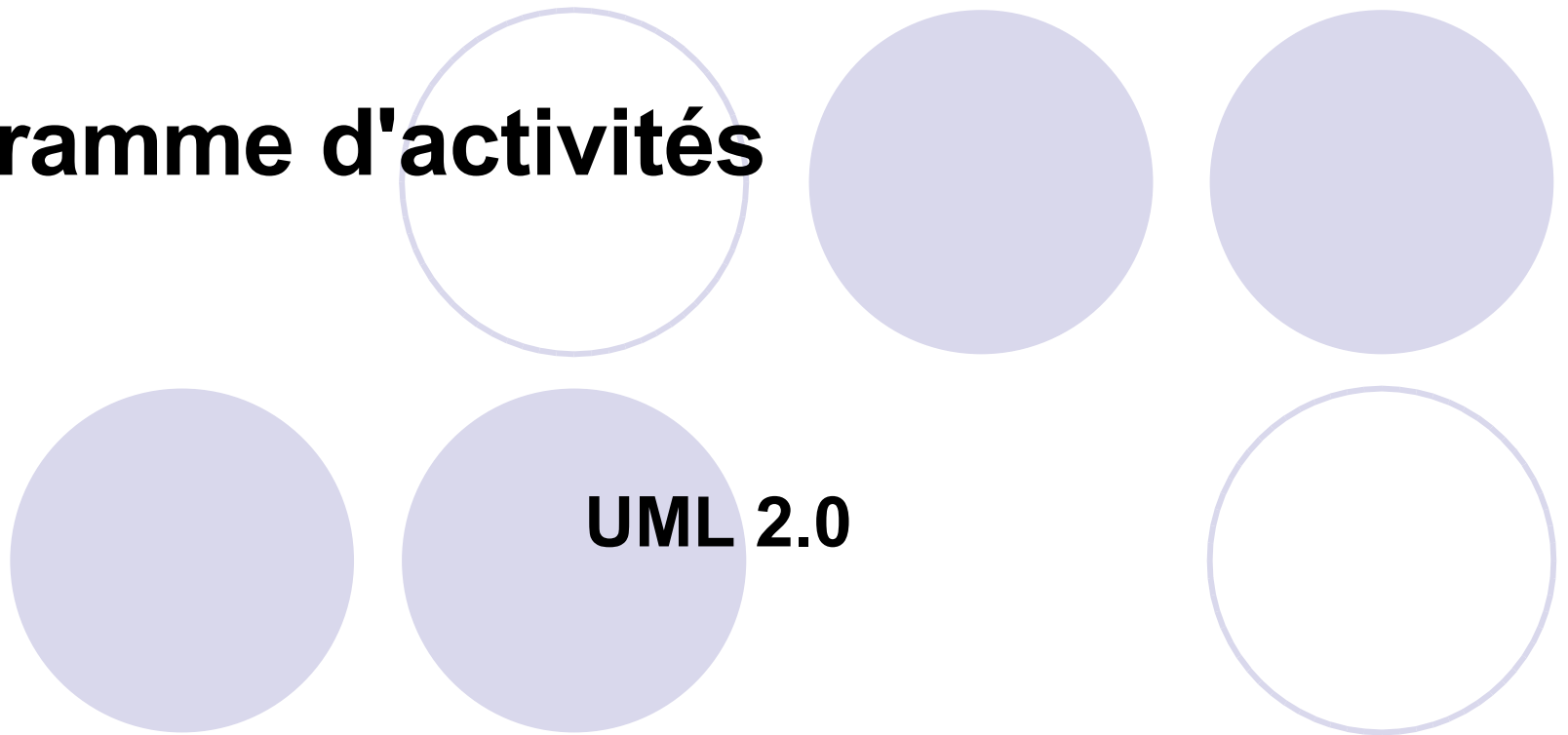


Diagramme d'activités





Modélisation des traitements

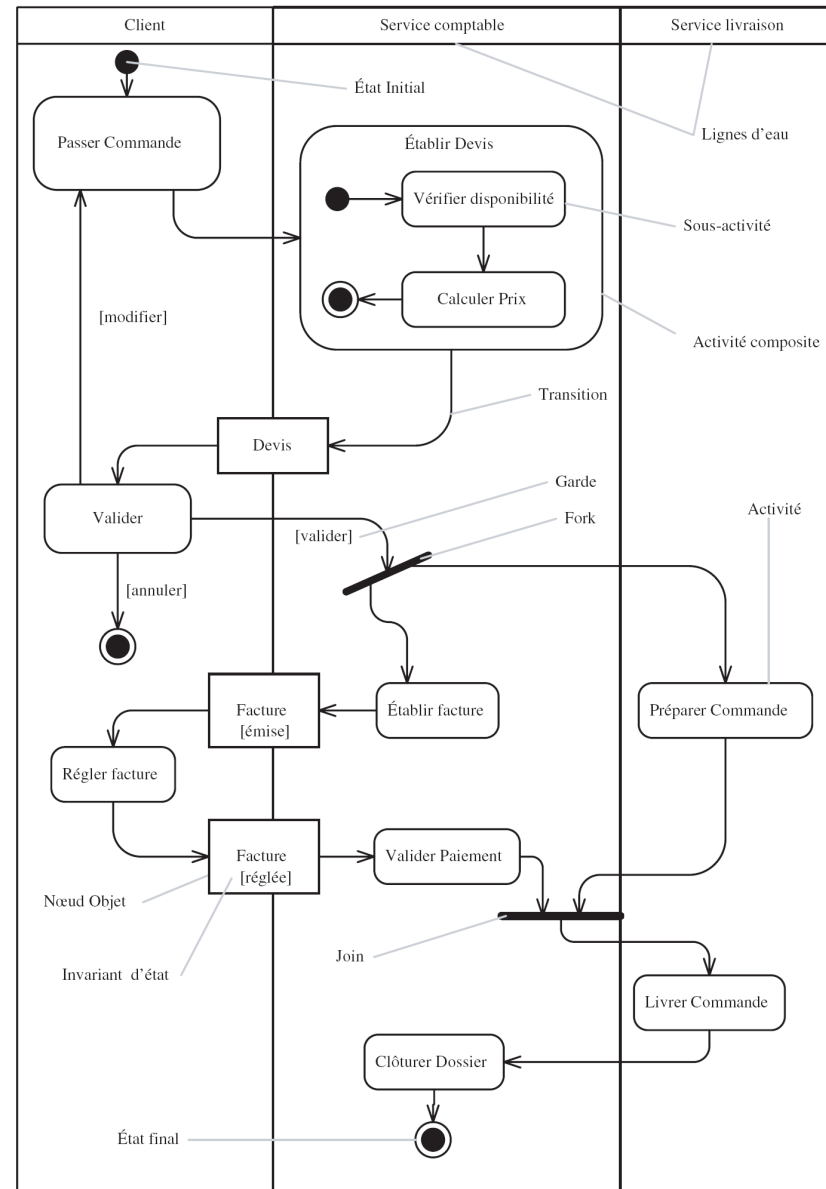
- Les *diagrammes d'activité* offrent une manière graphique et non ambiguë pour modéliser les traitements.
 - Comportement d'une méthode
 - Déroulement d'un cas d'utilisation
- Une *activité* représente une exécution d'un mécanisme, un déroulement d'étapes séquentielles. Le passage d'une activité à l'autre est matérialisé par une *transition*.
- Ces diagrammes sont assez semblables aux états-transitions mais avec une interprétation différente.

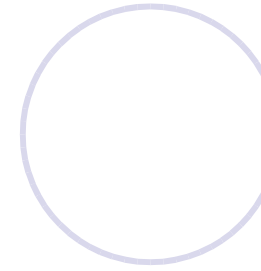
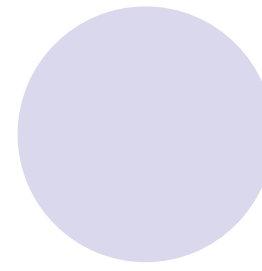
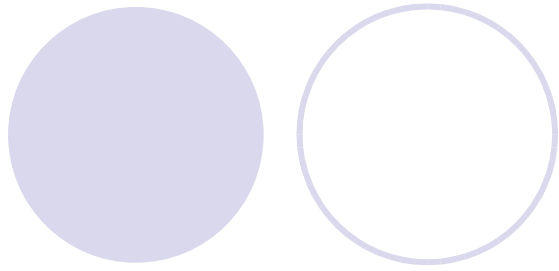


Une vision transversale des traitements

- Les diagrammes d'états-transitions sont définis pour chaque classeur et n'en font pas intervenir plusieurs.
- A l'inverse, les diagrammes d'activité permettent une description s'affranchissant (partiellement) de la structuration de l'application en classeurs.
- La vision des diagrammes d'activités se rapproche des langages de programmation impérative (C, C++, Java)
 - Les états représentent des calculs
 - Il n'y a pas d'événement externes mais des attentes de fins de calculs
 - Il peut y avoir de la concurrence entre activités

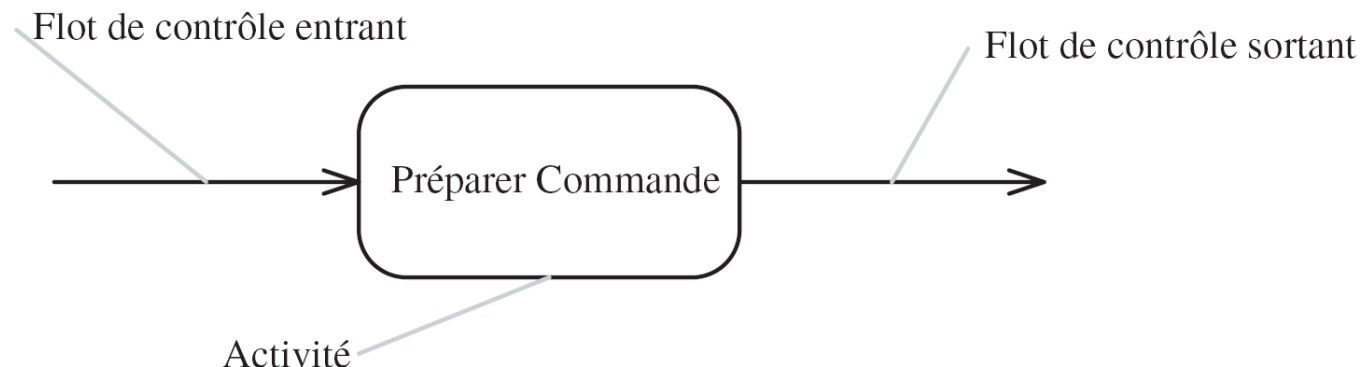
Exemple de diagramme d'activités

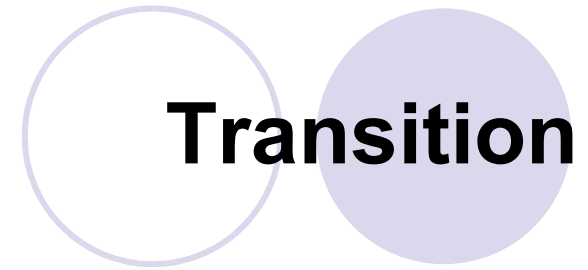
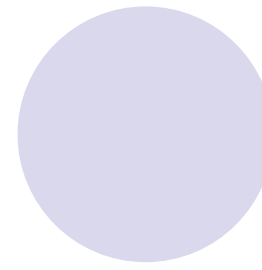
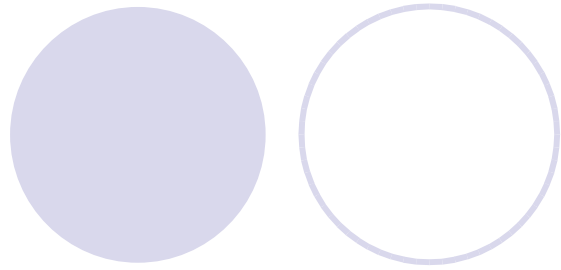




Activité

- Les activités décrivent un traitement.
 - Le flot de contrôle reste dans l'activité jusqu'à ce que les traitements soient terminés.
 - On peut définir des variables locales à une activité et manipuler les variables accessibles depuis le contexte de l'activité (classe contenante en particulier).
 - Les activités peuvent être imbriquées hiérarchiquement : on parle alors d'activités composites.





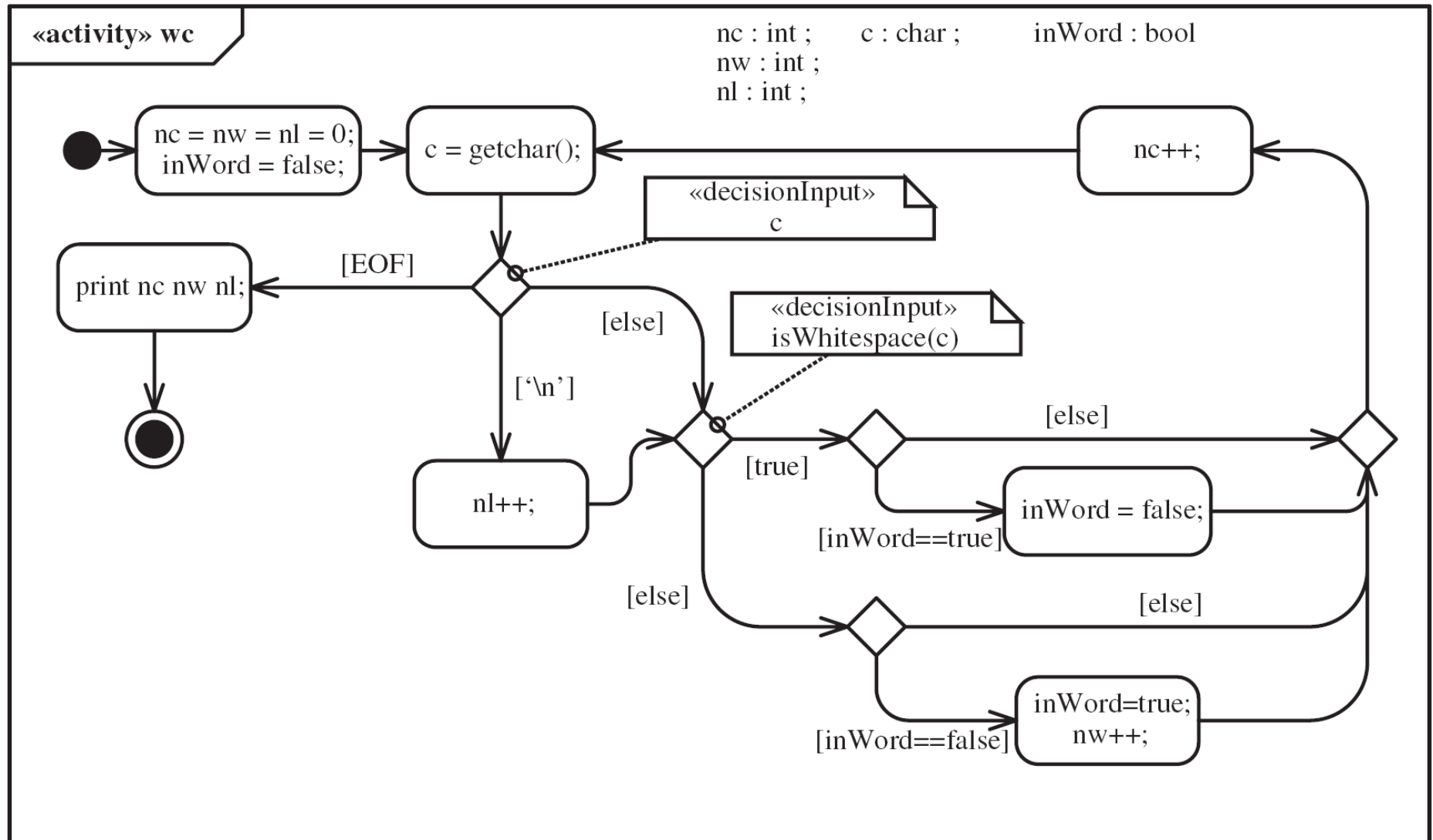
- Les *transitions* sont représentées par des flèches pleines qui connectent les activités entre elles.
 - Elles sont déclenchées dès que l'activité source est terminée.
 - Elles provoquent automatiquement le début immédiat de la prochaine activité à déclencher (l'activité cible).
 - Contrairement aux activités, les transitions sont franchies de manière atomique, en principe sans durée perceptible.
- Les transitions spécifient l'enchaînement des traitements et définissent le flot de contrôle.



Structure de contrôle conditionnelle

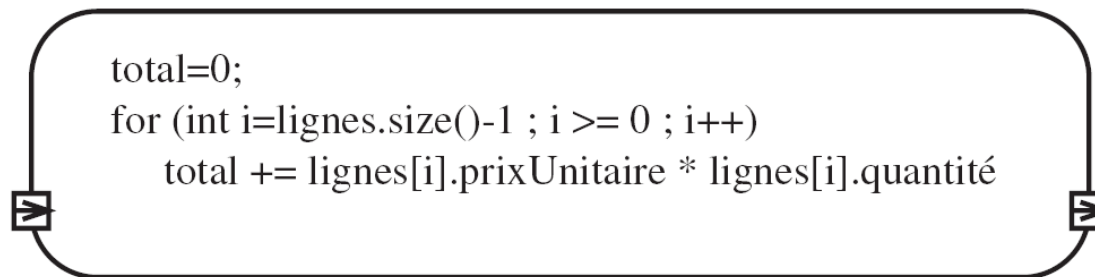
- Expression de conditions au moyen de transitions munies de *gardes conditionnelles*
 - Ces transitions ne peuvent être empruntées que si la garde est vraie.
 - On dispose d'une clause *[else]* qui est validée si et seulement si toutes les autres gardes des transitions ayant la même source sont fausses.
- Les conditions sont notées entre crochets
- Pour mieux mettre en évidence un branchement conditionnel, on peut utiliser les points de choix (losanges).
 - Les points de choix expriment un aiguillage du flot de contrôle.

Exemples de structures conditionnelles



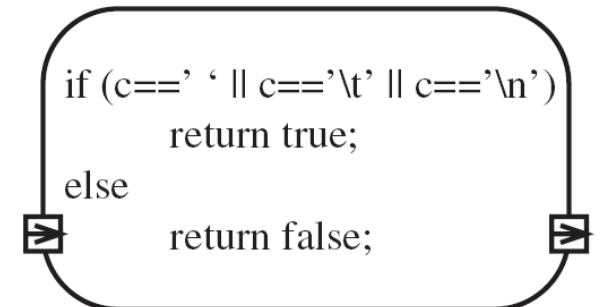
Activités structurées

- Les activités structurées utilisent les structures de contrôle usuelles (conditionnelles et boucle) à travers une syntaxe qui dépend de l'outil.
 - La syntaxe précise de ces annotations pas définie dans la norme UML.
 - Dans une activité structurée, on définit les arguments d'entrée et les sorties par des flèches encadrées.



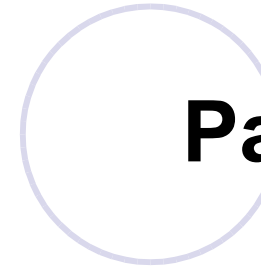
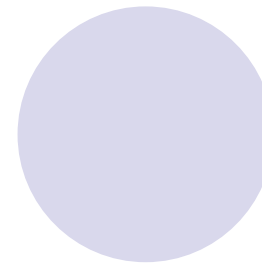
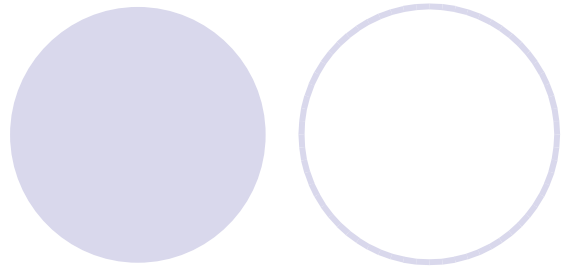
lignes : vector<ligneCommande>

total:float



c : char

bool

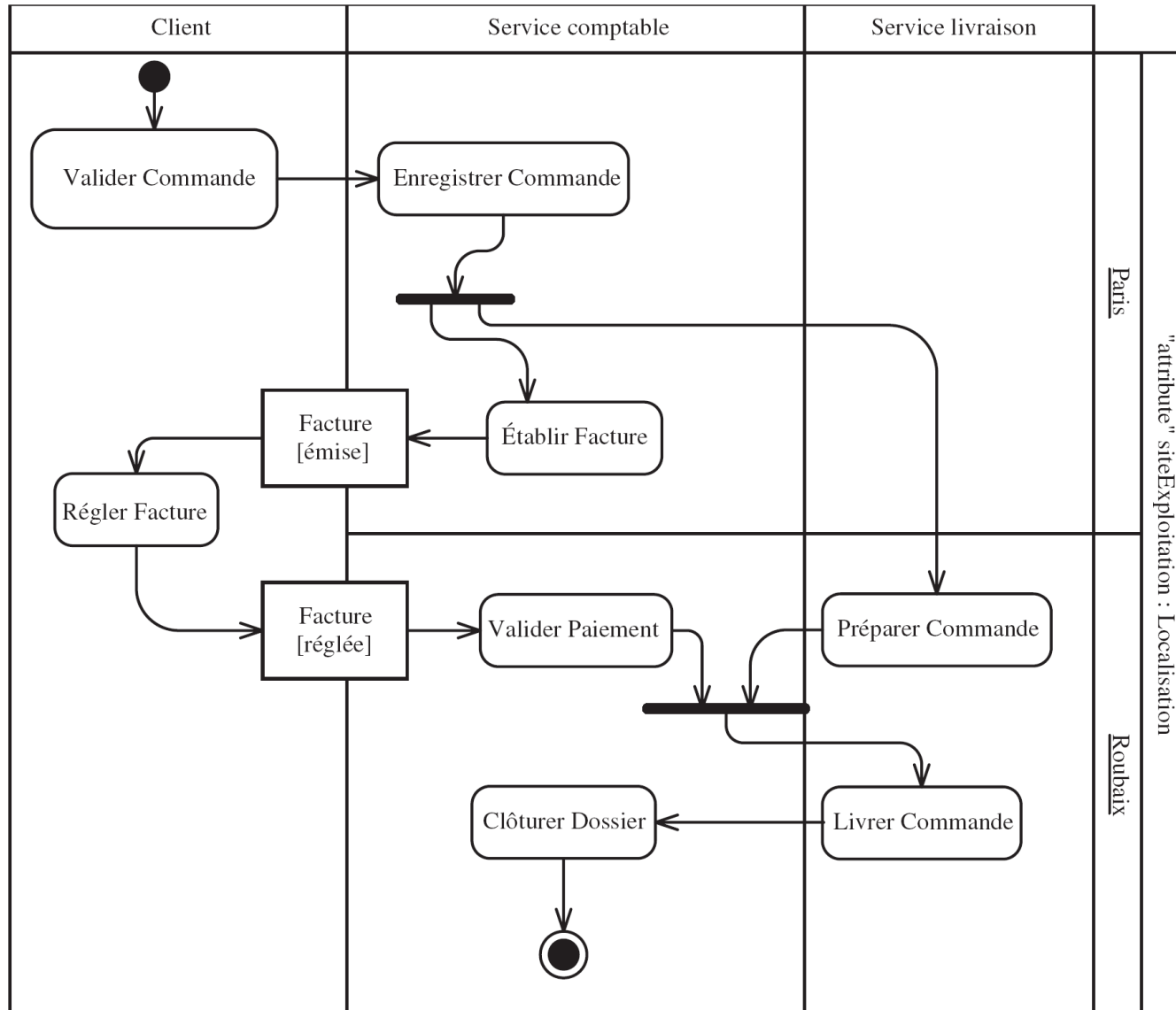


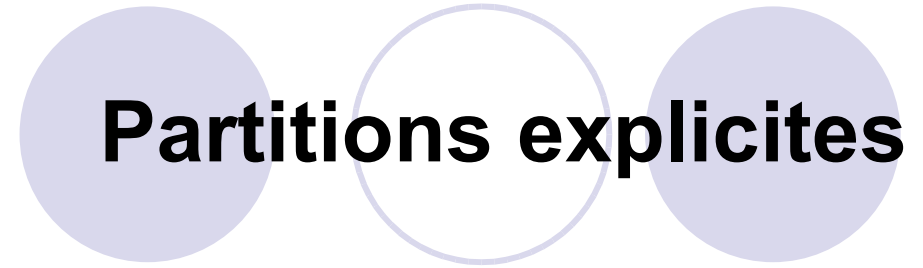
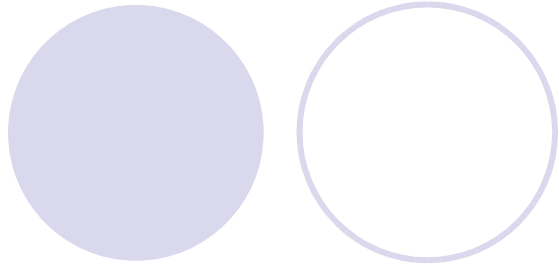
Partitions

- Pour modéliser un traitement mettant en oeuvre plusieurs classeurs, on peut spécifier le classeur responsable de chaque activité.
- Les *partitions* permettent d'attribuer les activités à des éléments particuliers du modèle.
 - Une partition peut elle-même être décomposée en sous-partitions.

| «external» | «attribute» libelléService : Service | |
|------------|--------------------------------------|--------------------------|
| Client | <u>Service comptable</u> | <u>Service livraison</u> |
| | | |

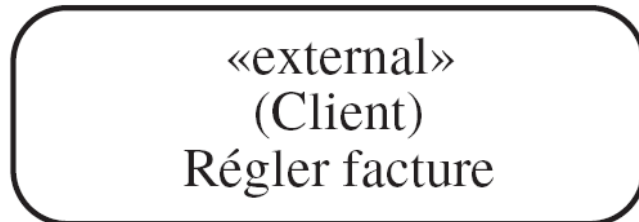
Partitions multidimensionnelles





Partitions explicites

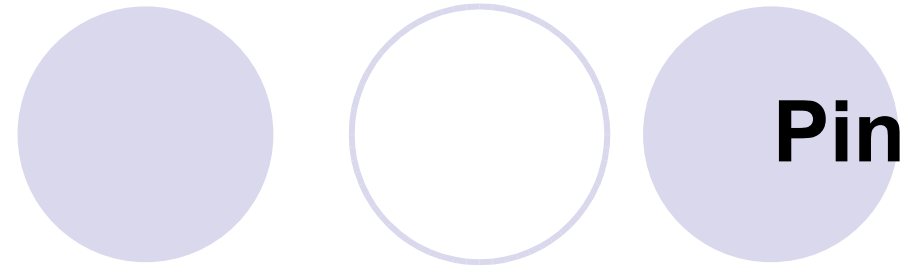
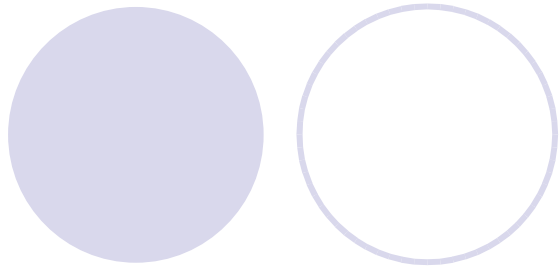
- Cette notation est moins encombrante graphiquement
- Toutefois, elle met moins bien en valeur l'appartenance de groupes d'activités à un même conteneur.



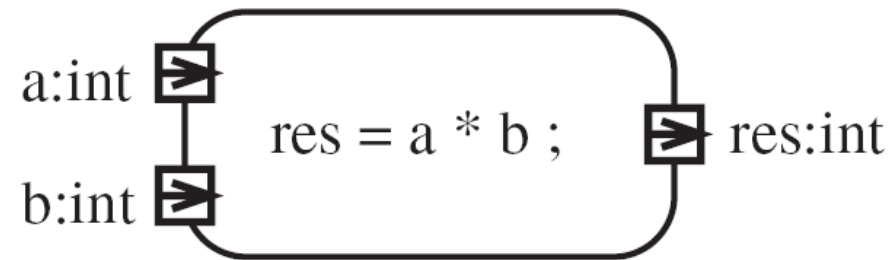


Arguments et valeurs retournées

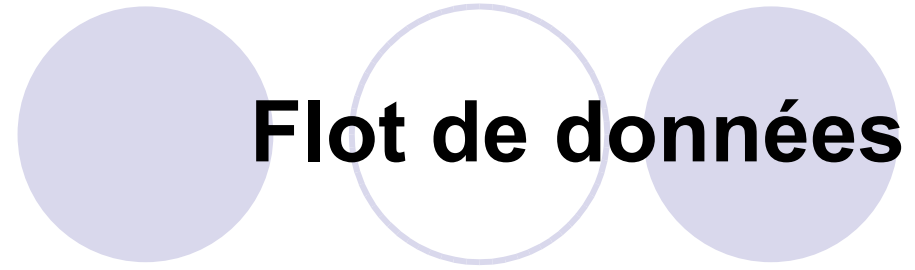
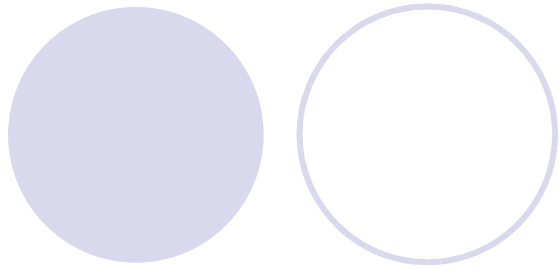
- Les diagrammes d'activités présentés jusqu'ici montrent bien le comportement du flot de contrôle.
- Or le flot de données n'apparaît pas clairement.
 - Si une activité est bien adaptée à la description d'une opération d'un classeur, il faut un moyen de spécifier les arguments et valeurs de retour de l'opération. C'est le rôle
 - des *pins*,
 - des *noeuds*,
 - des *flots d'objets* associés.



- Un *pin* représente un point de connexion pour une action.
 - L'action ne peut débuter que si l'on affecte une valeur à chacun de ses pins d'entrée.
 - Les valeurs sont passées par copie.

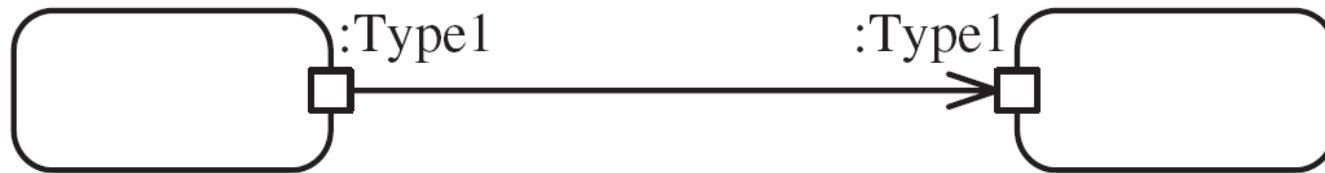


- Quand elle se termine, une valeur doit être affectée à chacun des pins de sortie.



Flot de données

- Un *flot d'objets* permet de passer des données d'une activité à une autre.
 - De fait, un arc qui a pour origine et destination un pin correspond à un flot de données.

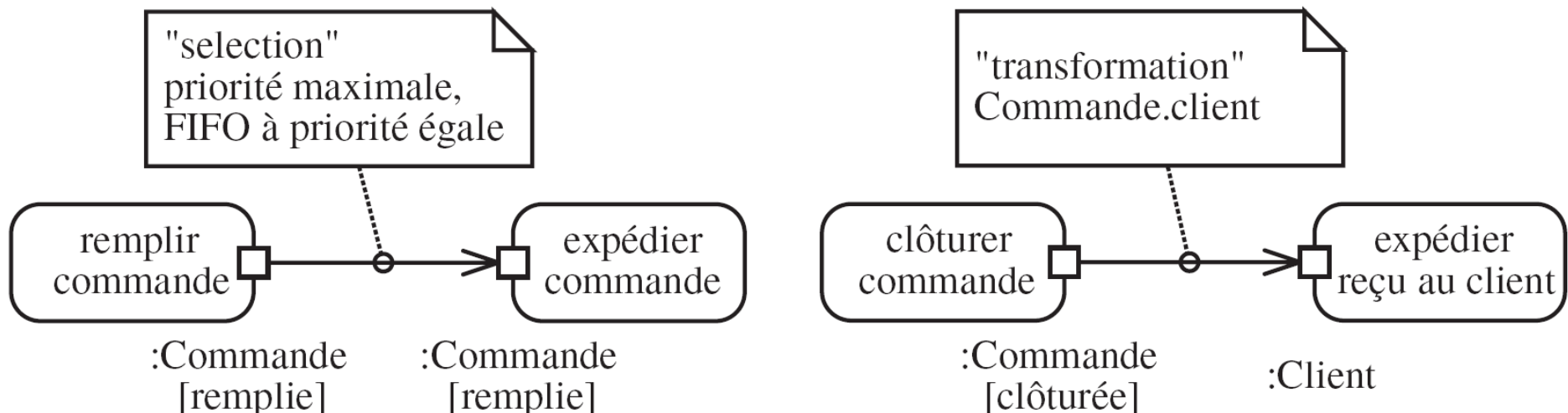


- Un *noeud d'objets* permettent de mieux mettre en valeur les données.
 - C'est un conteneur typé qui permet le transit des données.



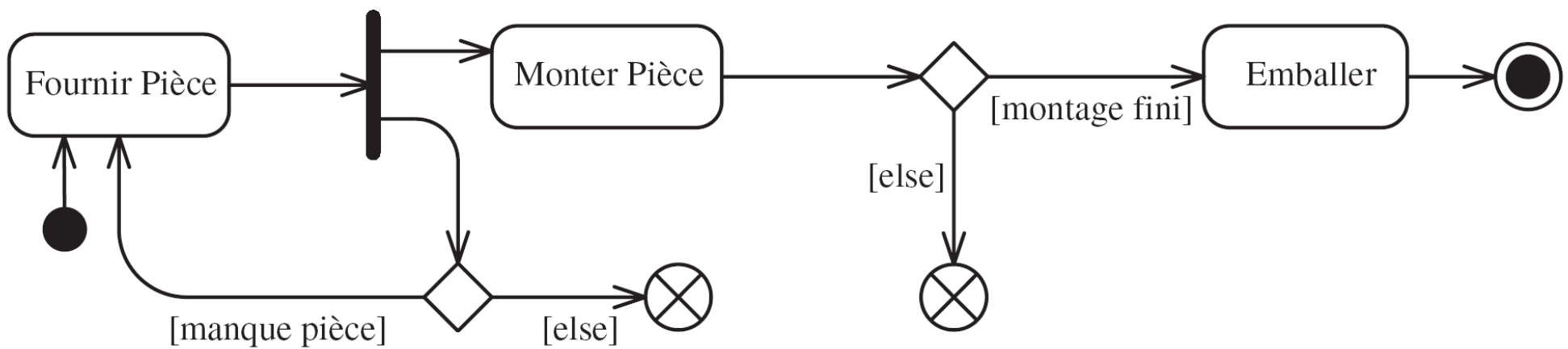
Annotation des flots de données

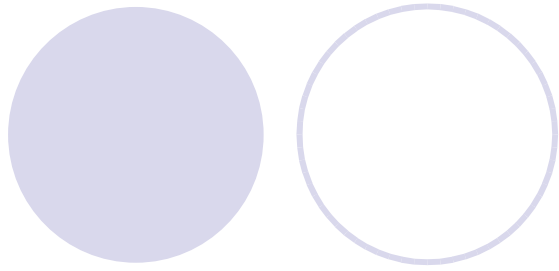
- Un flot d'objets peut porter une étiquette mentionnant deux annotations particulières :
 - « *transformation* » indique une interprétation particulière de la donnée transmise par le flot.
 - « *selection* » indique l'ordre dans lequel les objets sont choisis dans le noeud pour le quitter.



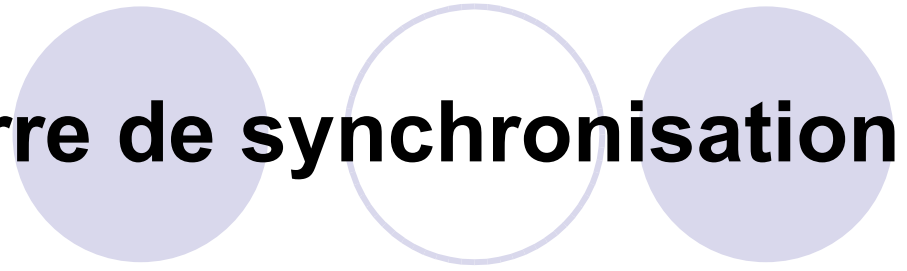
Concurrence

- Les diagrammes d'activités permettent en principe de représenter des activités séquentielles.
- Néanmoins, on peut représenter des *activités concurrentes* avec :
 - Les barres de synchronisation,
 - Les noeuds de contrôle de type « flow final ».





Barre de synchronisation



- Plusieurs transitions peuvent avoir pour source ou pour cible une *barre de synchronisation*.
 - Lorsque la barre de synchronisation a plusieurs transitions en *sortie*, on parle de transition de type *fork* qui correspond à une *duplication du flot de contrôle* en plusieurs flots indépendants.
 - Lorsque la barre de synchronisation a plusieurs transitions en *entrée*, on parle de transition de type *join* qui correspond à un *rendez-vous* entre des flots de contrôle.
- Pour plus de commodité, il est possible de fusionner des barres de synchronisation de type *join* et *fork*.
 - On a alors plusieurs transitions entrantes et sortantes sur une même barre.

Noeud de contrôle de type « flow final »

- Un flot de contrôle qui atteint un noeud de contrôle de type « *flow final* » est détruit.
 - Les autres flots de contrôle ne sont pas affectés.

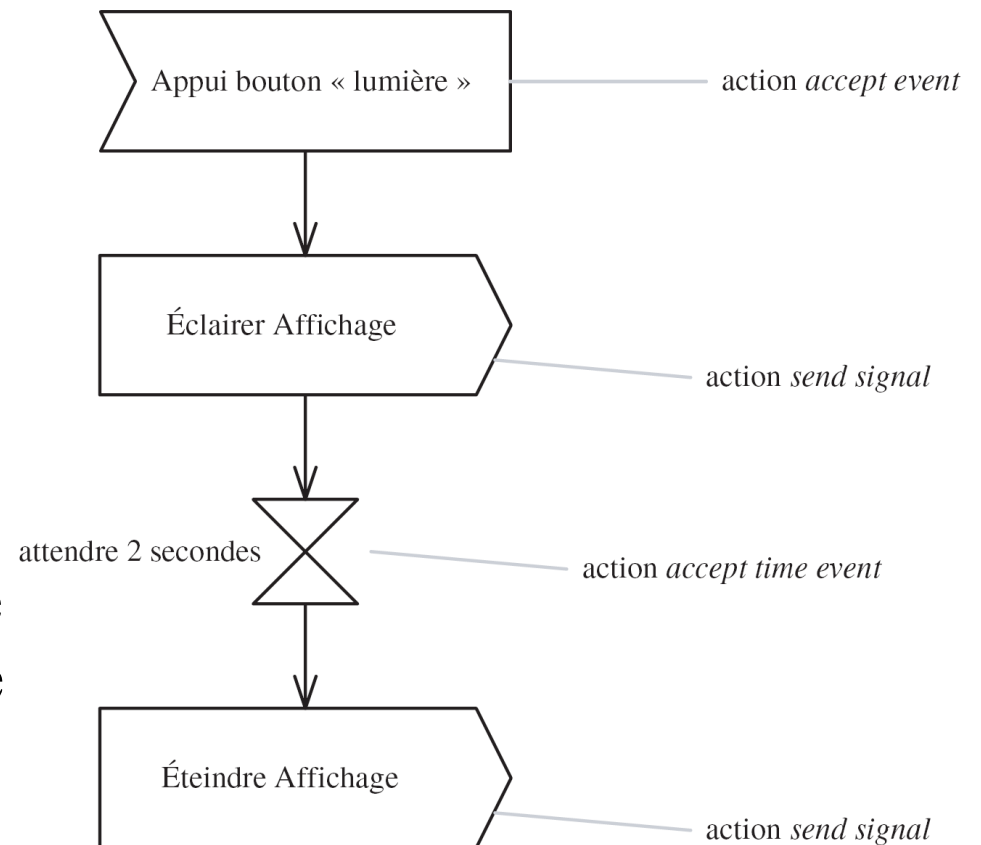


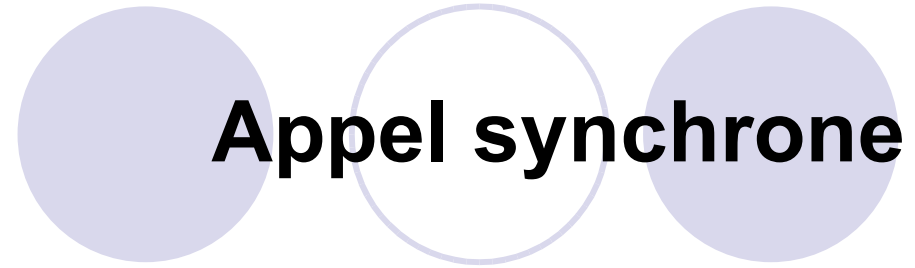
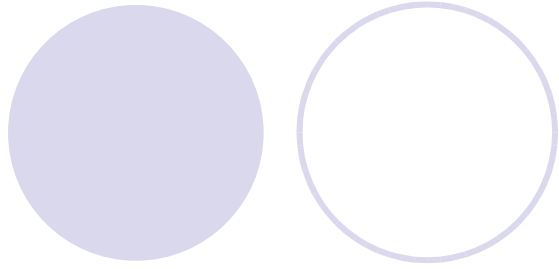
- Ce type de noeud est moins « fort » qu'un noeud de contrôle *final*.
 - Dans ce cas, tous les autres flots de contrôle de l'activité sont interrompus et détruits.



Actions de communication

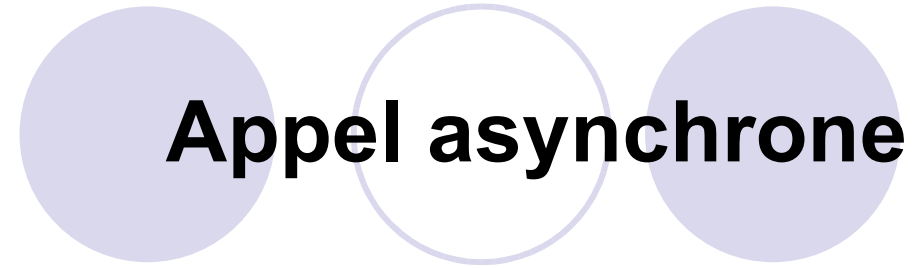
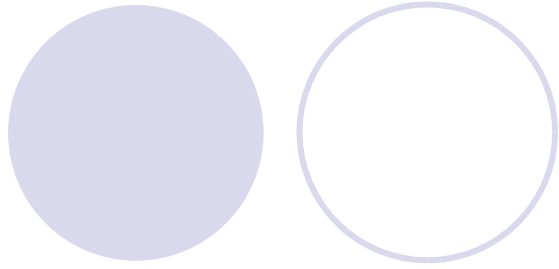
- Les actions de communication gèrent
 - le passage et le retour de paramètres,
 - les mécanismes d'appels d'opérations synchrones et asynchrones.
- Les actions de communications peuvent être employés comme des activité dans les diagrammes d'activité.





Appel synchrone

- Les actions de type *call* correspondent à des appels de procédure ou de méthode.
 - *Call operation* correspond à l'appel d'une opération sur un classeur.
 - *Call behavior* correspond à l'invocation d'un comportement spécifié à l'aide d'un diagramme UML
 - Dans les deux cas, il est possible de spécifier des arguments et d'obtenir des valeurs en retour.
- Les actions *accept call* et *reply* peuvent être utilisées du côté récepteur pour décomposer la réception de l'appel.

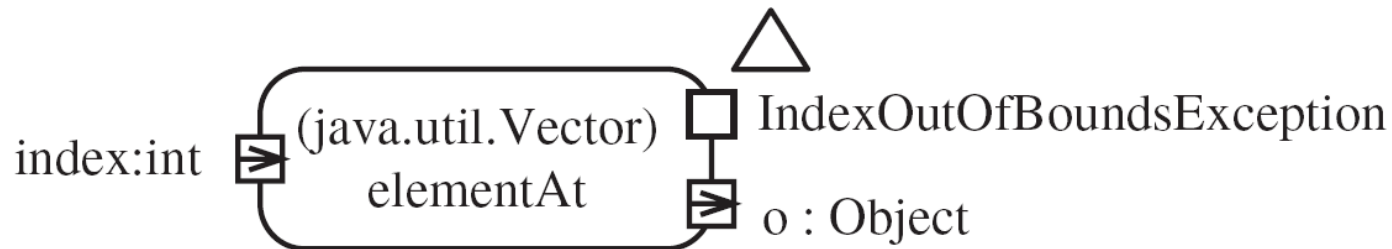


Appel asynchrone

- Les appels asynchrones de type *send* correspondent à des envois de messages asynchrones.
- Le *broadcast signal* permet d'émettre vers plusieurs destinataires à la fois
 - Cette possibilité est rarement offerte par les langages de programmation.
- L'action symétrique côté récepteur est *accept event*, qui permet la réception d'un signal.

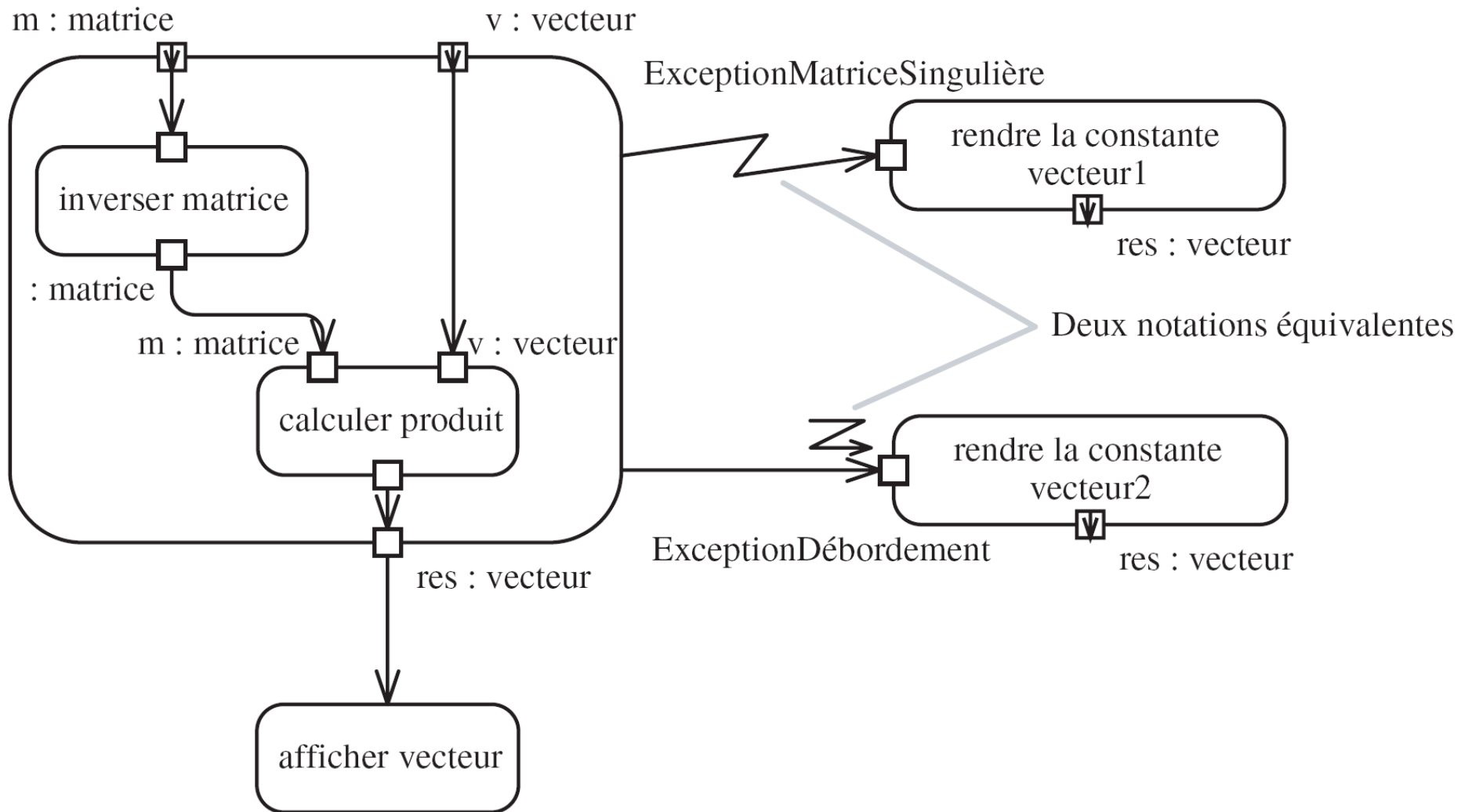
Exceptions

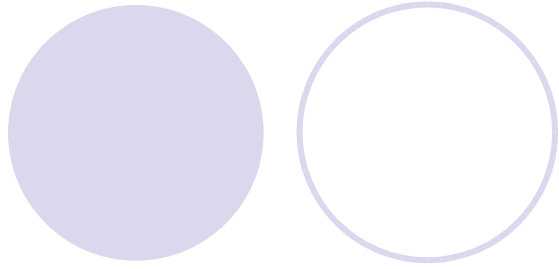
- Les *exceptions* permettent d'interrompre un traitement quand une situation qui dévie du traitement normal se produit. Elles assurent une gestion plus propre des erreurs qui peuvent se produire au cours d'un traitement.
- On utilise des pins d'exception (avec un triangle) un pour gérer l'envoi d'exceptions et la capture d'exceptions.



- Un flot de données correspondant à une exception est matérialisé par une flèche en « zigue zague ».

Exemple de traitement d'exceptions

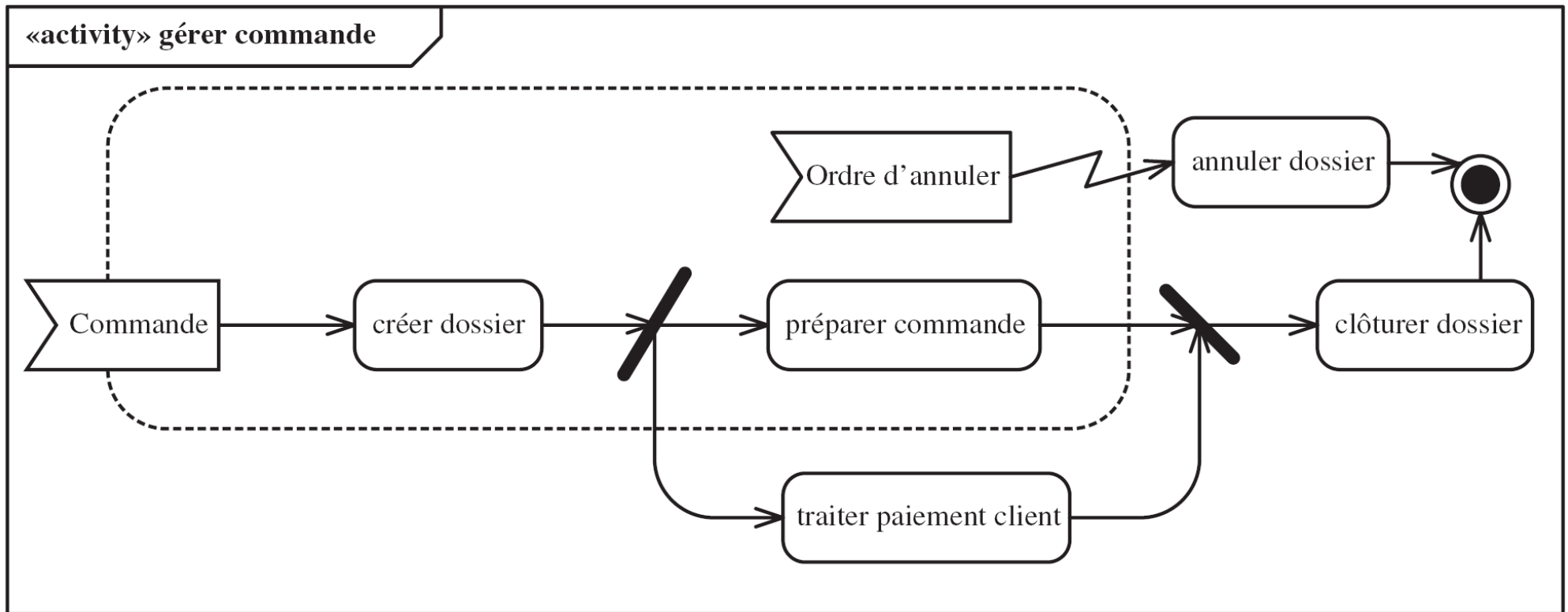


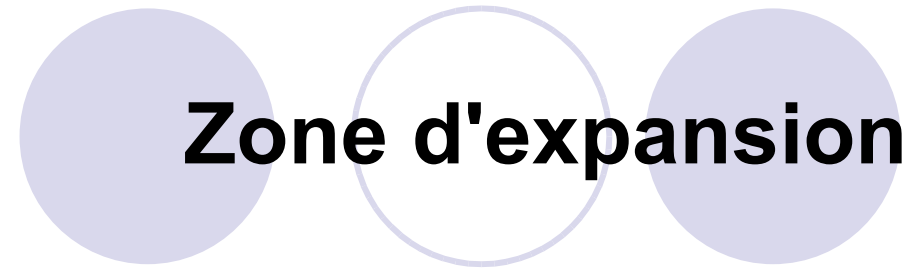
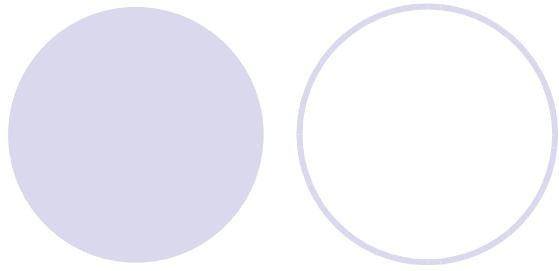


Régions interruptibles

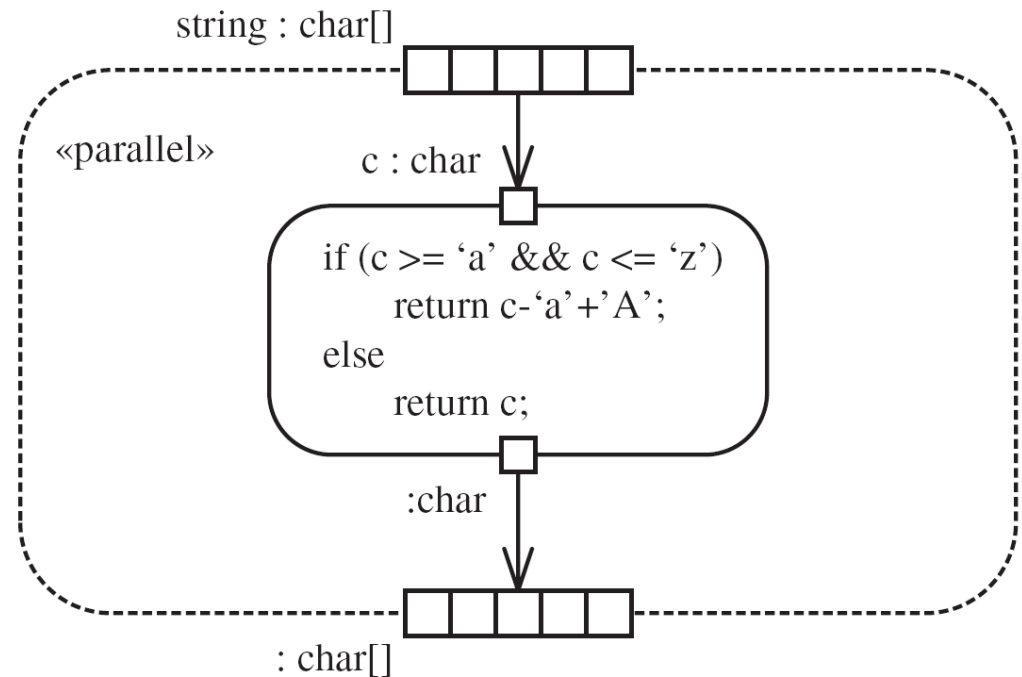
- Ce mécanisme est analogue à celui des interruptions, mais il est moins détaillé.
 - Les régions interruptibles sont mieux adaptées aux phases de modélisation fonctionnelles.
- Une région interruptible est représentée par un cadre arrondi en pointillés.
 - Si l'événement d'interruption se produit, toutes les activités en cours dans la région interruptible sont stoppées
 - Le flot de contrôle suit la flèche en zigzag qui quitte la région.

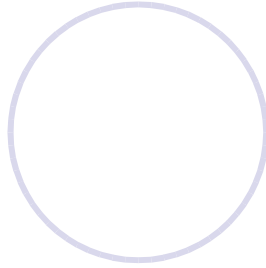
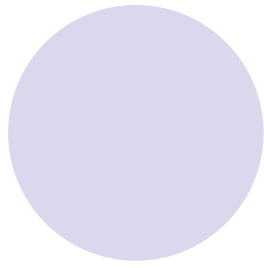
Exemple de région interruptible



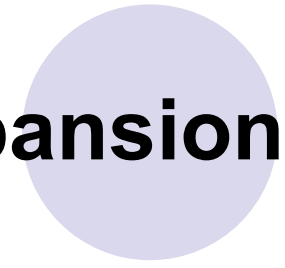
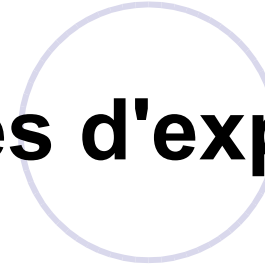


- Une *zone d'expansion* exprime une action répétée sur chaque élément d'une collection.
 - Elle est représentée par un cadre en pointillés
 - L'activité à l'intérieur de la région d'expansion est exécutée une fois pour chaque item de la collection.





Types de zones d'expansion

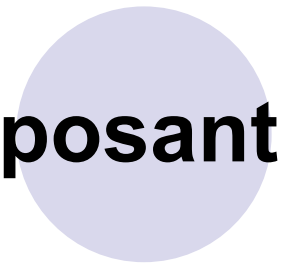
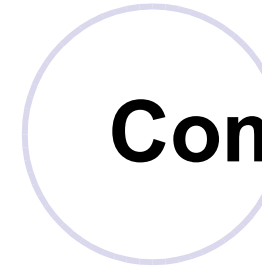
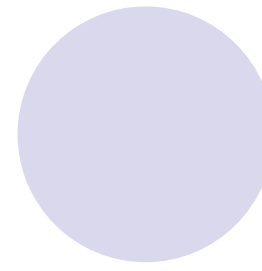
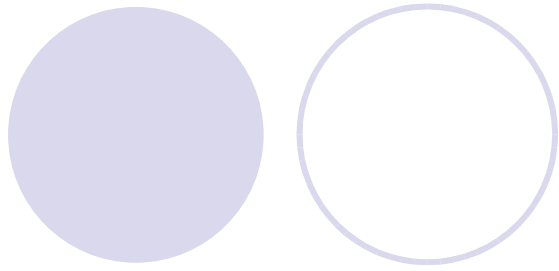


- *Parallel*
 - Les exécutions de l'activité interne sur les éléments de la collection sont indépendantes et peuvent être réalisées dans n'importe quel ordre ou même simultanément.
- *Iterative*
 - Les occurrences d'exécution de l'activité interne doivent s'enchaîner séquentiellement, en suivant l'ordre de la collection d'entrée.
- *Stream*
 - Les éléments de la collection sont passés sous la forme d'un flux de données à l'activité interne, qui doit être adaptée aux traitements de flux.

Diagrammes de composants et de déploiement

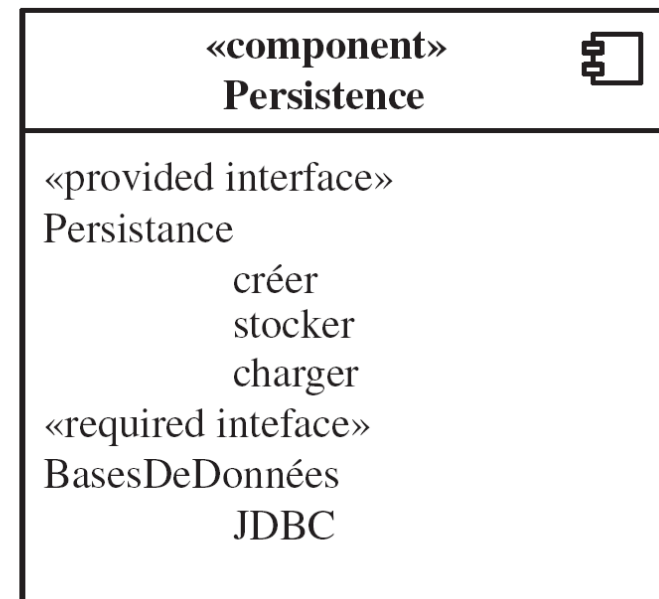


UML 2.0



Composant

- Un *composant* est une unité autonome au sein d'un système ou sous-système.
- C'est un classeur structuré particulier, muni d'une ou plusieurs interfaces requises ou offertes.



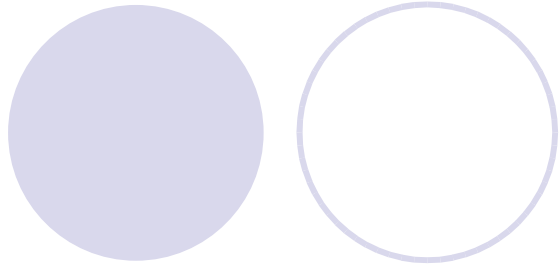
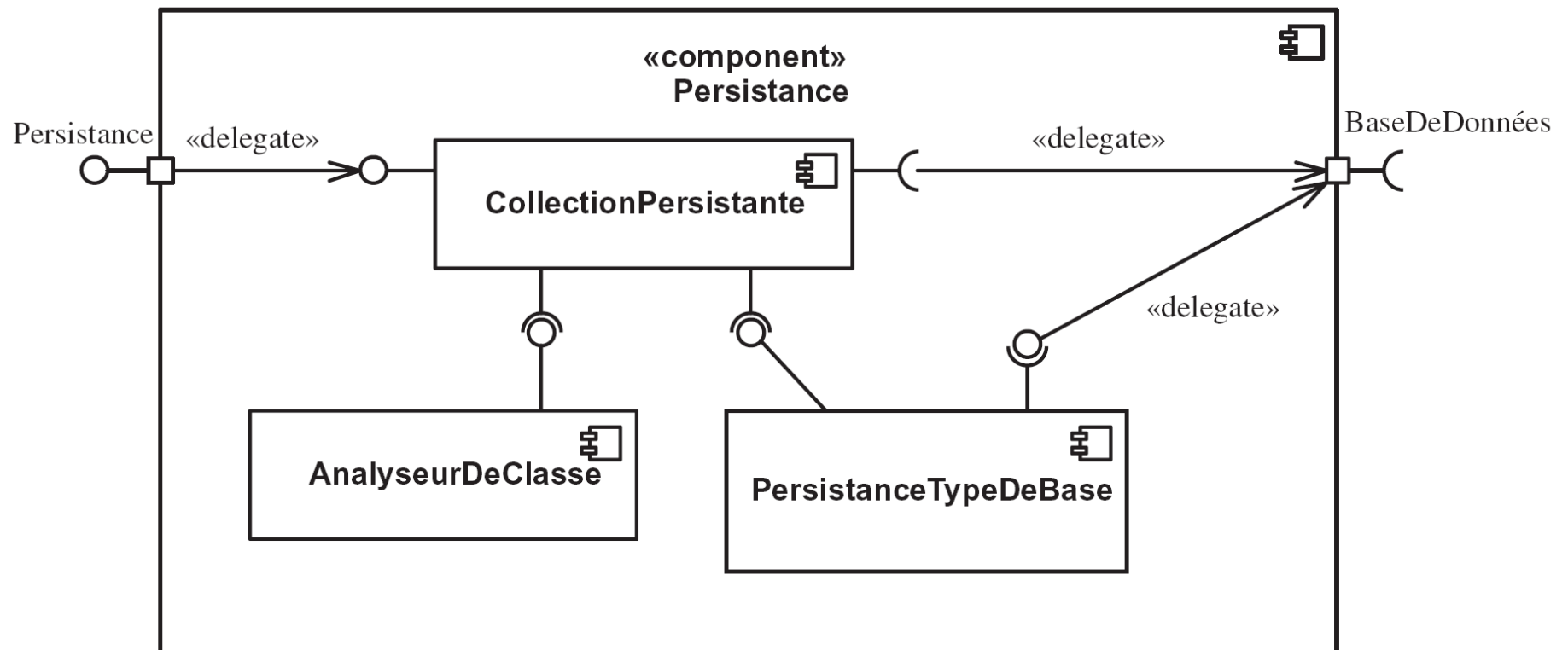


Diagramme de composant



- Les *diagrammes de composants* représentent l'architecture logicielle du système.
 - Les composants peuvent être décrits à l'aide des composants qui les composent, le cas échéant.
 - Les liens entre composants sont spécifiés à l'aide de dépendances entre leurs interfaces.
 - Le « câblage interne » d'un composant est spécifié par les *connecteurs de délégation*. Un tel connecteur connecte un port externe du composant avec un port de l'une de ses sous-composants internes.

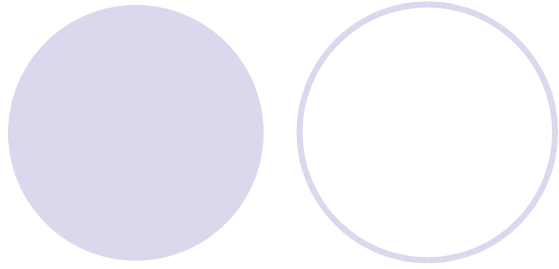
Exemple de modélisation d'un composant



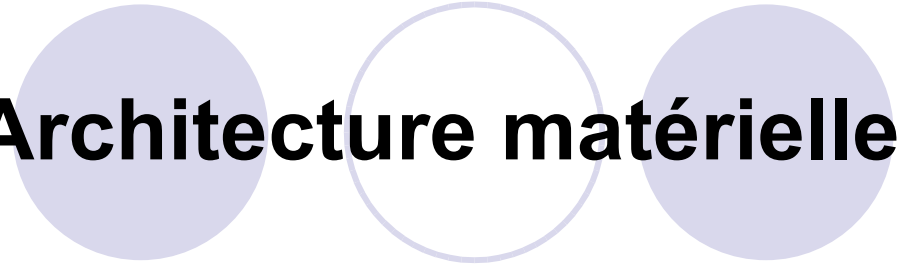


Intérêt des diagrammes de composants

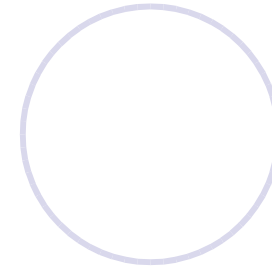
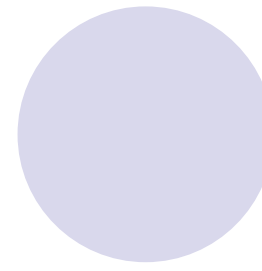
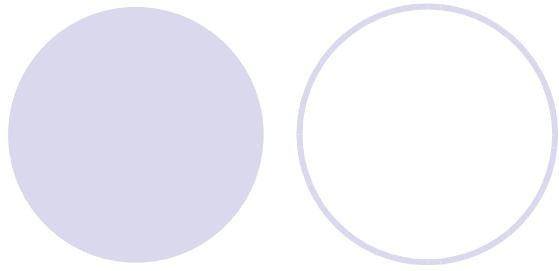
- Les diagrammes de composants permettent de
 - Spécifier l'intégration de briques logicielles tierces (composants EJB, CORBA, COM+ ou .Net, WSDL...) ;
 - Identifier les composants réutilisables.
- Un composant est un espace de noms qu'on peut employer pour organiser les éléments de conception, les cas d'utilisation, les interactions et les artefacts de code.



Architecture matérielle

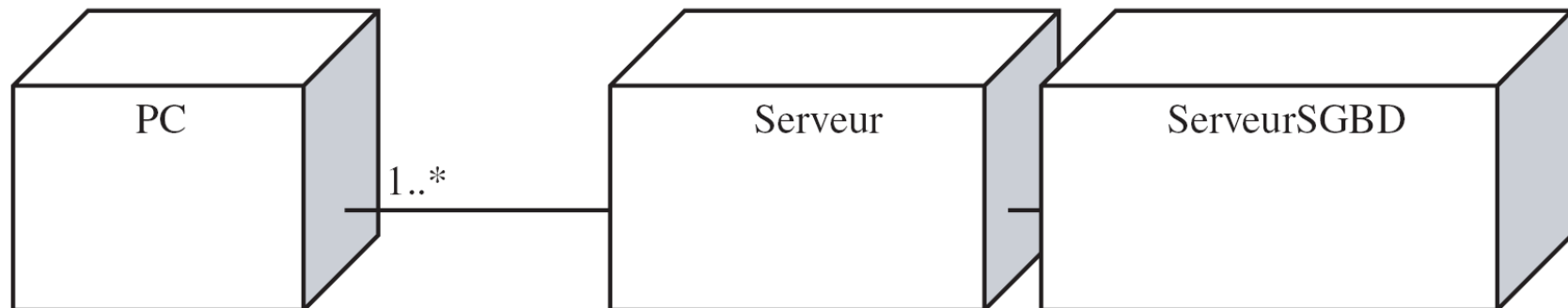


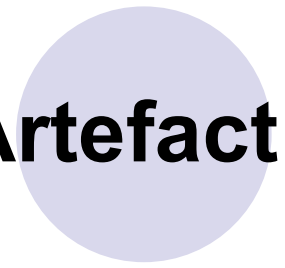
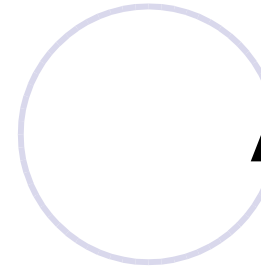
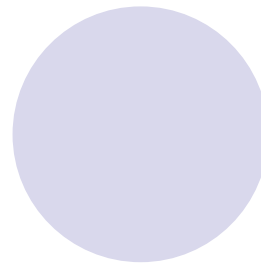
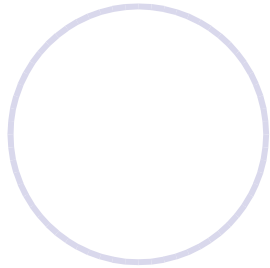
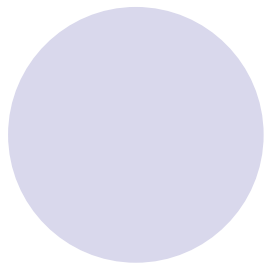
- Au final, un système dernier va s'exécuter sur des ressources matérielles dans un environnement particulier.
- UML permet de représenter un environnement d'exécution ainsi que des ressources physiques (avec les parties du système qui s'y exécutent) grâce aux *diagrammes de déploiement*.
 - L'environnement d'exécution ou les ressources matérielles sont appelés « noeuds ».
 - Les parties d'un système qui s'exécutent sur un noeud sont appelées « artefacts ».



Noeud

- Un *noeud* est une ressource sur laquelle des artefacts peuvent être déployés pour être exécutés.
- C'est un classeur qui peut prendre des attributs.
 - Un noeud se représente par un cube dont le nom respecte la syntaxe des noms de classes.
 - Les noeuds peuvent être associés comme des classes et on peut spécifier des multiplicités.



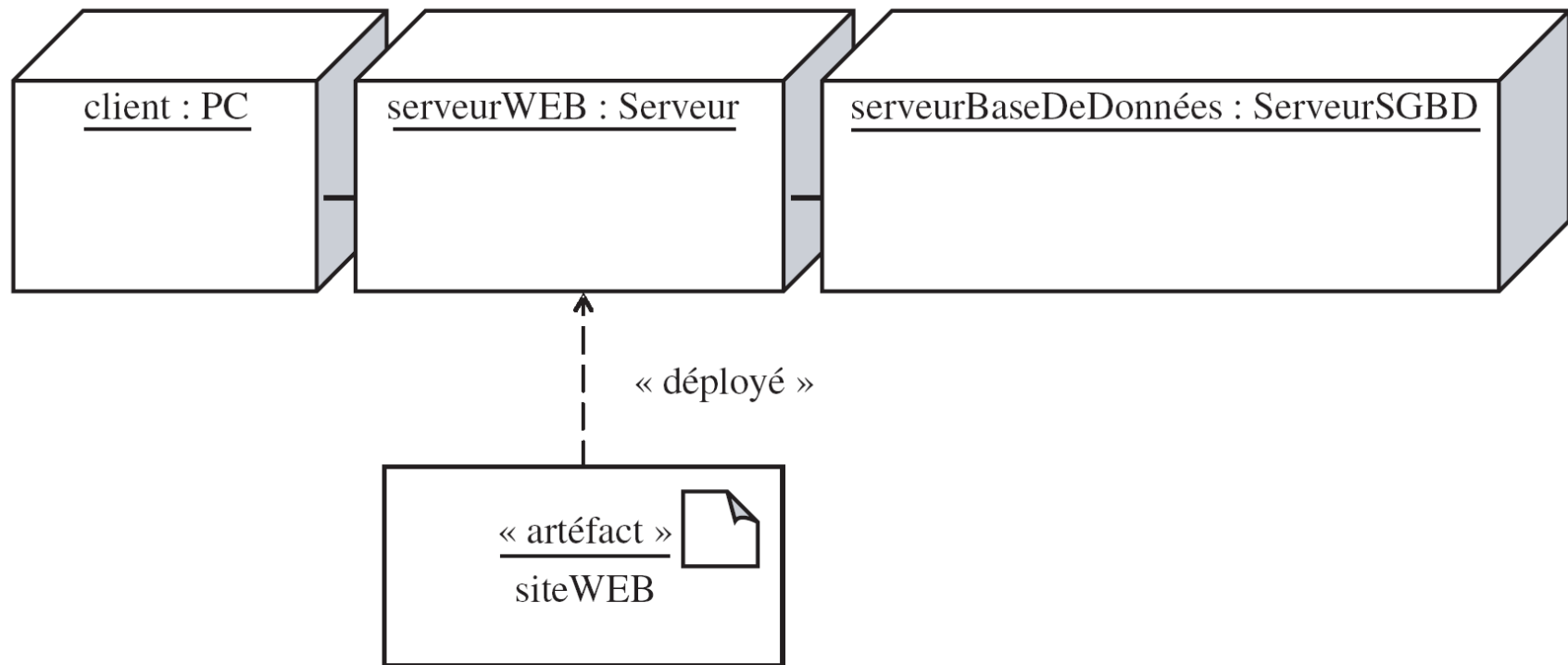


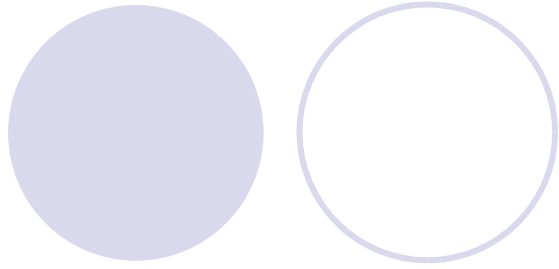
Artefact

- Un *artefact* est la spécification d'une entité physique du monde réel.
- Il se représente comme une classe par un rectangle contenant le mot-clé *artefact* suivi du nom de l'artefact.
 - Un artefact déployé sur un noeud est symbolisé par une flèche en trait pointillé qui porte le stéréotype déployé et qui pointe vers le noeud.
 - L'artefact peut aussi être inclus directement dans le cube représentant le noeud.
- Plusieurs stéréotypes standard existent pour les artefacts : *document*, *exécutable*, *fichier*, *librairie*, *source*.

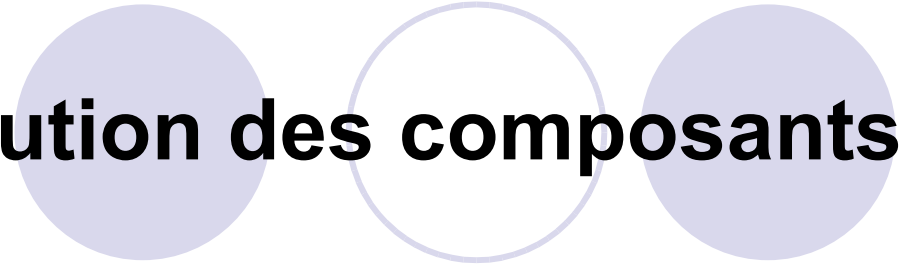
Instantiation de noeuds et d'artefacts

- Les noms des instances de noeuds et d'artefacts sont soulignés.



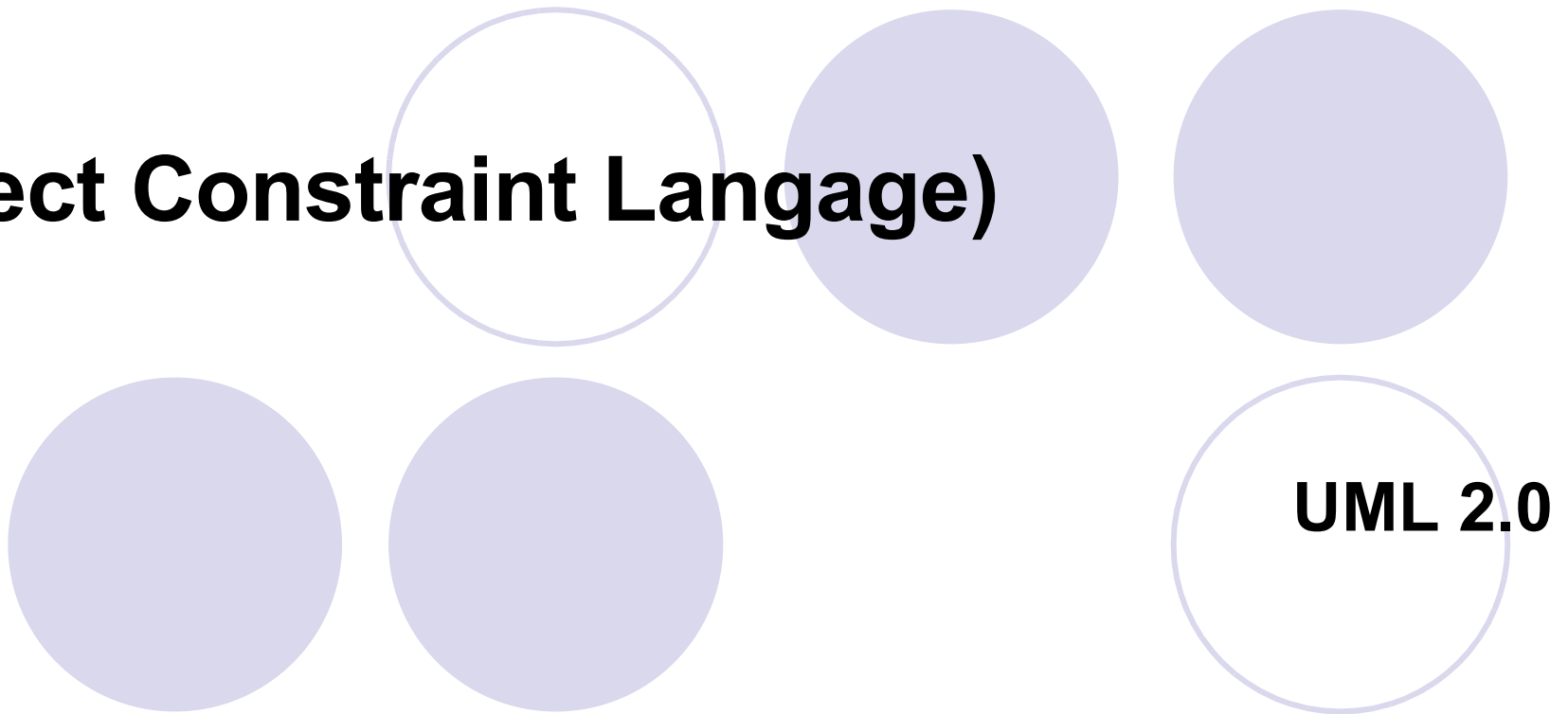


Exécution des composants



- On utilise des flèches de dépendance pour montrer la capacité d'un noeud à prendre en charge un type de composant.

OCL **(Object Constraint Language)**





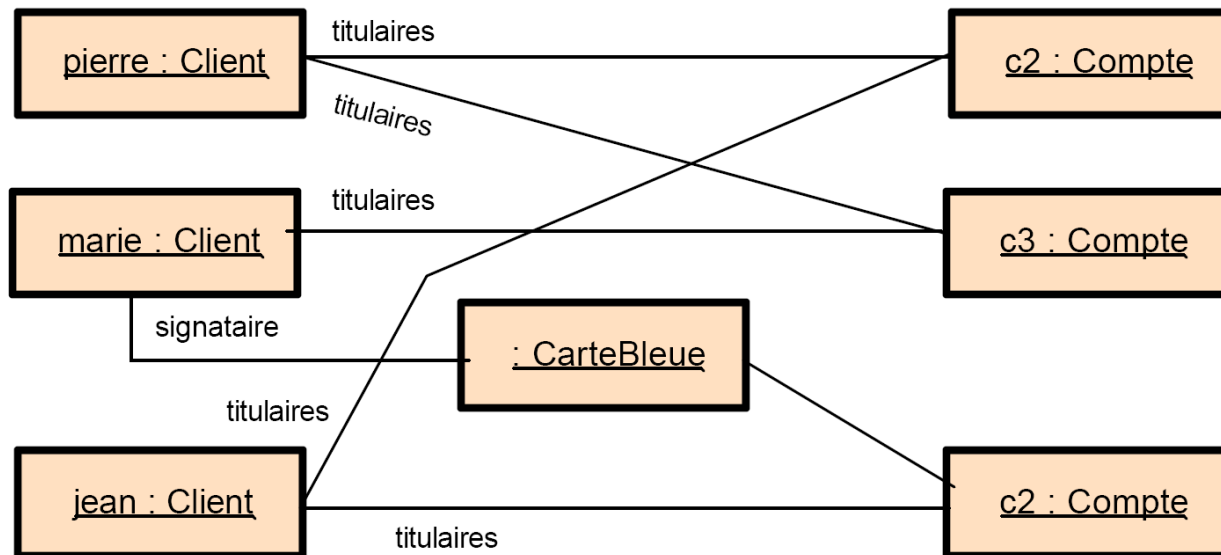
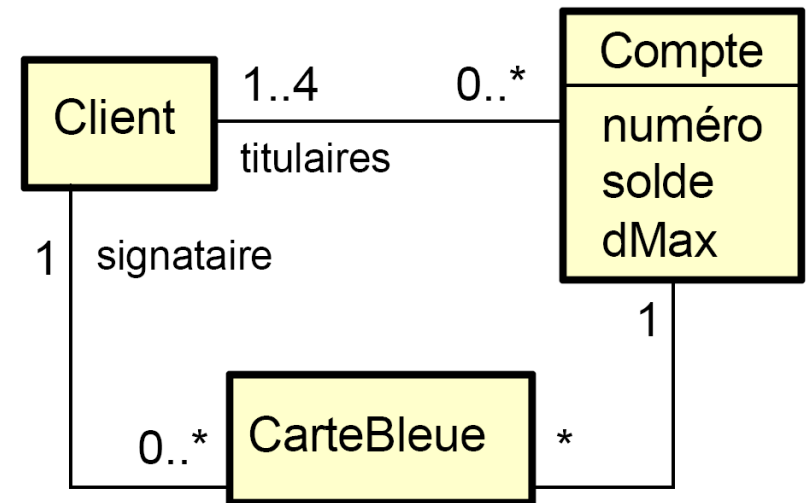
Expression de contraintes avec UML

- Les différents diagrammes d 'UML permettent d'exprimer certaines contraintes
 - Graphiquement
 - Contraintes structurelles (un attribut dans une classe)
 - Contraintes de types (sous-typage)
 - Contraintes diverses (omposition, cardinalité, etc.)
 - Via des propriétés prédéfinies
 - Sur des classes (`{abstract}`)
 - Ssur des rôles (`{ordered}`)
- C'est toutefois insuffisant

Insuffisances d'UML pour représenter certaines contraintes

- Certaines contraintes « évidentes » sont difficilement exprimables avec UML seul.

« Le signataire d'une carte bleue est titulaire du compte correspondant »

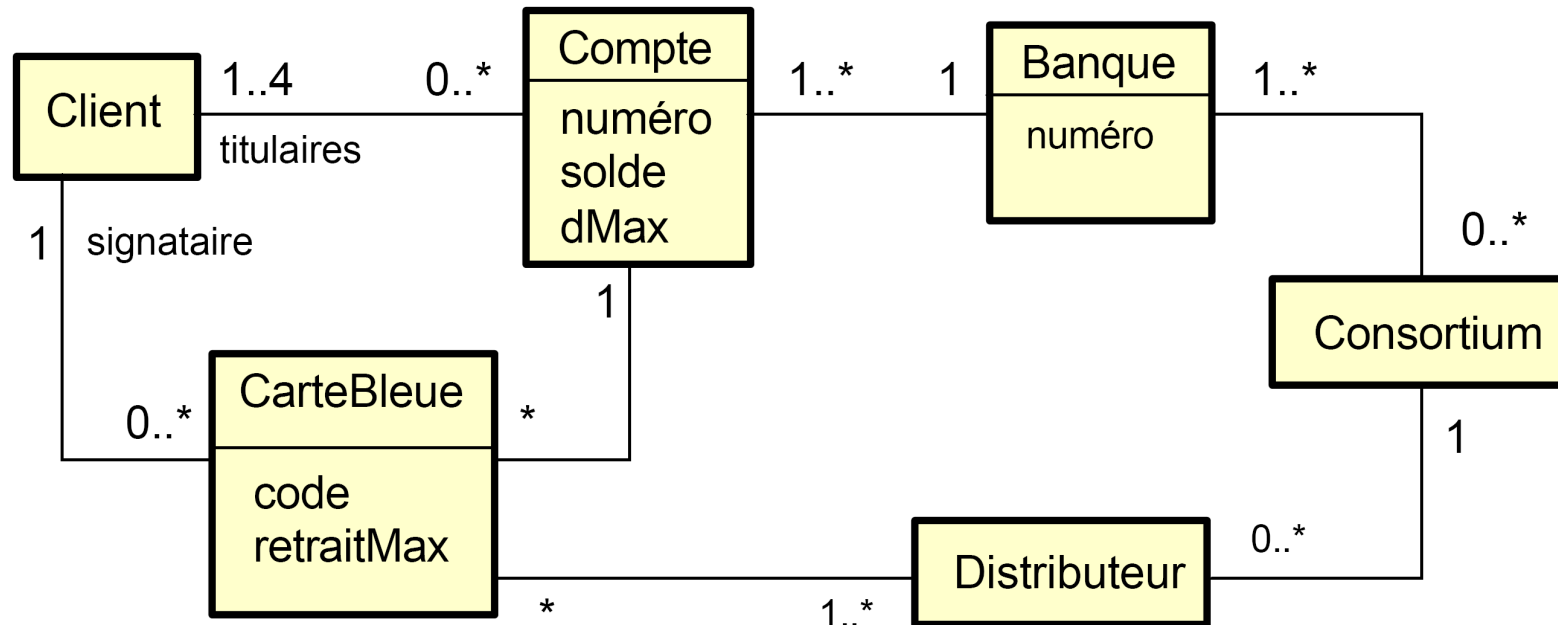


Expression des contraintes en langage naturel

- *Simple* à mettre en oeuvre :
 - Utilisation des notes en UML + texte libre,
 - Compréhensible par tous.
- *Indispensable* :
 - Documenter les contraintes est essentiel,
 - Détecter les problèmes le plus tôt possible.
- Problèmes
 - *Ambigu*, imprécis,
 - Difficile d'exprimer clairement des contraintes complexes,
 - Difficile de lier le texte aux éléments du modèle.

Exemples de contraintes exprimables en OCL

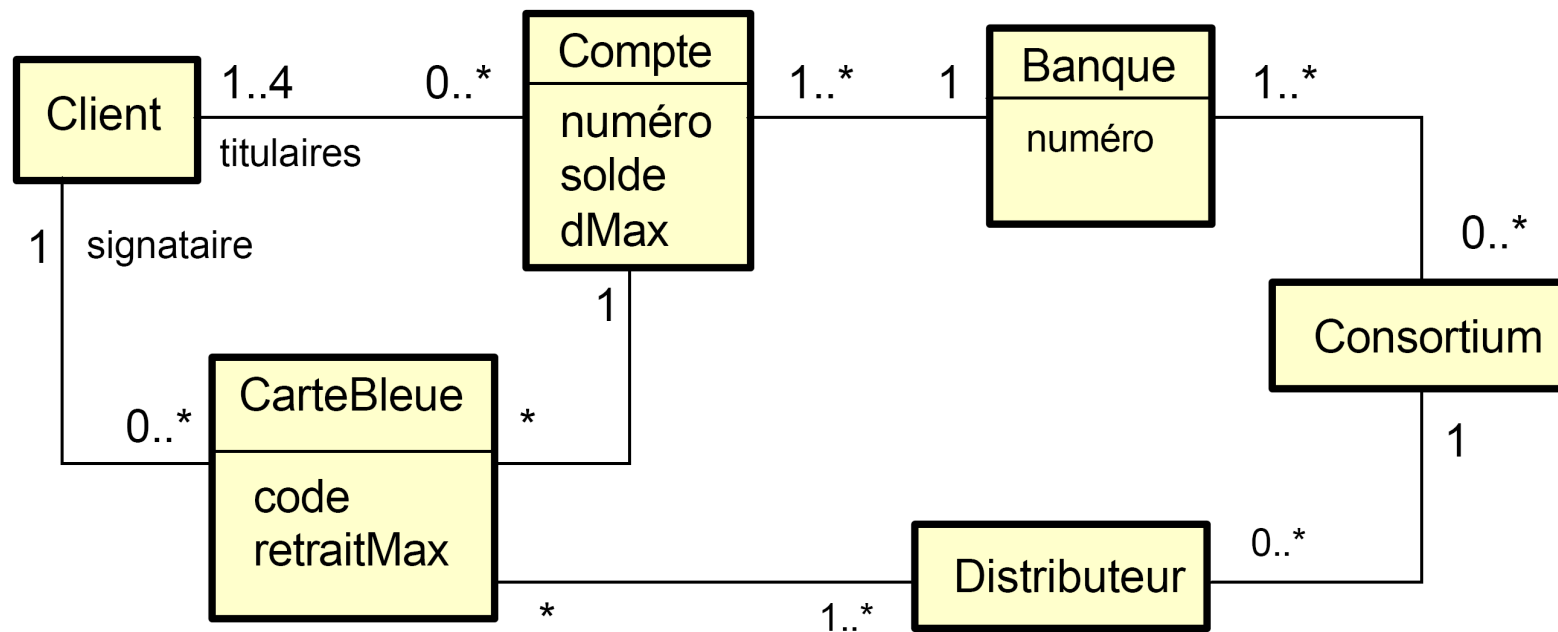
- (1) Le solde d'un compte ne doit pas être inférieur au découvert maximum.
- (2) Le signataire d'une carte bleue associée à un compte en est le titulaire.
- (3) Une carte bleue est acceptée dans tous les distributeurs des consortiums de la banque.
- (4) Les clients d'une banque non affiliée à un consortium ne peuvent pas avoir de carte bleues.



« Le solde d'un compte ne doit pas être inférieur au découvert maximum autorisé »

```
context c : Compte  
inv: solde > dMax
```

```
context Compte  
inv: dMax >= 0  
inv: solde > -dMax
```

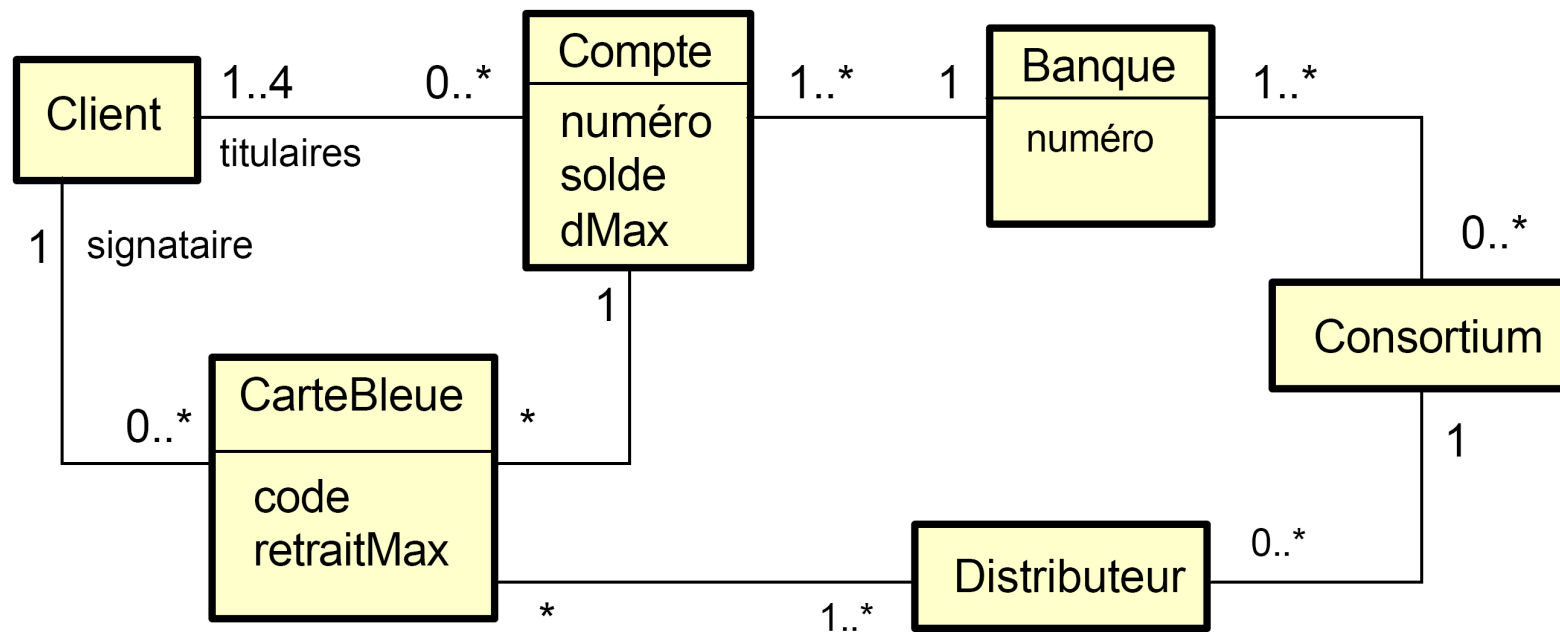


« Le signataire d'une carte bleue associée à un compte est titulaire de ce compte »

- ... est *le* titulaire ...

context CarteBleue

inv: self.signataire= self.Compte.titulaires

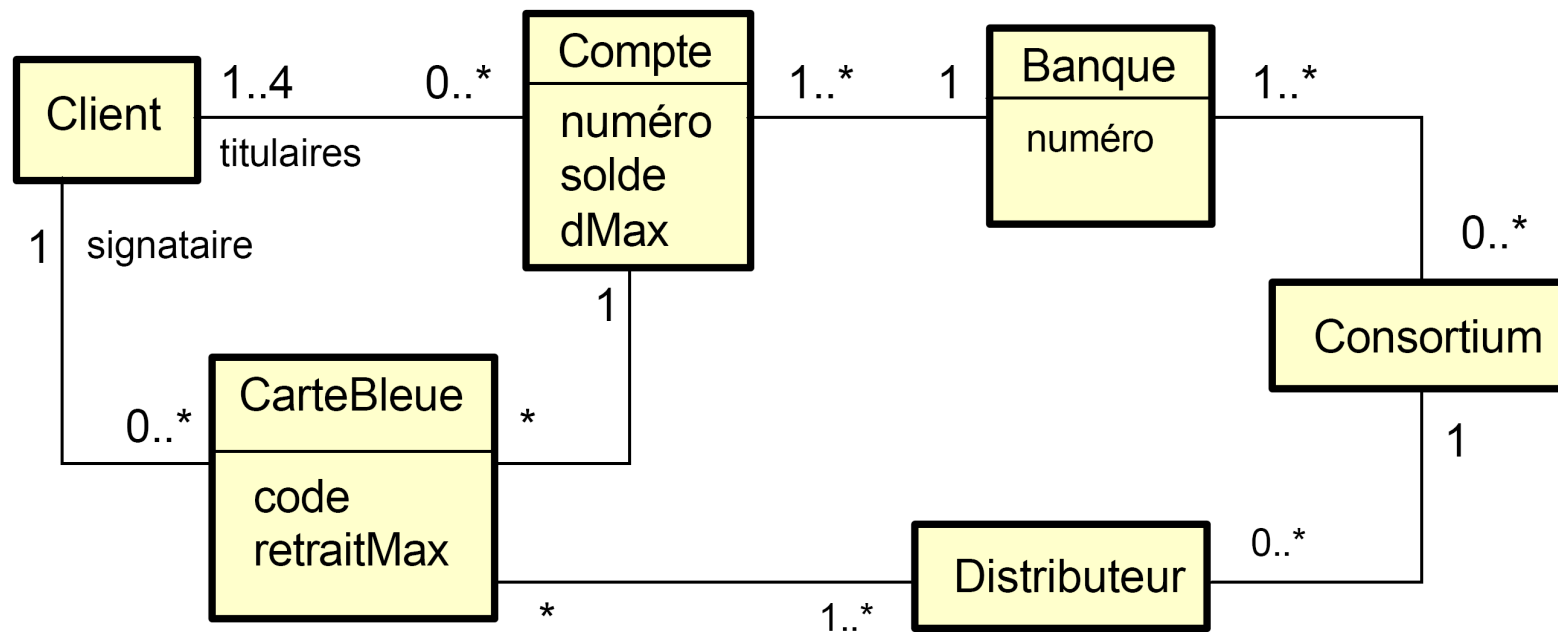


« Le signataire d'une carte bleue associée à un compte est titulaire de ce compte »

- ... est *un des* titulaires ...

context CarteBleue

```
inv: self.Compte.titulaires ->  
      includes(self.signataire)
```



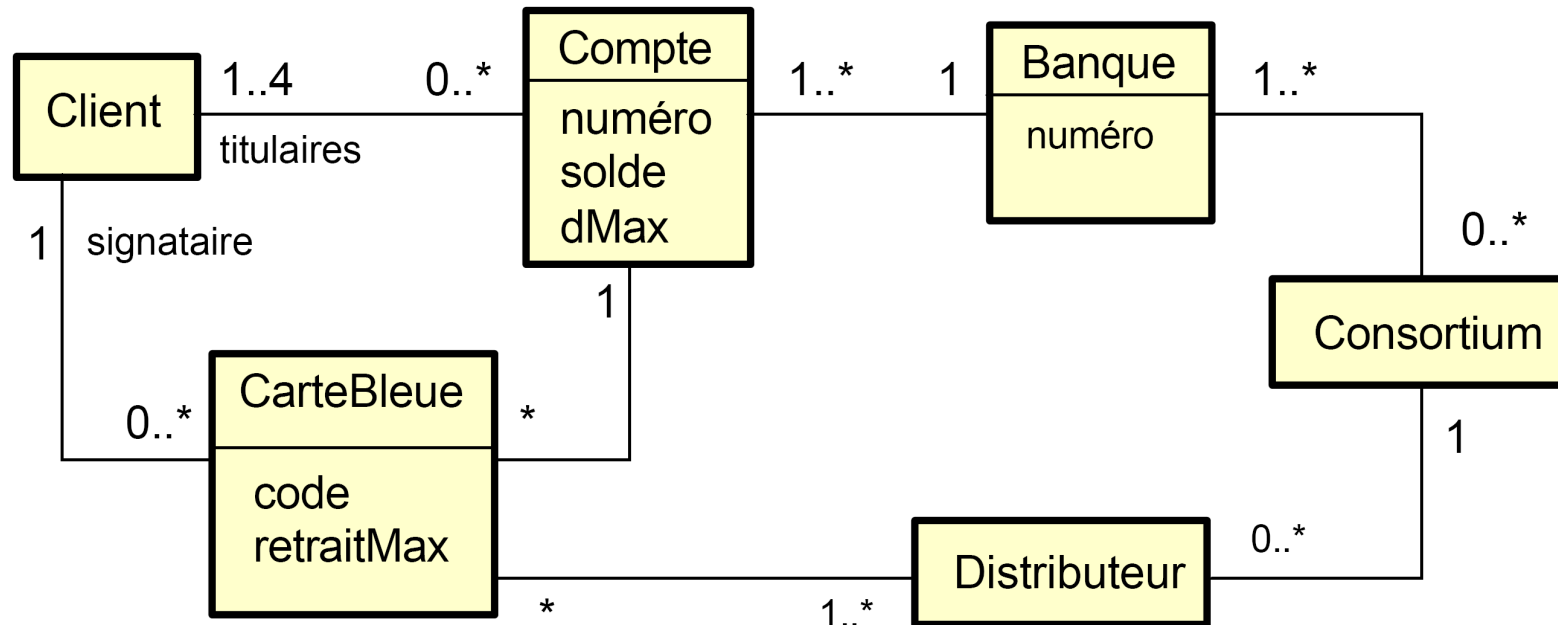
« Une carte bleue est acceptée dans tous les distributeurs des consortiums de la banque »

context CarteBleue

inv:

Distributeur

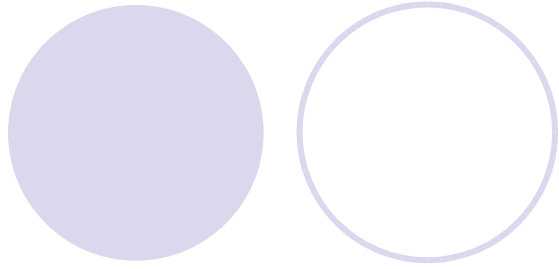
= Compte.Banque.Consortium.Distributeur->asSet





OCL pour l'écriture de contraintes

- OCL : *Object Constraint Language*
 - Langage de requêtes et de contraintes
 - Relativement simple à écrire et à comprendre
 - syntaxe purement textuelle sans symboles étranges
- Parfaitement intégré à UML
 - Sémantique d'UML écrite en OCL
- OCL est en pleine expansion
 - Nouvelle version avec UML2.0
 - Essentiel pour avoir des modèles suffisamment précis
 - De plus en plus d'outils
 - édition, vérification, génération (partielle) de code...



Caractéristiques d'OCL

- Langage d'expressions (*fonctionnel*)
 - Valeurs, expressions, types
 - Fortement typé
 - Pas d'effets de bords
- Langage de *spécification*, pas de programmation
 - Haut niveau d'abstraction
 - Pas forcément exécutable (seul un sous-ensemble l'est)
 - Permet de trouver des erreurs beaucoup plus tôt dans le cycle de vie



Avantages et inconvénients d'OCL

- Points *faibles*
 - Pas aussi rigoureux que VDM, Z ou B
 - Pas de preuves formelles possibles
 - Puissance d'expression limitée
- Points *forts*
 - Relativement simple à écrire et à comprendre
 - Très bien intégré à UML
 - Bon compromis entre simplicité et puissance d'expression
 - Passage vers différentes plateformes technologiques

Contexte d'une contrainte

- Une contrainte toujours associée à un élément de modèle : le *contexte* de la contrainte.
- Deux techniques pour spécifier le contexte :
 - En écrivant la contrainte entre accolades {...} dans une *note*.
L'élément pointé par la note est alors le contexte de la contrainte
 - En utilisant le mot clé « context » dans un document quelconque.

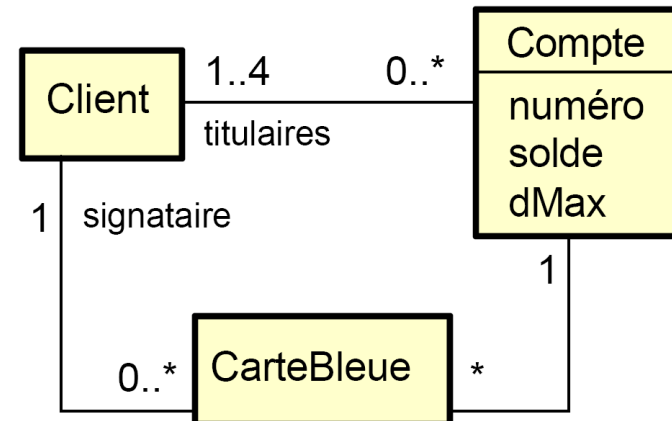
```
context CarteBleue
```

```
inv: Compte.titulaires
```

```
->includes(signataire)
```

```
inv: code>0 and code<=9999
```

```
inv: retraitMax>10
```





Description de prédicats avec OCL

- OCL peut être utilisé pour décrire trois types de prédicats avec les mots clé :
 - *inv*: invariants de classes
`inv: solde < decouvertMax`
 - *pre*: pré-conditions d'opérations
`pre: montantARetirer > 0`
 - *post*: post-conditions d'opérations
`post: solde > solde@pre`

Description d'expressions avec OCL

- OCL peut également être utilisé pour décrire des expressions avec les mots clés :
 - *def*: déclarer des attributs ou des opérations
`def: nbEnfants:Integer`
 - *init*: spécifier la valeur initiale des attributs
`init: enfants->size()`
 - *body*: exprimer le corps de méthodes {query}
`body: enfants->select (age< a)`
 - *derive*: définir des éléments dérivés (/)
`derive: age<18`



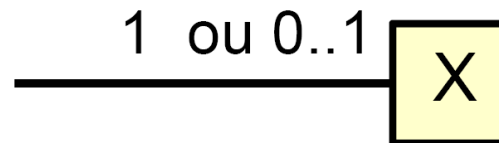
Accès à un attribut ou à une méthode

- Accès à un attribut
 - `objet.attribut`
 - Ex : `self.dateDeNaissance`
- Accès à une méthode
 - `objet.méthode(expr1, expr2, ...)`
 - Ex : `self.impôts(1998)`

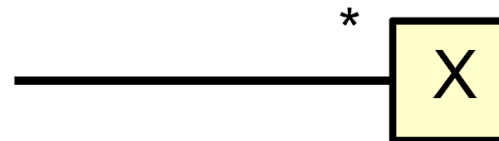
Navigation via une association

- Accéder à l'ensemble des objets liés à un objet donné
 - `objet.nomderole`
 - Le type du résultat dépend de la cardinalité et de `{ordered}`

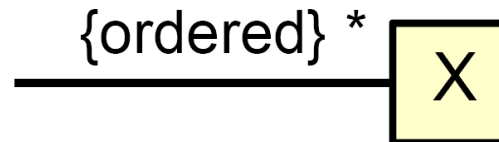
- X



- Set(X)

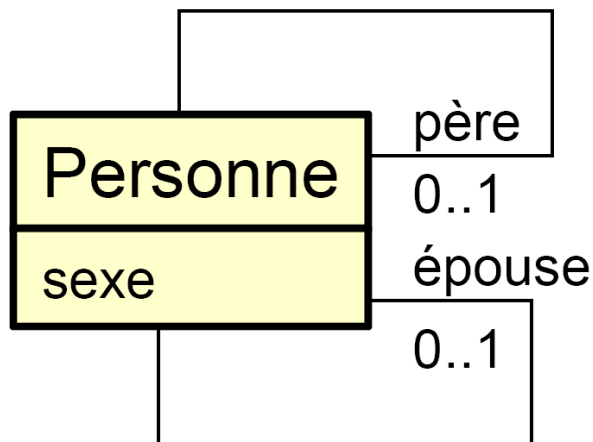


- OrderedSet(X)



Notes sur la navigation via les associations

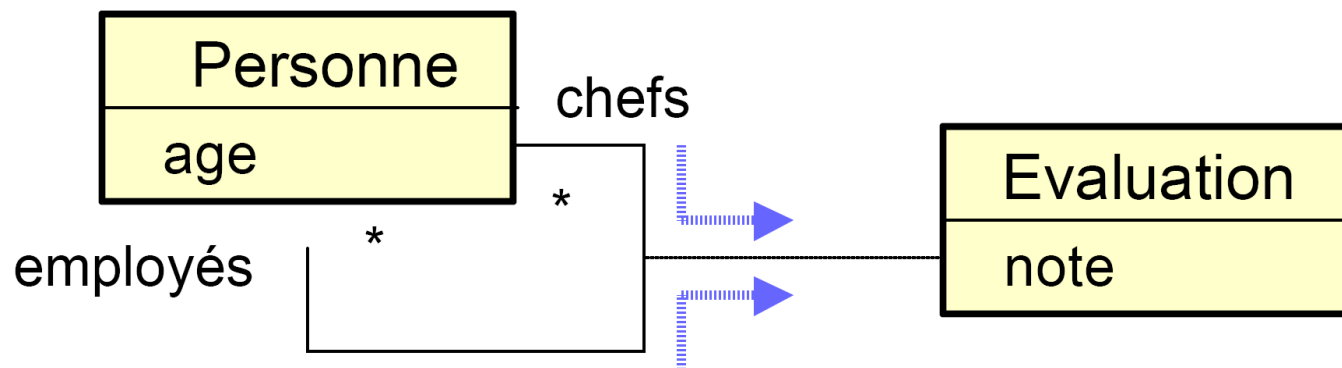
- Un élément est converti en singleton lorsqu'une opération sur une collection est appliquée
 - `Self.père->size() = 1`
- La navigation permet de tester si la valeur est définie (l'ensemble vide représente la valeur indéfinie)
 - `self.père->isEmpty()`
 - `self.épouse->notEmpty() implies self.épouse.sexe = Sexe::féminin`



- Si une association n'a pas de nom de rôle alors on peut utiliser le nom de la classe destination

Navigation vers une association réflexive

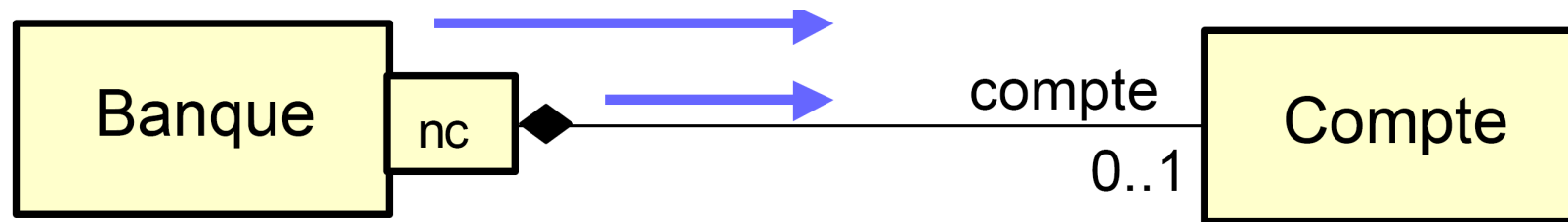
- Si l'association est réflexive, pour éviter les ambiguïtés, il faut indiquer avec un rôle entre crochets [...] comment est parcourue l'association
 - `objet.nomAssociation[nomDeRole]`



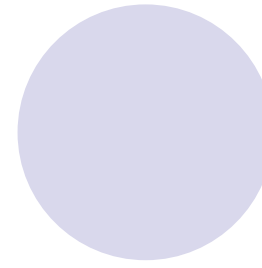
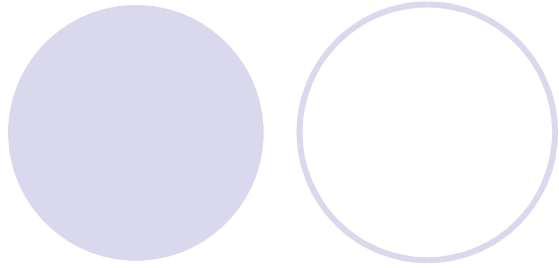
- `p.Evaluation[chefs]`
- `p.Evaluation[employés]`
- `p.Evaluation[chefs].note -> sum()`

Navigation via une association qualifiée

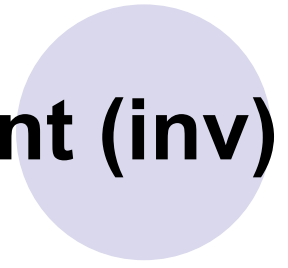
- Accès qualifié
 - `lien.nomderole[valeur, valeur, ...]`
- ou ensemble d'objets liés
 - `lien.nomderole`



- `b.compte[4029]`
- `b.compte`
- `compte`



Invariant (inv)



- *Prédicat* associé à une classe ou une association
 - Doit être vérifié à tout instant
- Le contexte est défini par un objet
 - Cet objet peut être référencé par self
- L'invariant peut être nommé

```
context Personne
```

```
inv pasTropVieux : age < 110
```

```
inv : self.age >= 0
```




Exemples d'invariants

```
context Personne
```

```
  inv : age>0 and self.age<110
```

```
  inv mariageLégal : marié implies age > 16
```

```
  inv enfantsOk : enfants->size() < 20
```

```
  inv : not enfants->includes(self)
```

```
  inv : enfants->includesAll( filles)
```

```
  inv : enfants->forall(e | self.age-e.age < 14)
```



Pré-condition et post-condition

- *Prédicats* associés à une opération
 - Les pré-conditions doivent être *vérifiées avant l'exécution*
 - Les post-conditions sont *vraies après l'exécution*
 - `self` désigne l'objet sur lequel l'opération a lieu
 - Dans une post-condition :
 - `@pre` permet de faire référence à la valeur avant l'opération
 - `result` désigne le résultat
 - `ocsIsNew()` indique si un objet n'existait pas dans l'état précédent
- ```
context Type::opération(param1:Type1,...):Type
pre nom1 : param1 < ...
post : ... result > ...
```

# Exemples de pré et post-conditions

```
context Personne::retirer(montant:Integer)
```

```
 pre : montant > 0
```

```
 post : solde < solde@pre - montant
```

```
context Personne::salaire() : integer
```

```
 post : result >= Legislation::salaireMinimum
```

```
context Compagnie::embaucheEmployé(p:Personne)
 : Contrat
```

```
 pre pasPrésent : not employés->includes(p)
```

```
 post embauché : employés = employés@pre ->including
 (p)
```

```
 post : result.oclIsNew()
```

```
 post : result.compagnie = self and result.employé =
 p
```



## Définition additionnelle (def)

- Il est possible en OCL de définir dans une classe existante:
  - de *nouveaux attributs*
  - de *nouvelles opérations*

```
context Classe
```

```
 def: nomatt : type = expr
```

```
 def: nomop(...) : type = expr
```

- Utile pour décomposer des requetes ou contraintes

```
context Personne
```

```
def: ancestres() :
```

```
 Set(Personne) = parents
```

```
 ->union(parents.ancestres())
```

```
 ->asSet())
```

```
inv: not ancestres()->includes(self)
```

# Expression de propriétés dérivées (derive)

- Préciser en OCL la valeur d'un attribut ou d'une association dérivée
  - Complète la notation « / »

```
context Personne::estMarié : Boolean
derive : conjoint->notEmpty
```

```
context Personne::filles : Set(Personne)
derive : enfants->select(sexe = Sexe::Feminin)
```

```
context Personne::grandParents : Set(Personne)
derive: parents.parents->asSet()
```

# Expression du corps d'une méthode (body)

- Description en OCL d'une méthode sans effet de bord ({isQuery})
- Equivalent à une requête

```
context Personne:acf(p:Personne):OrderedSet(Personne)
body : self.ancestres()
 ->intersection(p.ancestres())
 ->select(sexe=Sexe::Feminin)
 ->sortedBy(dateDeNaissance)

context Personne
def: ancestres : Set(Personne) = parents
 ->union(parents.ancestres->asSet())
```



# Syntaxe des expressions

- OCL est un langage simple d'*expressions*
  - constantes
  - identificateur
  - self
  - `expr op expr`
  - `exprojet.propobjet`
  - `exprojet.propobjet ( parametres )`
  - `exprcollection -> propcollection(parametres)`
  - `package::package::element`
  - `if cond then expr else expr endif`
  - `let var : type = expr in expr`

# Accès aux propriétés et aux éléments

- « . » permet d'accéder à une *propriété d'un objet*
- « -> » permet d'accéder à une *propriété d'une collection*
- « :: » permet d'accéder à un *élément* d'un paquetage, d'une classe, ...
- Des règles permettent de mixer collections et objets

```
self.impôts(1998) / self.enfants->size()
self.salaire() - 100
self.enfants->select(sexe = Sexe::masculin)
self.enfants->isEmpty()
self.enfants->forall(age>20)
self.enfants->collect(salaire)->sum()
self.enfants.salaire->sum()
self.enfants->union(self.parents)->collect(age)
```





# Types entiers et réels

- *Integer*
  - Valeurs : 1, -5, 34, 24343, ...
  - Opérations : +, -, \*, div, mod, abs, max, min
- *Real*
  - Valeurs : 1.5, 1.34, ...
  - Opérations : +, -, \*, /, floor, round, max, min
- Le type Integer est « conforme » au type Real



# Type booléen

- *Boolean*
  - Valeurs : true, false
  - Opérations : not, and, or, xor, implies, if-then-else-endif
- L'évaluation des opérateurs or, and, if est partielle
  - « true or x » est toujours vrai, même si x est indéfini
  - « false and x » est toujours faux, même si x est indéfini

```
(age<40 implies salaire>1000)
 and (age>=40 implies salaire>2000)
if age<40 then salaire > 1000
 else salaire > 2000 endif
salaire > (if age<40 then 1000 else 2000 endif)
```



# Chaînes de caractères

- String
  - Valeurs : ' ', 'une phrase'
  - Opérations :
    - « = »
    - s.size()
    - s1.concat(s2), s1.substring(i1,i2)
    - s.toUpperCase(), s.toLowerCase(),

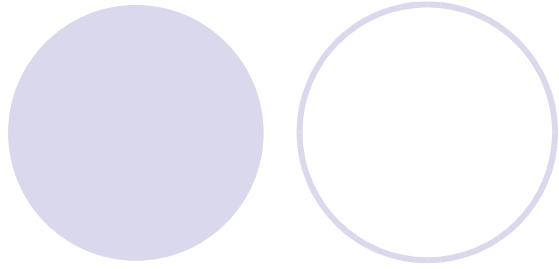
```
nom=nom.substring(1,1).toUpperCase()
 .concat(nom.substring(2,nom.size()).toLowerCase())
```
- Les chaînes ne sont pas des séquences de caractères
  - String  $\nleftrightarrow$  Sequence(character), le type character n'existe pas

# Utilisation des valeurs de types énumérés

- Jour::Mardi
  - noté « #Mardi » avant UML2.0
- Opérations
  - =, <>
  - Pas de relation d'ordre

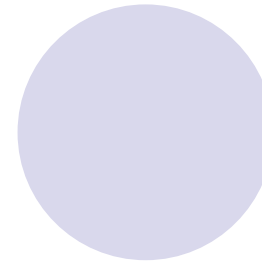
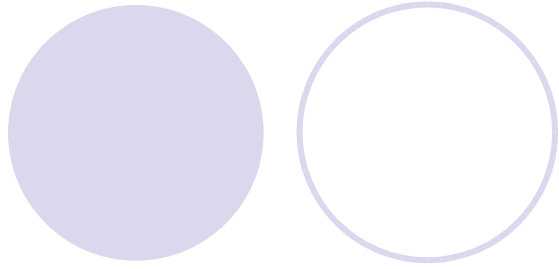


```
épouse->notEmpty()
 implies épouse.sexe = Sexe::Feminin
```



# Éléments et singletons

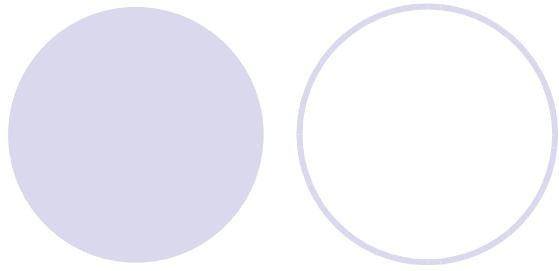
- Dans tout langage typé il faut distinguer
  - un élément  $e$
  - du singleton contenant cet élément  $\text{Set}\{e\}$
- Pour simplifier la navigation OCL, une conversion implicite est faite lorsqu'une opération sur une collection est appliquée à un élément isolé
  - $\text{elem} \rightarrow \text{prop}$  est équivalent à  $\text{Set}\{\text{elem}\} \rightarrow \text{prop}$
  - $\text{self} \rightarrow \text{size}() = 1$



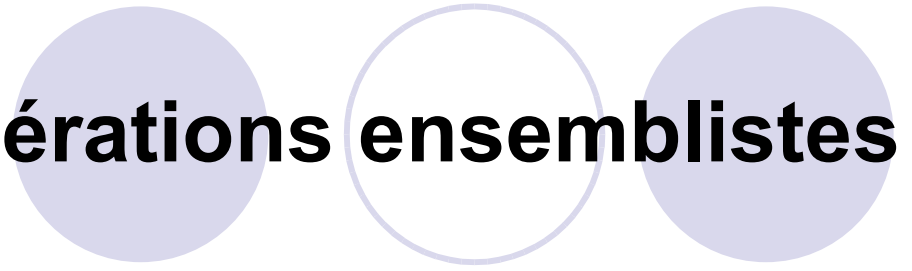
- Ensembles : `Set (T)`
  - Éléments unique non ordonnés
- Ensembles ordonnés : `OrderedSet (T)`
  - Éléments uniques et ordonnés
- Sac : `Bag (T)`
  - Éléments répétables sans ordre
- Sequence : `Sequence (T)`
  - Éléments répétables mais ordonnés
- Collection est le type de base `Collection (T)`

# Opérations de base sur les collections

- Cardinalité : `coll -> size()`
- Vide : `coll -> isEmpty()`
- Non vide : `coll -> nonEmpty()`
- Nombre d'occurrences : `coll -> count(elem)`
- Appartenance : `coll -> includes( elem )`
- Non appartenance : `coll -> excludes( elem )`
- Inclusion : `coll -> includesAll(coll)`
- Exclusion : `coll -> excludesAll(coll)`
- Somme des éléments : `coll -> sum()`

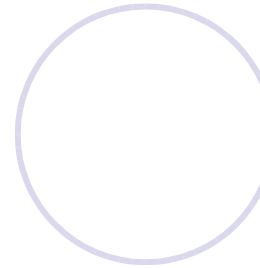
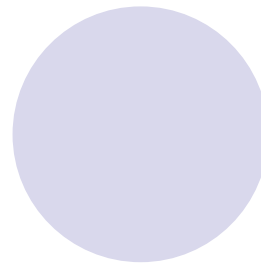
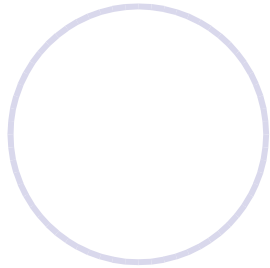
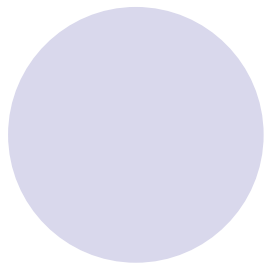


# Opérations ensemblistes



- Union : `ens->union(ens)`
- Intersection : `ens->intersection(ens)`
- Difference : `ens1-ens2`
- Ajout d 'un élément : `ens->including(elem)`
- Suppression d 'un élément : `ens->excluding(elem)`
- Conversion vers une liste : `ens->asSequence()`
- Conversion vers un sac : `ens->asBag()`
- Conv.vers un ens. ord. : `ens->asOrderedSet()`





## Filtrage

- *Select* retient les éléments vérifiant la condition
  - `coll -> select( cond )`
- *Reject* élimine ces éléments
  - `coll -> reject( cond )`
- *Any* sélectionne l'un des éléments vérifiant la condition
  - `coll -> any( cond )`
  - opération non déterministe
  - utile lors de collection ne contenant qu'un élément
  - retourne la valeur indéfinie si l'ensemble est vide

```
self.enfants->select(age>10 and sexe=Sexe::Masculin)
```



# Autres syntaxes pour le filtrage

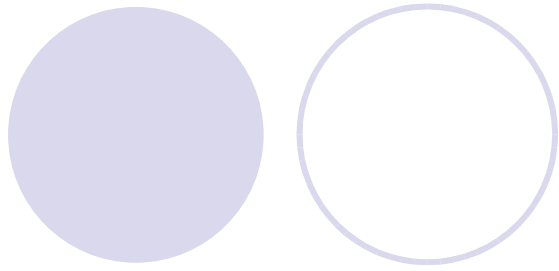
- Il est également possible de
  - nommer la variable
  - d'expliquer son type

```
self.employé->select (age > 50)
```

```
self.employé->select (p | p.age>50)
```

```
self.employé->select (p:Personne | p.age>50)
```

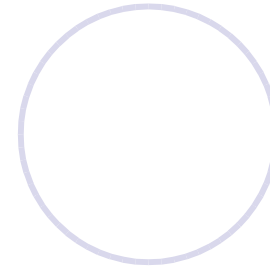
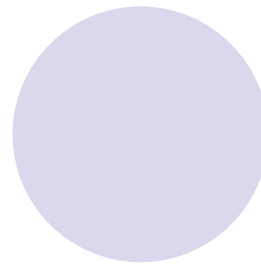
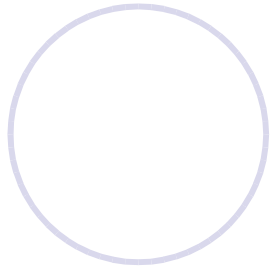
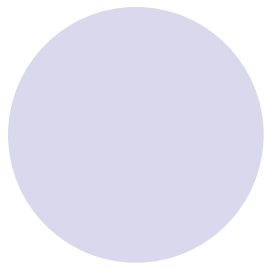
```
self.employé->reject (p:Personne | p.age<=50)
```



# Quantificateurs

- *ForAll* est le quantificateur *universel*
  - `coll -> forAll( cond )`
- *Exists* est le quantificateur *existantiel*
  - `coll -> exists( cond )`
  - `self.enfants->forall( age<10 )`
  - `self.enfants->exists( sexe=Sexe::Masculin )`
- Il est possible
  - de nommer la variable
  - d'expliquer son type
  - de parcourir plusieurs variables à la fois

```
self.enfants
 ->exists(e1, e2 | e1.age=e2.age and e1<>e2)
```



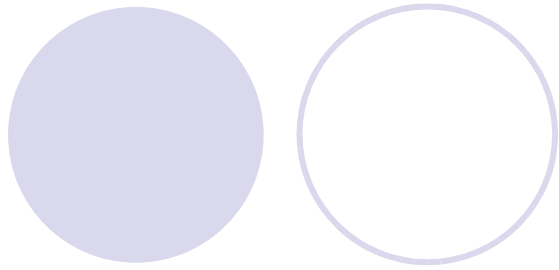
**Unicité**

- Retourne vrai si pour chaque valeur de la collection, l'expression retourne une valeur différente
  - `coll -> isUnique ( expr )`
  - Equivalence entre
    - `self.enfants -> isUnique ( prénom )`
    - `self.enfants->forall( e1,e2 : Personne | e1 <> e2 implies e1.prénom <> e2.prénom)`
- Utile pour définir la notion de clé

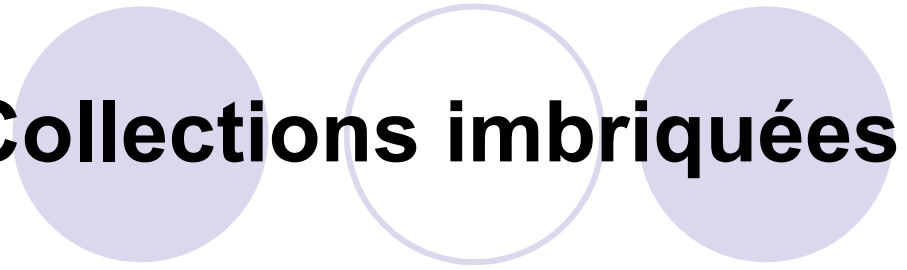


# Tuples

- Champs nommés, pas d'ordre entre les champs
  - Tuples (valeurs)
    - `Tuple{ x=-1.5 , y=12.6 }`
    - `Tuple{ y=12.6 , x=-1.5 }`
    - `Tuple{ y:Real=12.6, x:Real=-1.5 }`
    - `Tuple{ nom = 'dupont', prénom='léon', age=43 }`
  - A partir d'UML 2.0
- Définition de types tuples
  - `TupleType(x:Real, y:Real)`
  - `TupleType(y:Real, x:Real)`
  - `TupleType(nom:String, prénom:String, age:Integer)`



# Collections imbriquées



- Collections imbriquées
  - Jusqu'à UML 2.0 pas de collections de collections car peu utilisé et plus complexe à comprendre
  - Mise à plat intuitive lors de la navigation
    - self.parents.parents
  - Mise à plat par default lié à l'opération implicite collect
- A partir de UML 2.0 imbrications arbitraires des constructeurs de types
  - `Set ( Set (Personne) )`
  - `Sequence ( OrderedSet (Jour) )`
- Très utile pour spécifier des structures non triviales

# Conservation de l'imbrication pour la navigation

- L'opération *collect* met à plat les collections

- utile pour naviger

```
enfants.enfants.prénom
```

```
= enfants->collect(enfants)->collect(prénom)
```

```
= Bag{ 'pierre', 'paul', 'marie', 'paul' }
```

- *CollectNested* permet de conserver l'imbrication

```
enfants->collectNested(enfants.prénom)
```

```
= Bag{Bag{ 'pierre', 'paul' },
```

```
 Bag{ 'marie', 'paul' } }
```

The header features a series of circles. On the left, there is a solid light purple circle followed by an outlined light purple circle. On the right, there is a solid light purple circle, an outlined light purple circle, and another solid light purple circle. The text 'Itérateur général' is positioned over the rightmost outlined circle.

# Itérateur général

- L'itérateur le plus général est *iterate*
  - Les autres itérateurs (forall, exist, select, etc.) en sont des cas particulier

```
coll -> iterate(
 élém : Type ;
 accumulateur:Type2=<valInit>
 | <expr>)
```

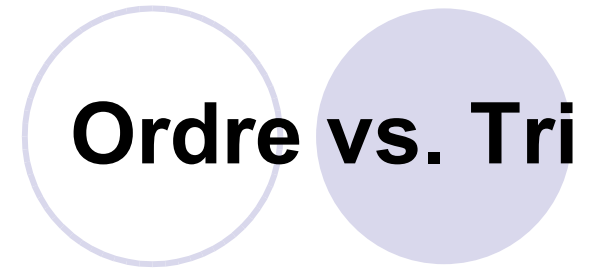
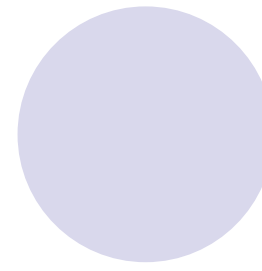
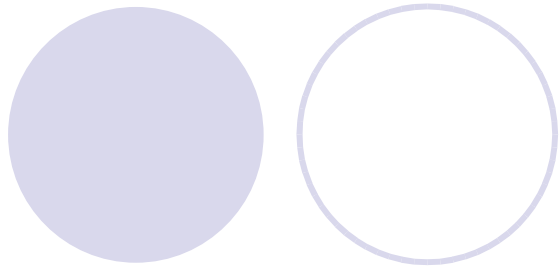
- Exemple

- enfants->iterate(e:Enfant;ac:Integer=0|acc+e.age)  
équivalent en pseudo code à

```
○ ac : Integer = 0 ;
 e:Enfant in enfants
 acc+e.age
```

```
foreach
 acc :=
return acc
```





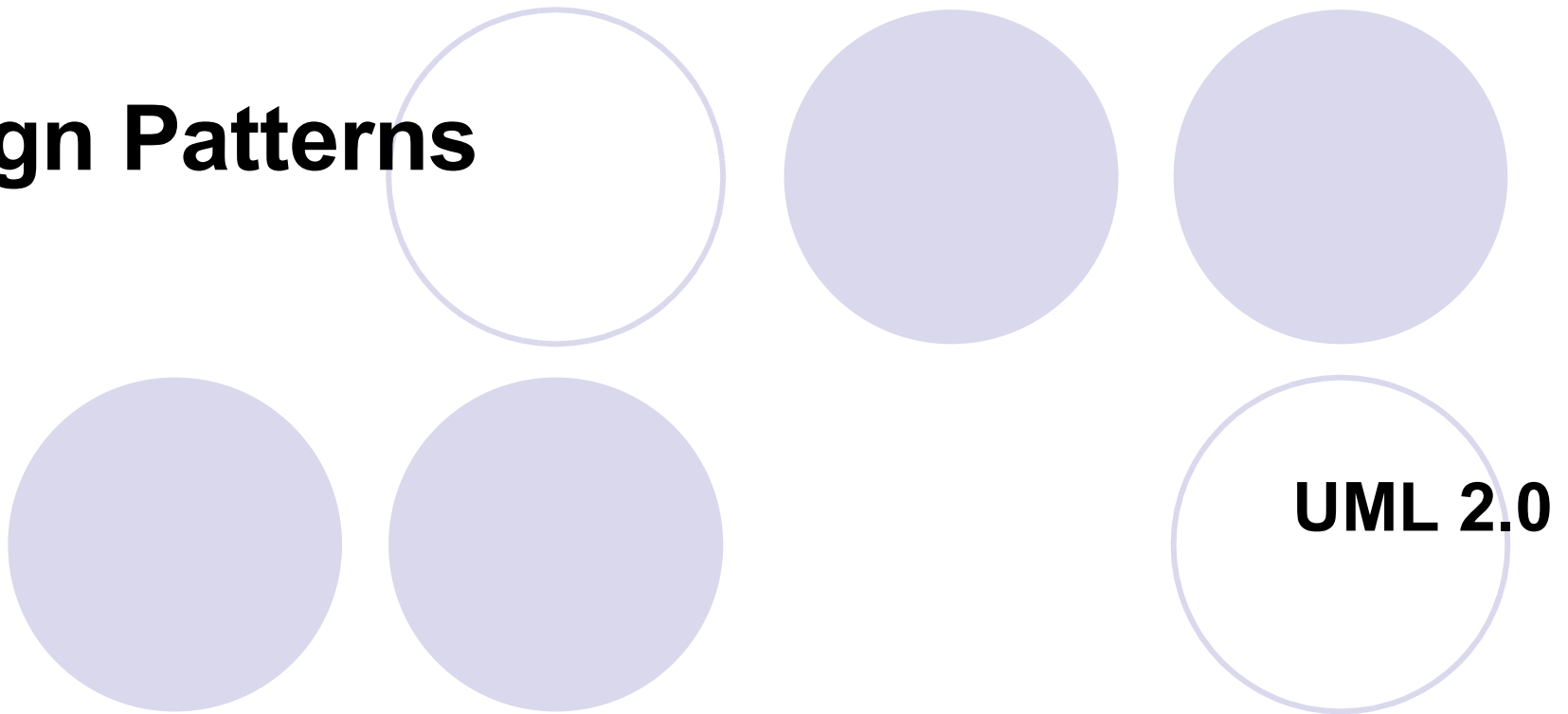
- Collections ordonnées
  - Sequence
  - OrderedSet
- ... mais pas forcément triées
  - Séquence simple (l'ordre compte) :
    - Sequence { 12, 23, 5, 5, 12 }
  - Séquence triée :
    - Sequence { 5, 5, 12, 12, 23 }



# Tri d'une collection

- Pour trier une collection en fonction d'une expression
  - `coll -> sortBy( expr )`
  - L'opération « < » doit être définie sur le type correspond à `expr`  
`enfants->sortBy( age )`  
`let ages = enfants.age->sortBy( a | a ) in`  
`ages.last() - ages.first()`  
`enfants->sortBy( enfants->size() )->last()`
- Le résultat est de type
  - `OrderedSet` si l'opération est appliquée sur un `Set`
  - `Sequence` si l'opération est appliquée sur un `Bag`

# Design Patterns





# Un concept issu de l'architecture

- Christopher Alexander, architecte, définit en 1977 les *patrons de conception* comme
  - La description d'un problème rémanent et de sa solution
  - Une solution pouvant être utilisée des millions de fois sans être deux fois identique
  - Une forme de conception, pattern, modèle, patron de conception
- Ce concept attire les chercheurs en COO dès les années 80
  - « **Design Patterns of Reusable Object-Oriented** »
    - *Le GOF : Erich Gamma, Richard Helm, Ralph Johson et John Vlissides*
    - Addison-Wesley, 1995



# Patron de conception logicielle

- Un *patron de conception* (design pattern) est la description d'une solution classique à un problème récurrent.
  - Il décrit une partie de la solution...
  - avec des relations avec le système et les autres parties.
  - C'est une technique d'architecture logicielle.
- Ce n'est pas
  - Une brique : l'application d'un pattern dépend de l'environnement
  - Une règle : un pattern ne peut pas s'appliquer mécaniquement
  - Une méthode : ne guide pas une prise de décision ; un pattern est la décision prise

# Documentation d'un patron de conception

- Les principaux éléments de description d'un pattern sont :
  - Le *nom* du pattern résume le problème de design.
  - Son *intention* est une courte description des objectifs du pattern, de sa raison d'être.
  - Les *indication* d'utilisation décrivent les cas où le pattern peut être utile.
  - La *motivation* montre un cas particulier dans lequel le patron peut être utilisé.
  - La *structure* est une représentation graphique des classes du modèle.
- Dans l'ouvrage du GOF (et ce cours), utilisation d'OMT.



# Avantages des patrons de conception

- Capitalisation de l'expérience : *réutilisation* de solutions qui ont prouvé leur efficacité
  - Rendre la conception beaucoup plus rapide
- Elaboration de constructions logicielles de meilleure *qualité* grâce à un niveau d'abstraction plus élevé
  - Réduction du nombre d'erreurs, d'autant que les patrons sont examinés avec attention avant d'être publiés
- *Communication* plus aisée
  - Ecrire du code facilement compréhensible par les autres
- Apprentissage en suivant de bons exemples

# Inconvénients des patrons de conception

- Nécessité d'un *effort de synthèse* conséquent
  - Reconnaître, abstraire...
- Nécessité d'un *apprentissage* et d'expérience
- Les patterns « se dissolvent » en étant utilisés
- Les patrons sont *nombreux* (23 dans l'ouvrage du GOF, d'autres sont publiés régulièrement)
  - Lesquels sont semblables ?
  - Les patrons sont parfois de niveaux différents : certains patterns s'appuient sur d'autres...





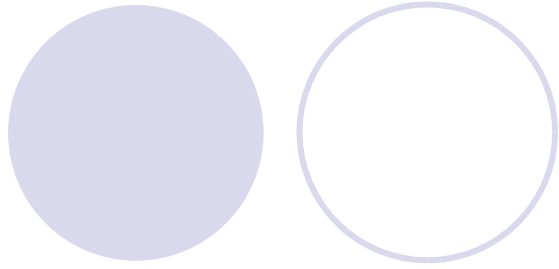
# Catégories de patrons de conception

- Patrons de *création*
  - Ils définissent comment faire l'instanciation et la configuration des classes et des objets.
- Patrons de *structure*
  - Ils définissent l'usage des classes et des objets dans des grandes structures ainsi que la séparation entre l'interface et l'implémentation.
- Patrons de *comportement*
  - Ils définissent la manière de gérer les algorithmes et les divisions de responsabilités.



# Classification des patrons du GOF

- Création
  - *Objets* : Abstract factory, Builder, Prototype, Singleton
  - *Classes* : Factory method
- Structure :
  - *Objets* : Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
  - *Classes* : Adapter
- Comportement :
  - *Objets* : Chain of responsibility, Command, Iterator, Memento, Observer, State, Strategy
  - *Classes* : Interpreter, Template method



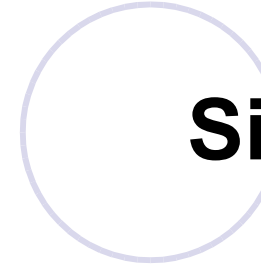
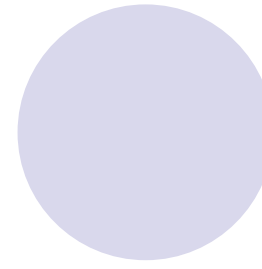
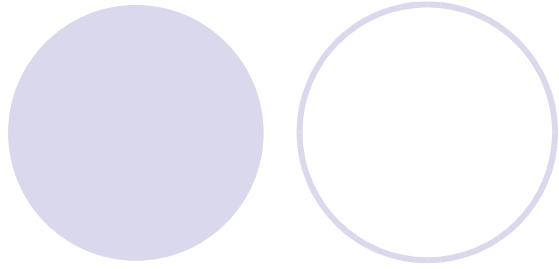
# Creational patterns

- Formes de création pour :
  - Abstraire le processus d'instanciation.
  - Rendre indépendant de la façon dont les objets sont créés, composés, assemblés, représentés.
  - Encapsuler la connaissance de la classe concrète qui instancie.
  - Cacher ce qui est créé, qui crée, comment et quand.



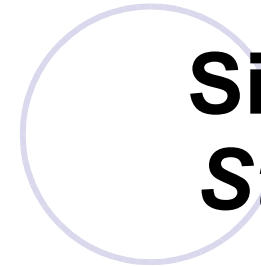
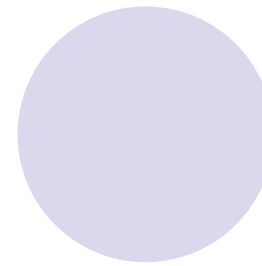
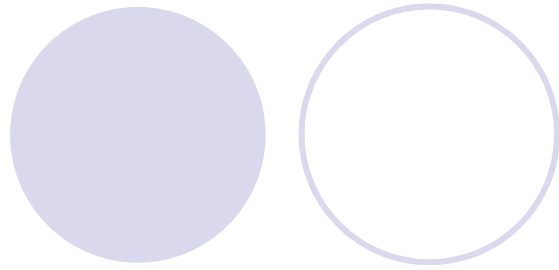
# Exemples de Creational patterns

- *AbstractFactory* pour passer un paramètre à la création qui définit ce que l'on va créer
- *Builder* pour passer en paramètre un objet qui sait construire l'objet à partir d'une description
- *FactoryMethod* pour que la classe sollicitée appelle des méthodes abstraites
- *Prototype* : des prototypes variés existent qui sont copiés et assemblés
- *Singleton* pour garantir qu'il n'y ait qu'une seule instance de la classe

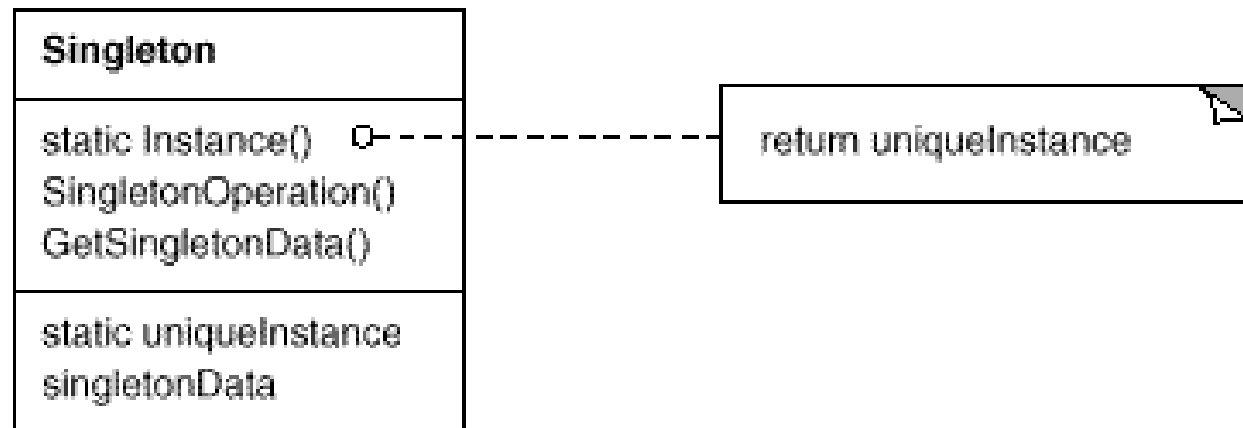


# Singleton

- *Intention* :
  - Garantir qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette classe.
- *Indications* :
  - S'il doit n'y avoir exactement qu'une instance de la classe et qu'elle doit être accessible aux clients de manière globale.
  - Si l'instance doit pouvoir être sous-classée et si l'extension doit être accessible aux clients sans qu'ils n'aient à modifier leur code.
- *Motivation* :
  - Spooler d'impression, gestionnaire d'affichage
  - Générateur de nombres aléatoires avec graine



# Singleton *Structure*

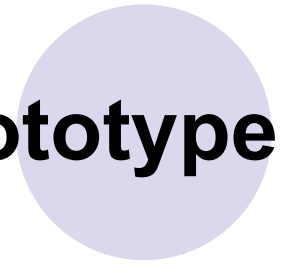
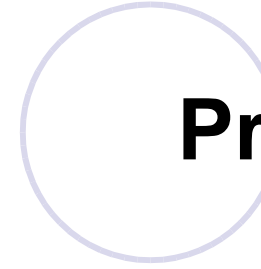
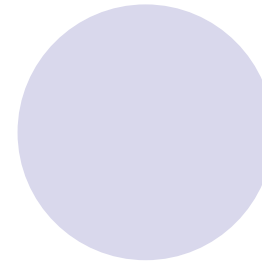
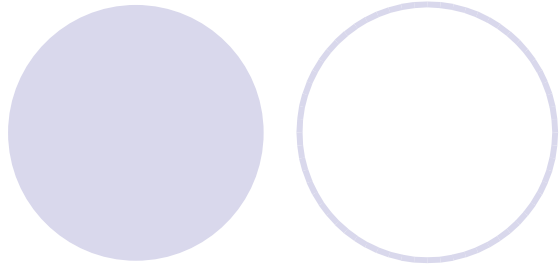


The slide features a decorative header with five circles. From left to right: a solid light purple circle, an outlined light purple circle, a solid light purple circle, an outlined light purple circle, and a solid light purple circle. The text 'Singleton Implémentation' is positioned to the right of the third and fourth circles.

# Singleton *Implémentation*

```
public class Singleton {
 private static Singleton uniqueInstance = null;
 private Singleton() {}
 public static Singleton Instance() {
 if (uniqueInstance == null)
 uniqueInstance = new Singleton();
 return uniqueInstance;
 }
}
```

- A ajouter :
  - Attributs et méthodes spécifiques de la classe singleton en question.
  - Paramètres de constructeur si besoin .

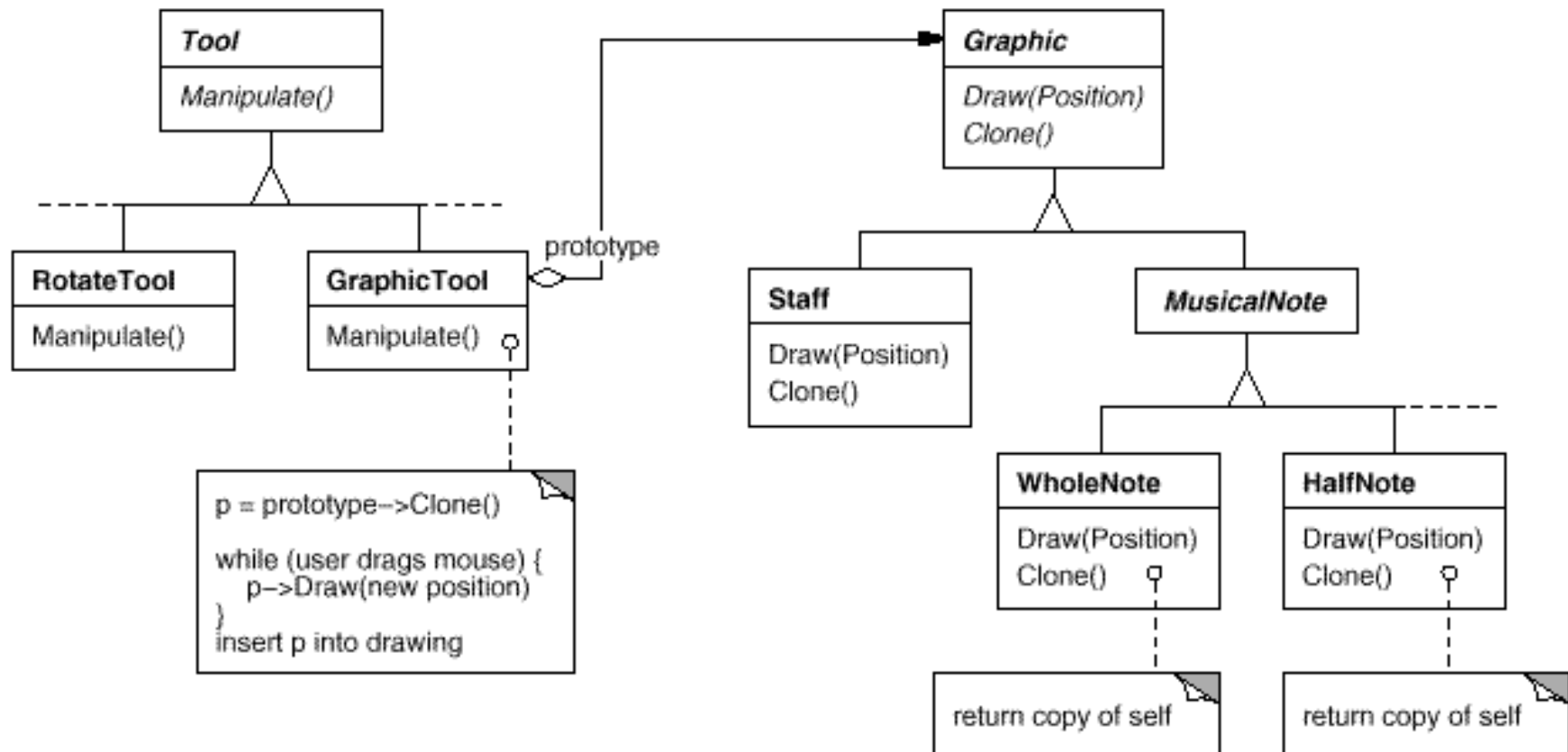


**Prototype**

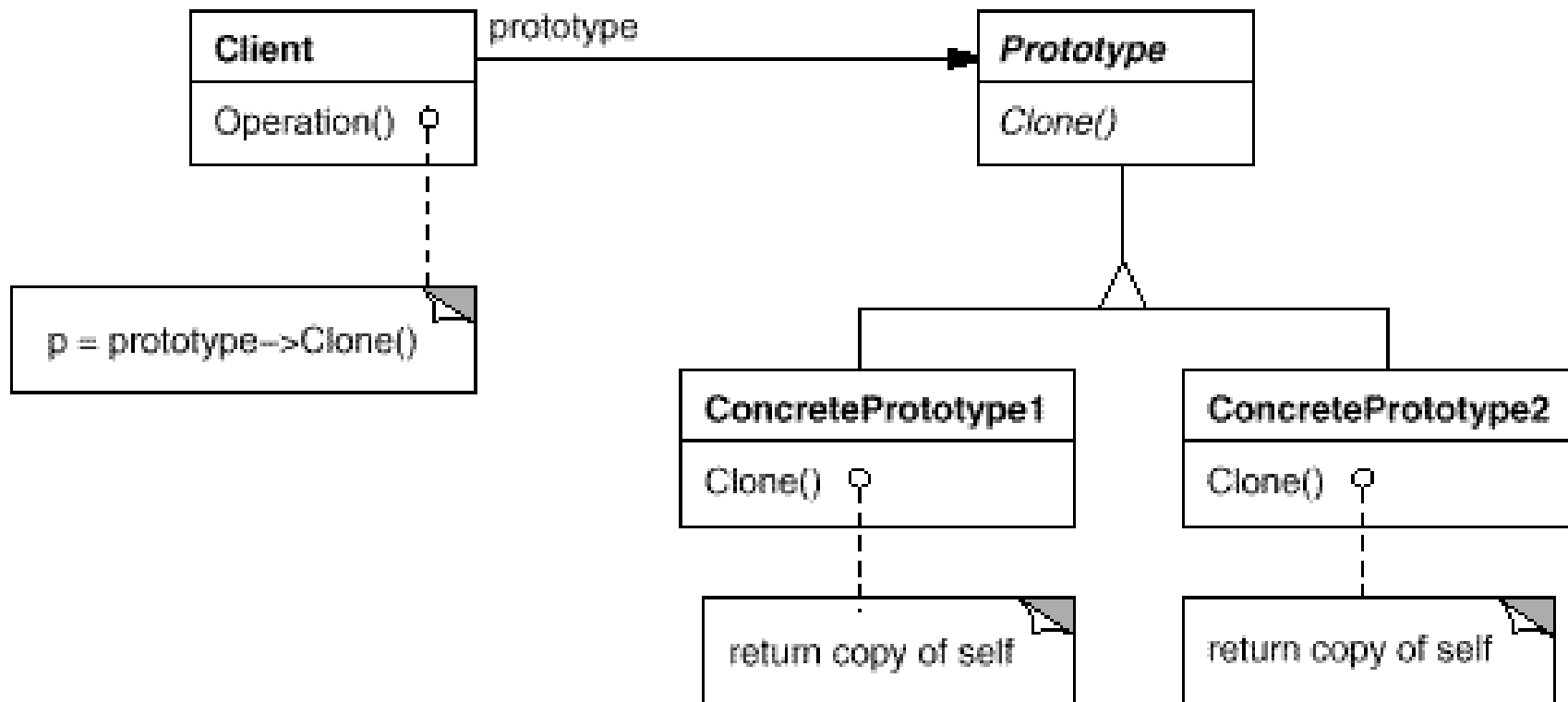
- *Intention* :
  - Spécifie le type d'objets à créer à partir d'une instance de prototype, et crée de nouveaux objets en copiant ce prototype
- *Indications* :
  - Si les classes à instancier sont spécifiées à l'exécution
  - Si l'on ne dispose pas de constructeur de copie adéquat
- *Exemples* :
  - Copier-coller sans connaître le type d'objet.

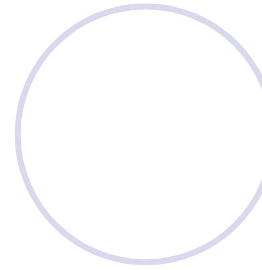
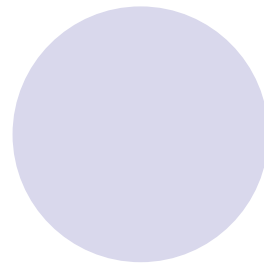
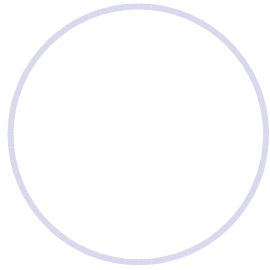
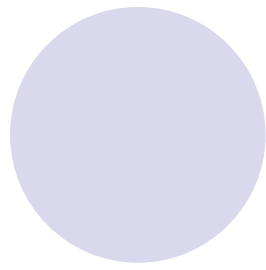


# Prototype *Motivation*



# Prototype Structure

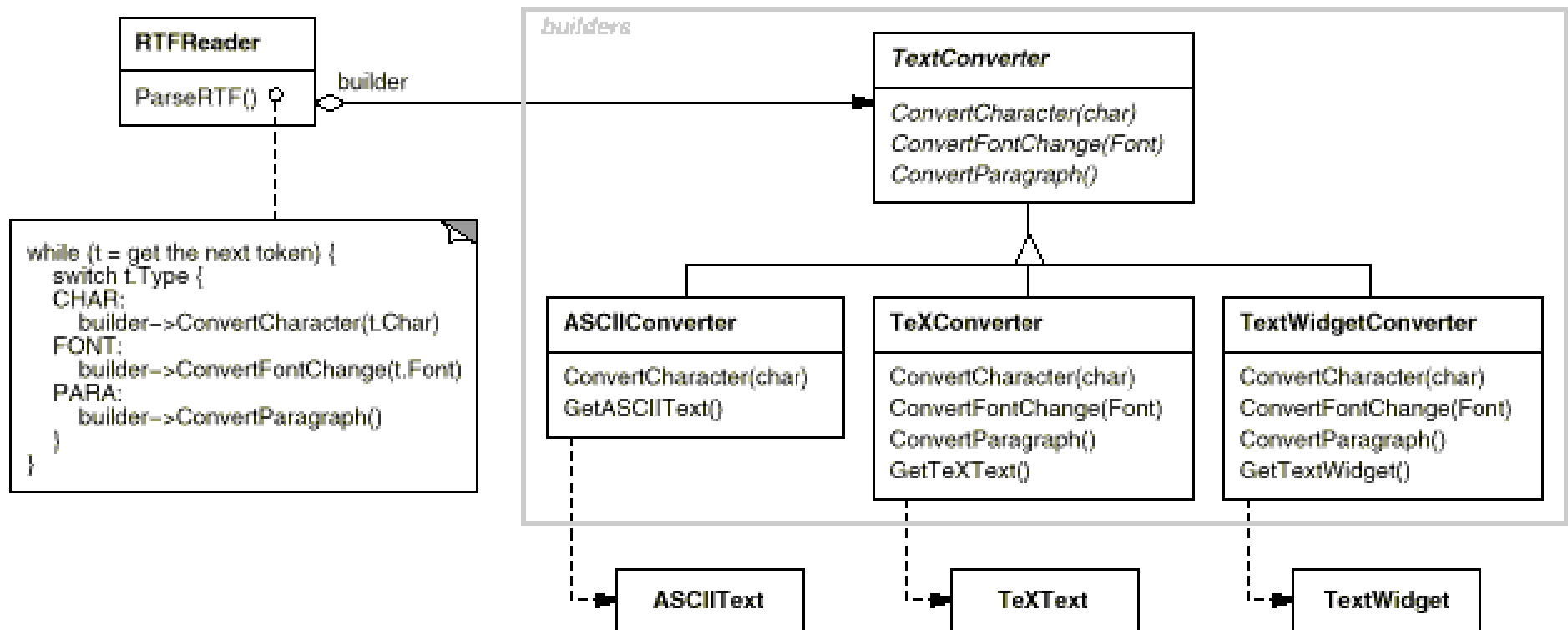




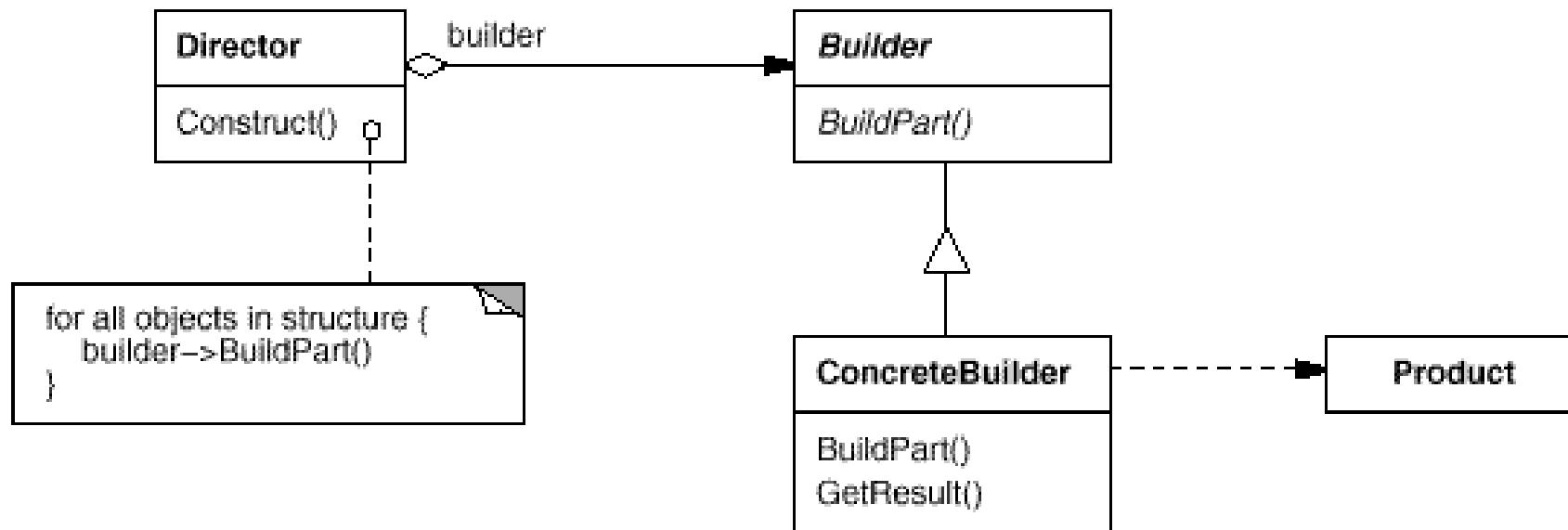
**Builder**

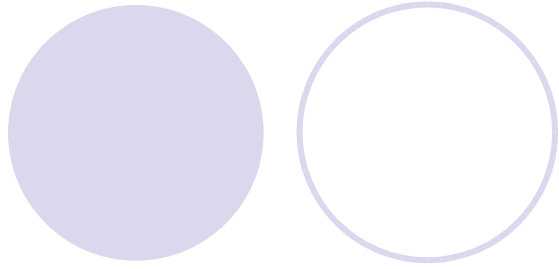
- *Intention* :
  - Dissocie la construction d'un objet complexe de sa représentation de manière à ce que le même processus de construction permette des représentations différentes
- *Indications* :
  - Quand l'algorithme de création d'un objet complexe doit être indépendant des parties qui composent l'objet et de la manière dont les parties sont agencées
  - Le processus de construction doit autoriser des représentations différentes de l'objet en construction

# Builder *Motivation*



# Builder Structure

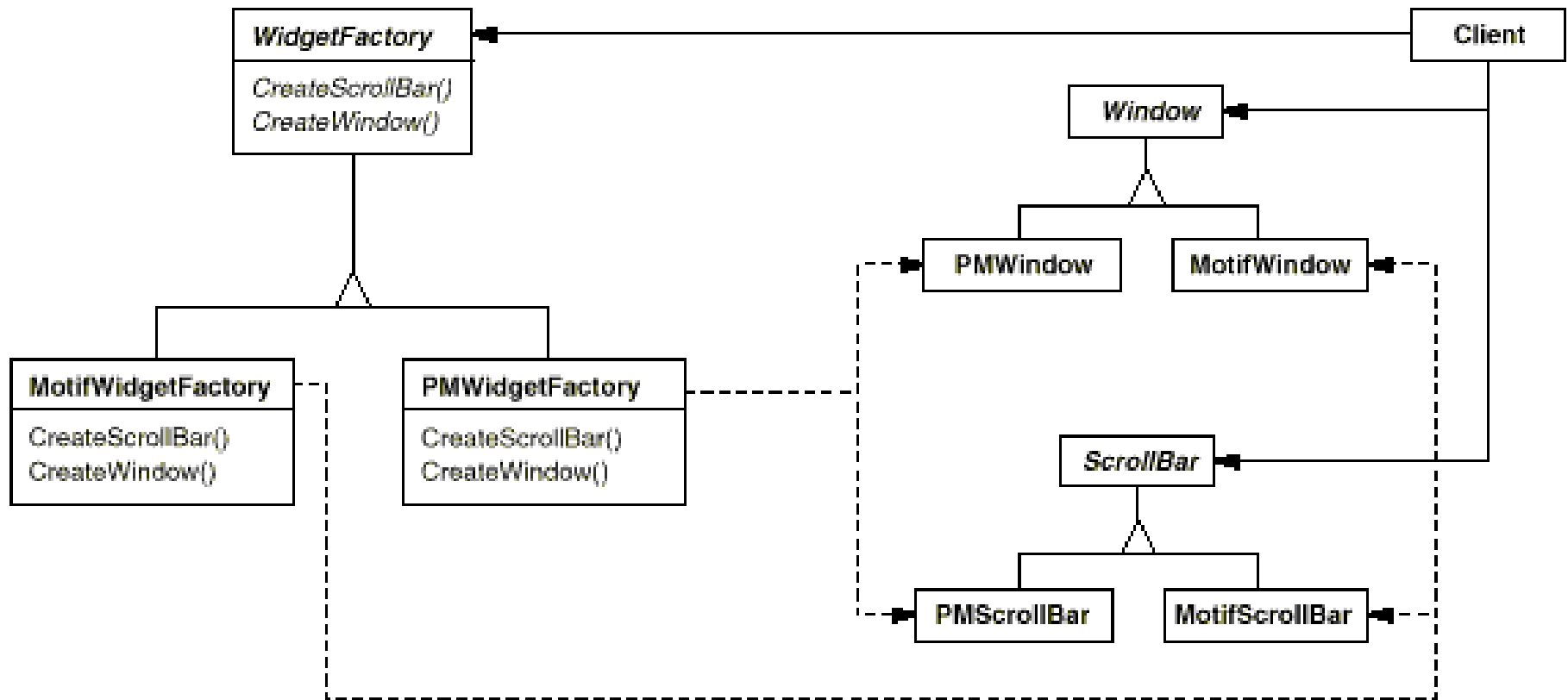




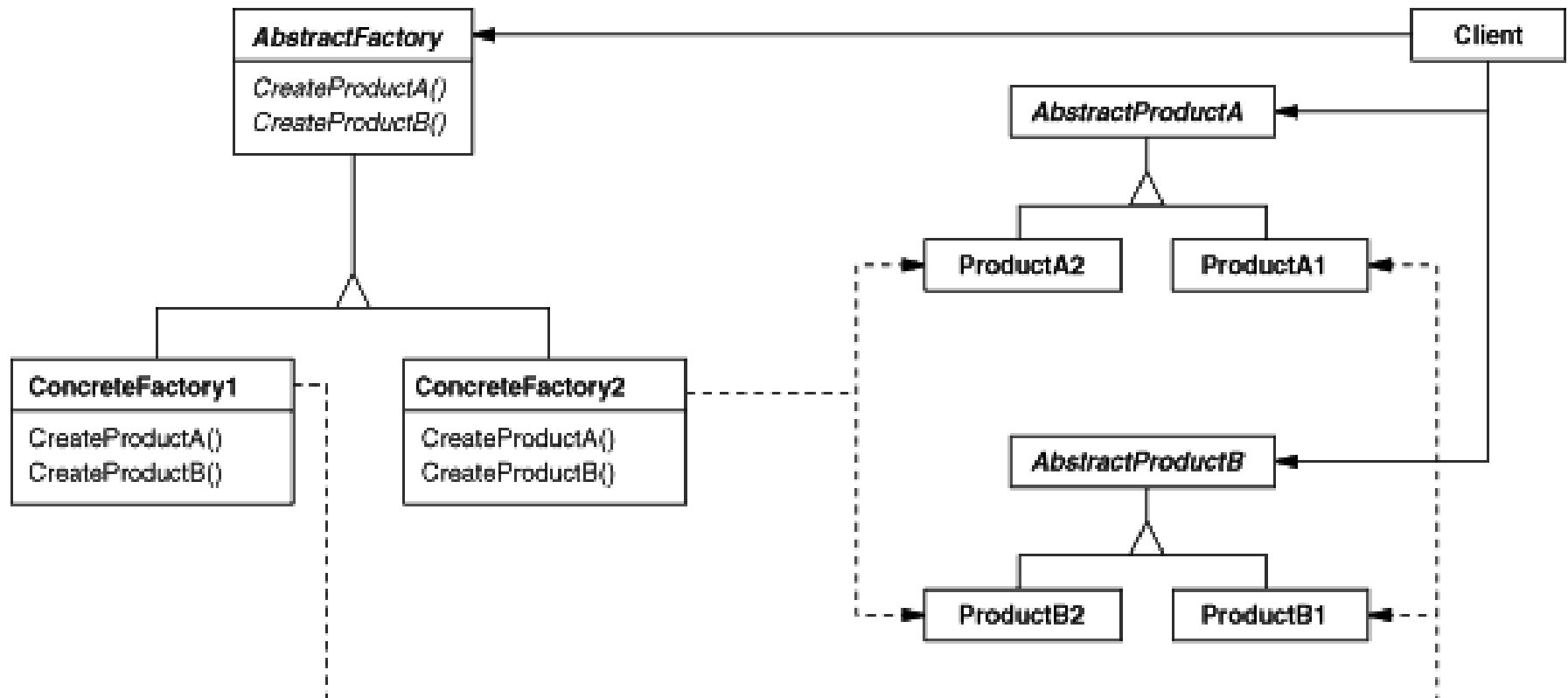
- *Intention* :
  - Fournir une interface pour la création d'objets apparentés ou interdépendants sans qu'il soit nécessaire de spécifier leurs classes concrètes.
- *Indications* :
  - Lorsqu'un système doit être indépendant de la façon dont ses produits ont été créés, combinés et représentés.
  - Si un système doit être constitué à partir d'une famille de produits, parmi plusieurs.
  - Quand on souhaite renforcer l'aspect « communauté » d'un ensemble de produits conçus pour être utilisés ensemble.

# Abstract Factory

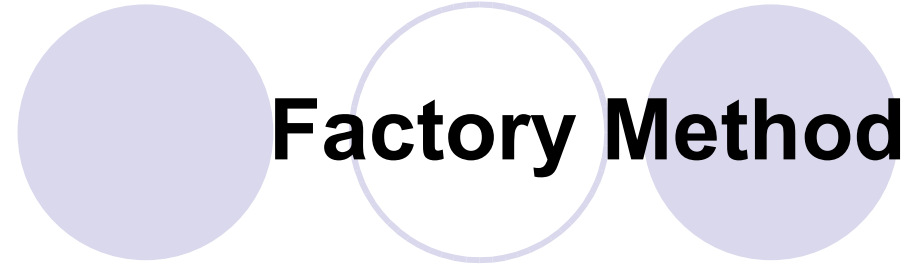
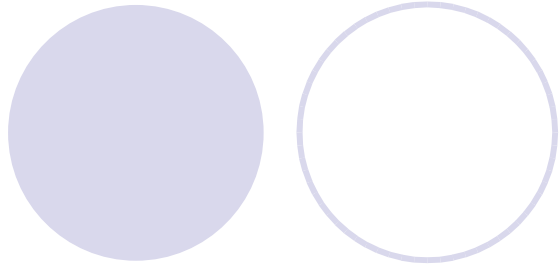
## *Motivation*



# Abstract Factory *Structure*



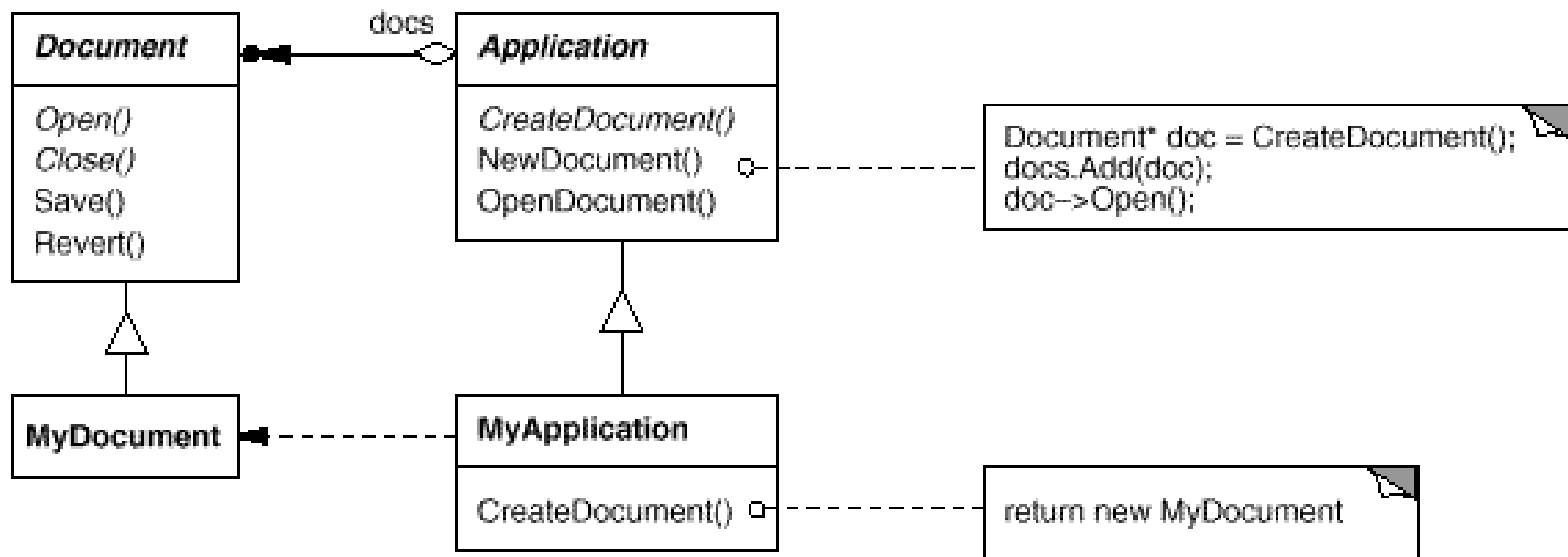




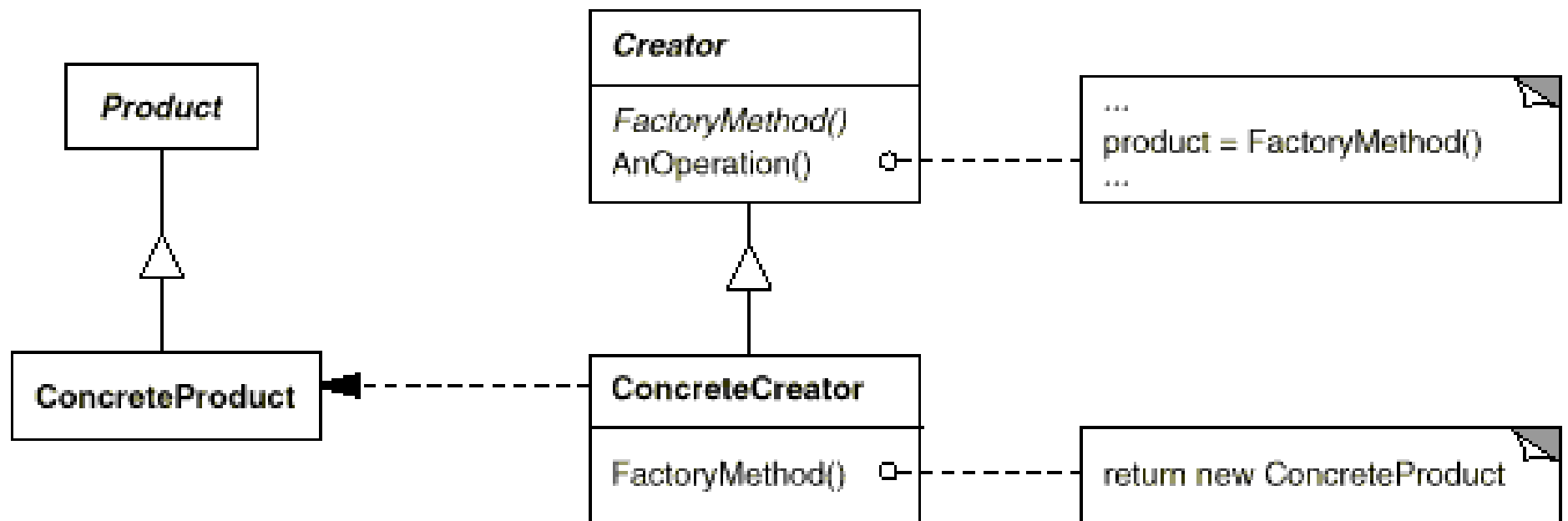
## Factory Method

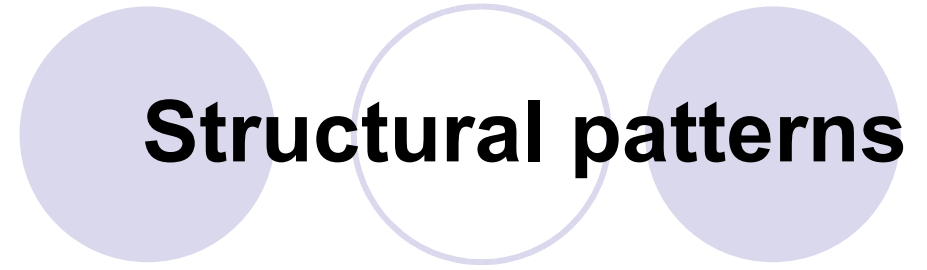
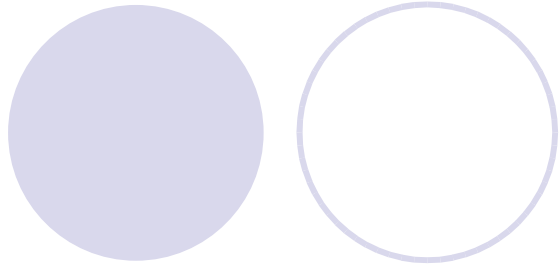
- *Intention* :
  - Définir une interface pour la création d'un objet, mais laisser à des sous-classes le choix des classes à instancier.
  - Permettre de déléguer l'instanciation à des sous classes.
- *Indications* :
  - Une classe ne peut prévoir la classe des objets qu'elle aura à créer.
  - Une classe attend des sous-classes qu'elles spécifient les objets qu'elle crée.
  - Les classes délèguent des responsabilités et on veut connaître localement la sous-classe assistante qui a reçu la délégation.

# Factory Method *Motivation*



# Factory Method *Structure*





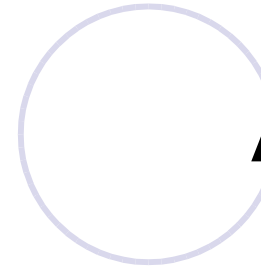
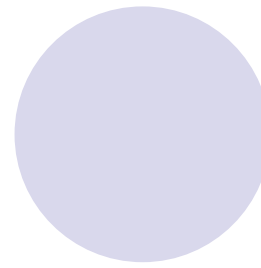
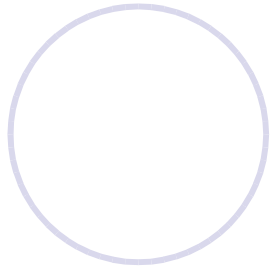
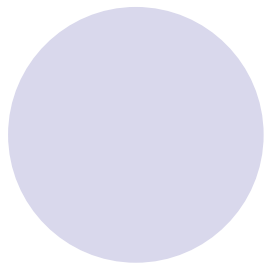
# Structural patterns

- Formes de structure :
  - Comment les objets sont assemblés
  - Les patterns sont complémentaires les uns des autres



# Exemples de Structural patterns

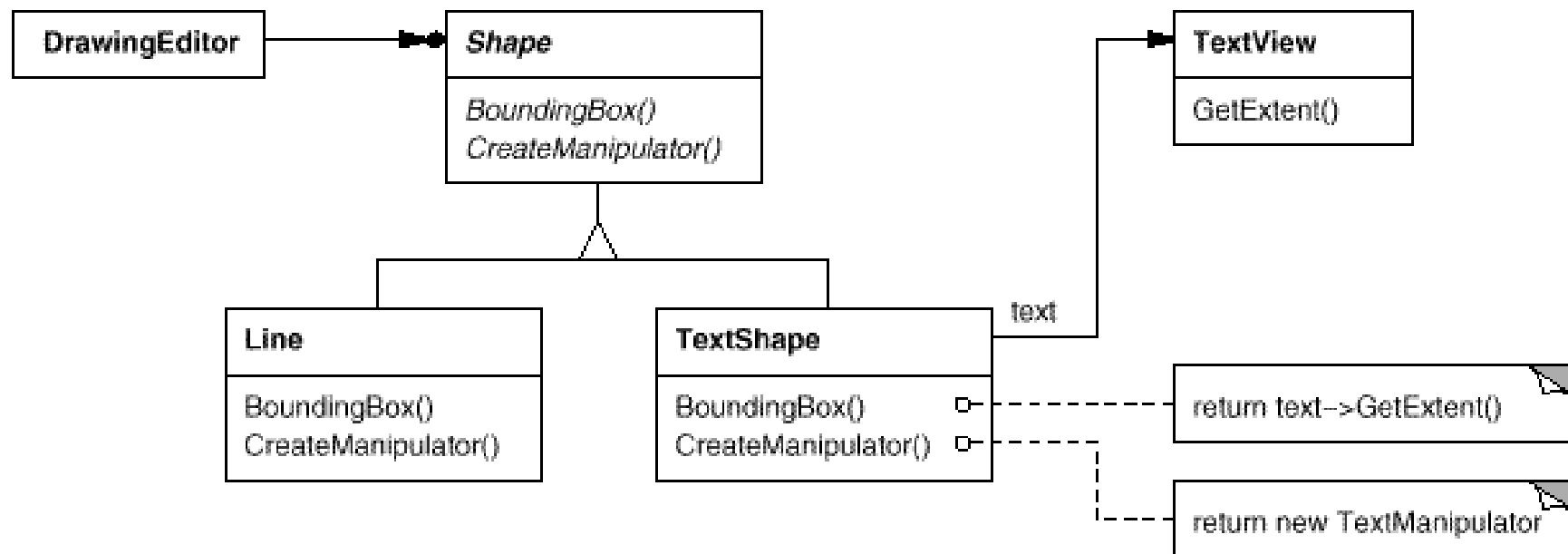
- *Adapter* pour rendre un objet conforme à un autre
- *Bridge* pour lier une abstraction à une implantation
- *Composite* : basé sur des objets primitifs et composants
- *Decorator* pour ajouter des services à un objet
- *Facade* pour cacher une structure complexe
- *Flyweight* pour de petits objets destinés à être partagés
- *Proxy* quand un objet en masque un autre



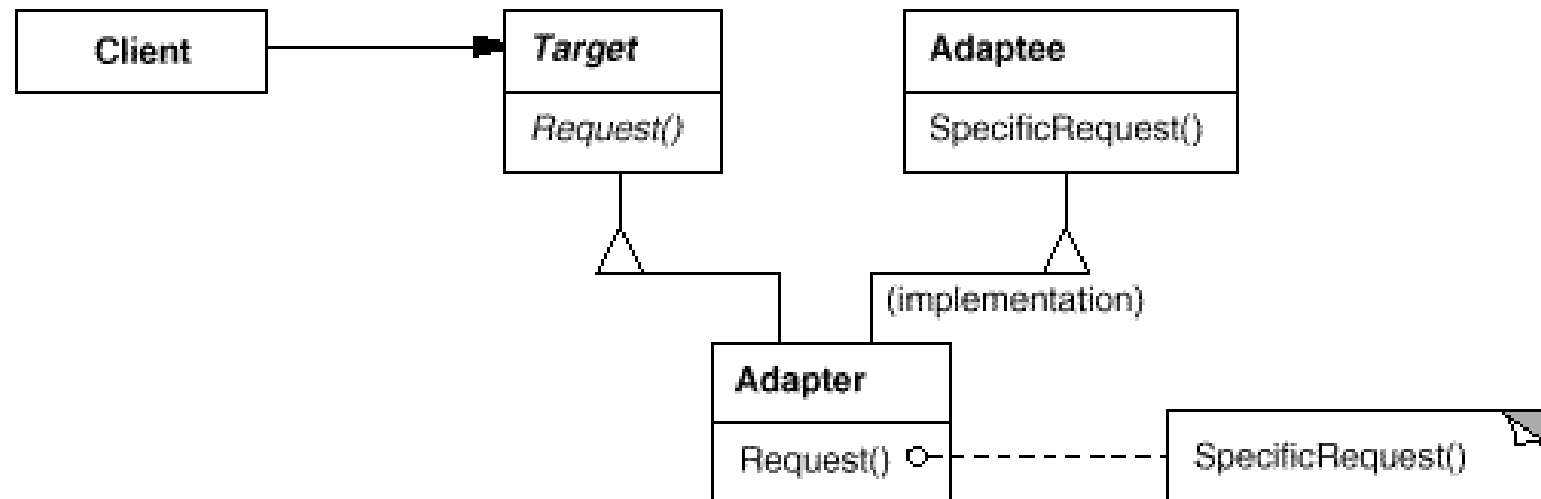
**Adapter**

- *Intention* :
  - Convertir l'interface d'une classe en une autre qui soit conforme aux attentes d'un client. L'adaptateur permet à des classes de collaborer malgré des interfaces incompatibles.
- *Indications* :
  - On veut utiliser une classe existante, mais son interface ne coïncide pas avec celle escomptée.
  - On souhaite créer une classe réutilisable qui collaborera avec des classes encore inconnues, mais qui n'auront pas nécessairement l'interface escomptée

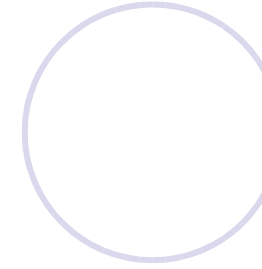
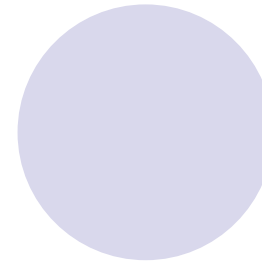
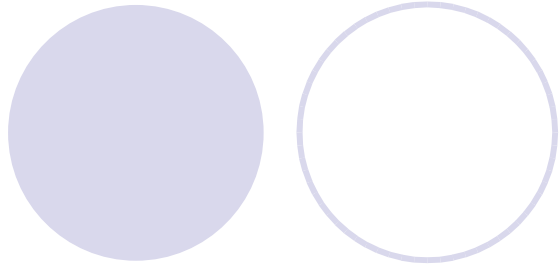
# Adapter *Motivation*



# Adapter Structure

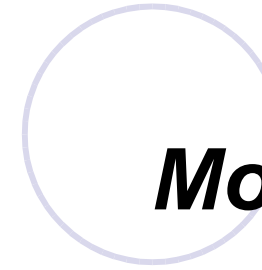
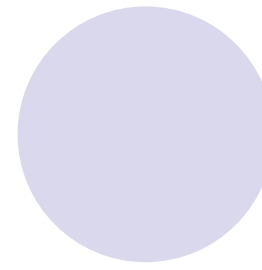
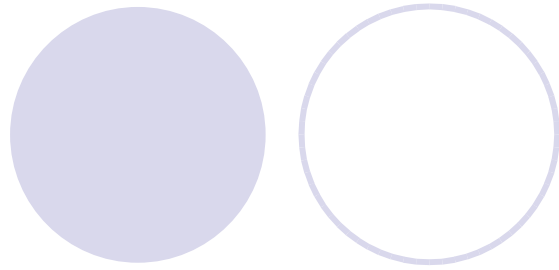




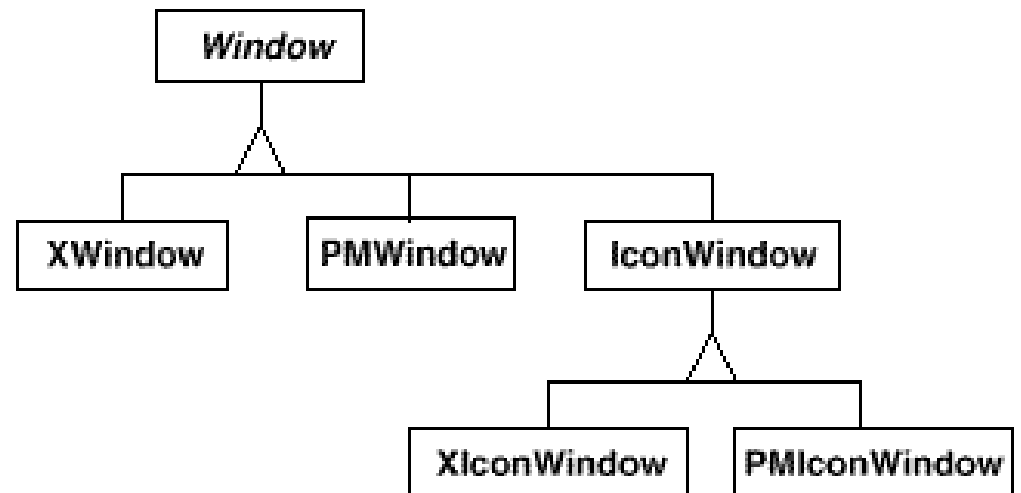
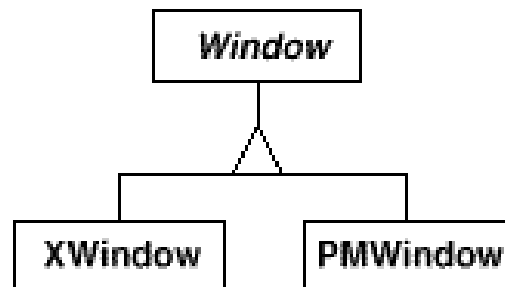


**Bridge**

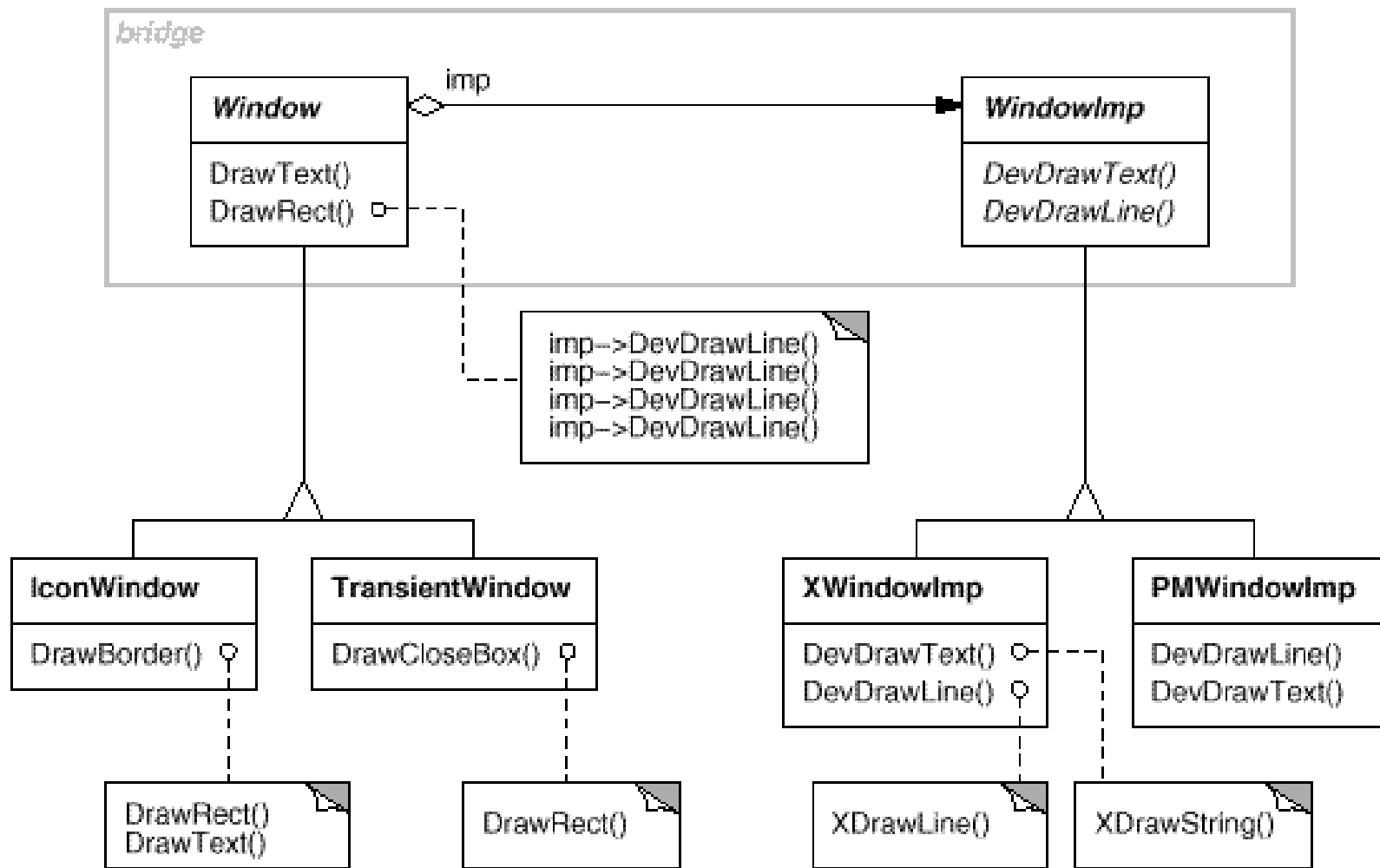
- *Intention* :
  - Découple une abstraction de son implémentation afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre
- *Indications* :
  - On souhaite éviter un lien définitif entre une abstraction et son implémentation, les deux devant pouvoir être étendues par héritage
  - Les modifications de l'implémentation d'une abstraction ne doivent pas avoir de conséquence sur le code client (pas de recompilation)



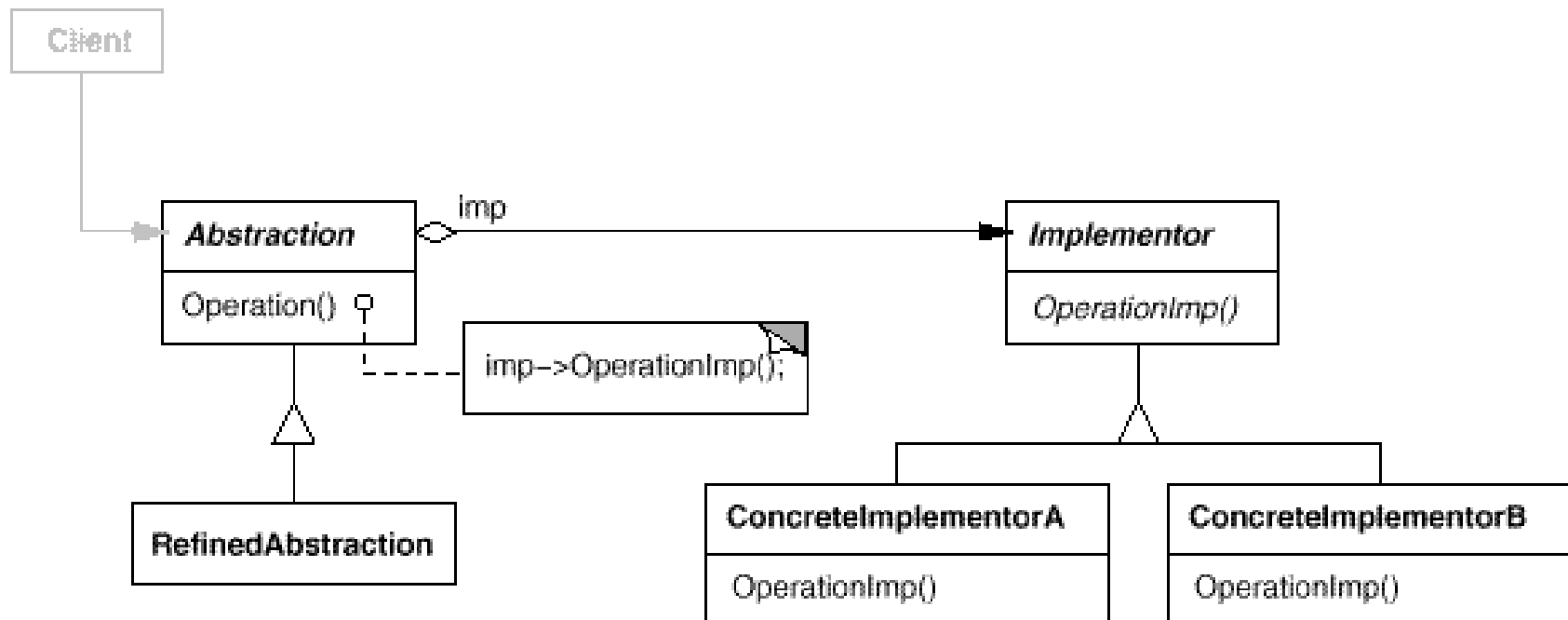
# Bridge *Motivation*

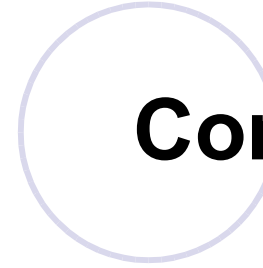
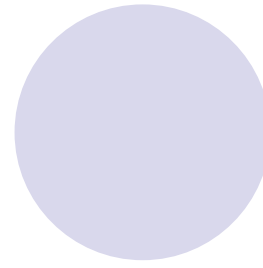
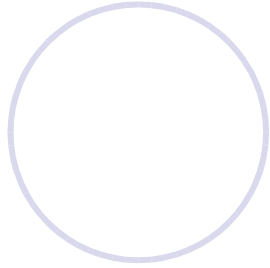
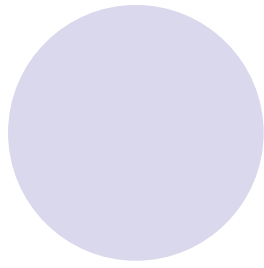


# Bridge *Motivation*



# Bridge Structure

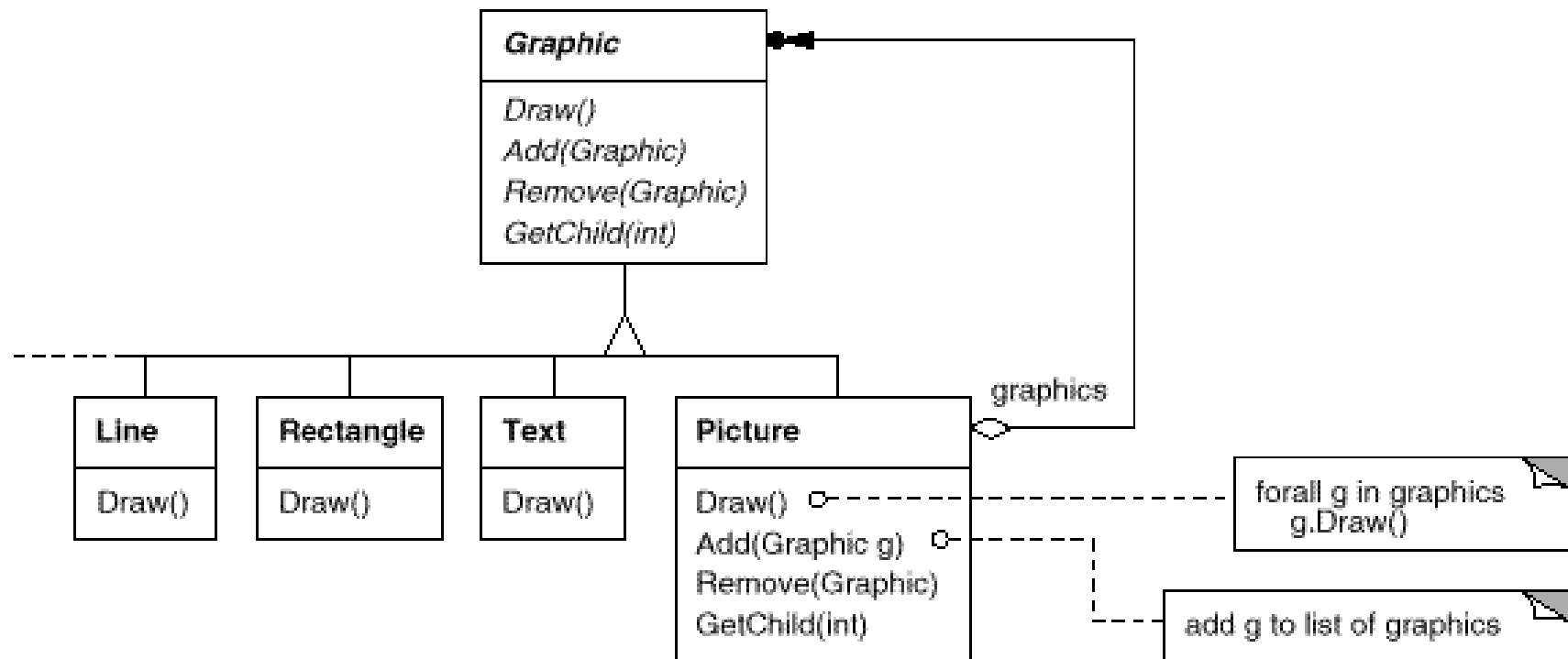




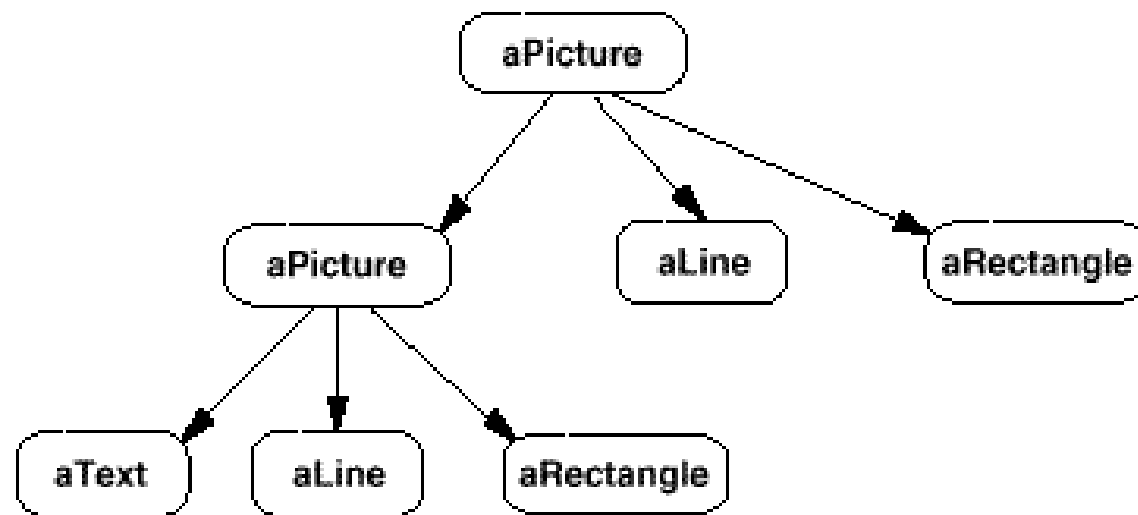
**Composite**

- *Intention* :
  - Composition d'objets en structures arborescentes pour représenter des structures composant/composé.
  - Permettre au client de traiter de la même manière les objets atomiques et les combinaisons de ceux-ci.
- *Indications* :
  - On souhaite représenter des hiérarchies d'individus à l'ensemble
  - On souhaite que le client n'ait pas à se préoccuper de la différence entre combinaisons d'objets et objets individuels

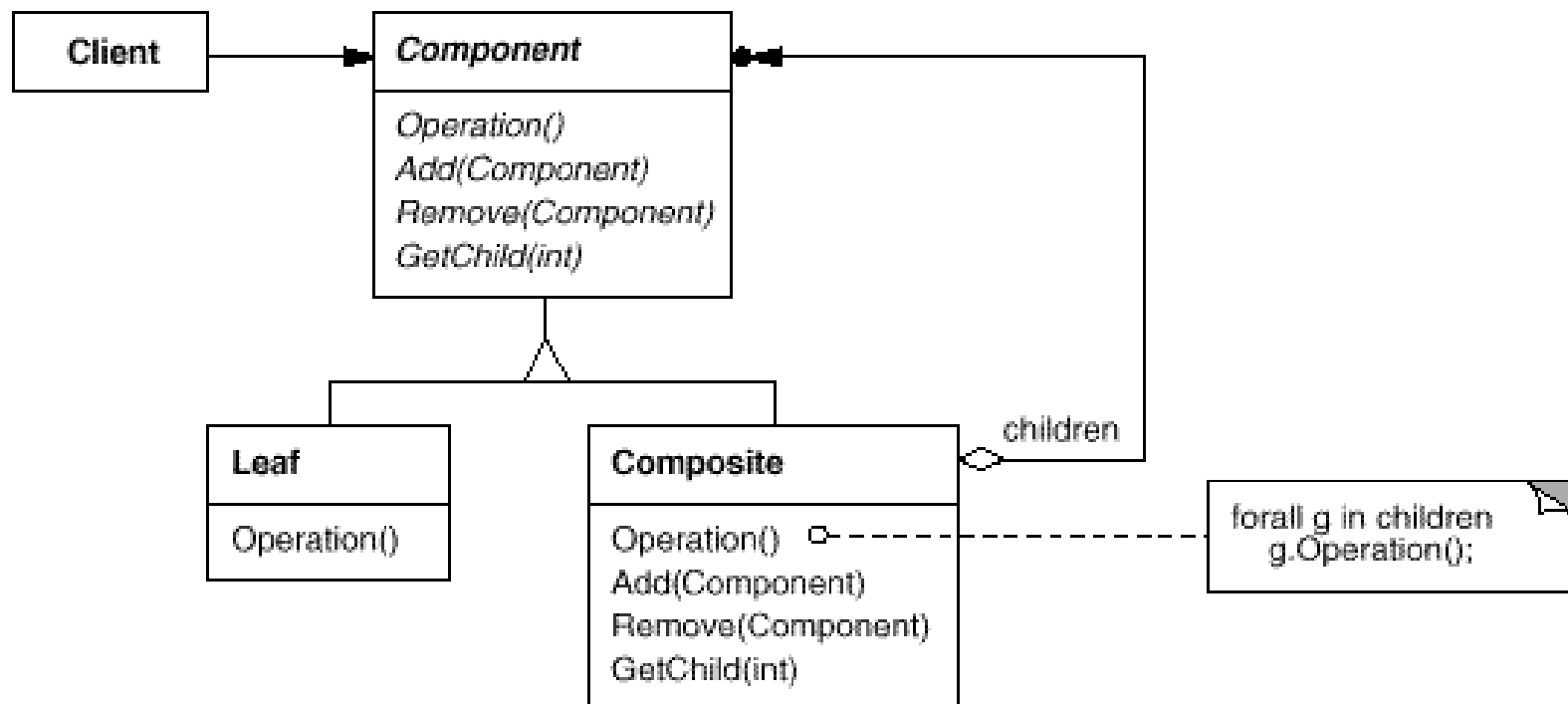
# Composite *Motivation*



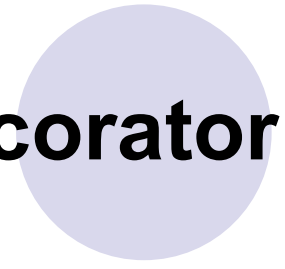
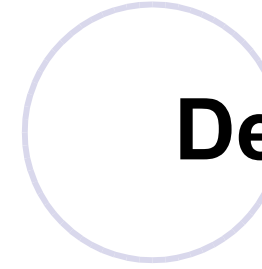
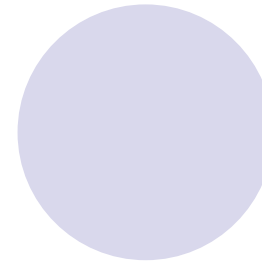
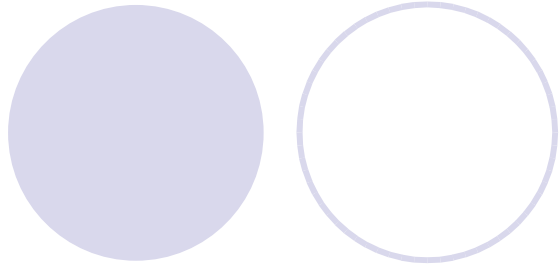
# Composite *Motivation*



# Composite Structure





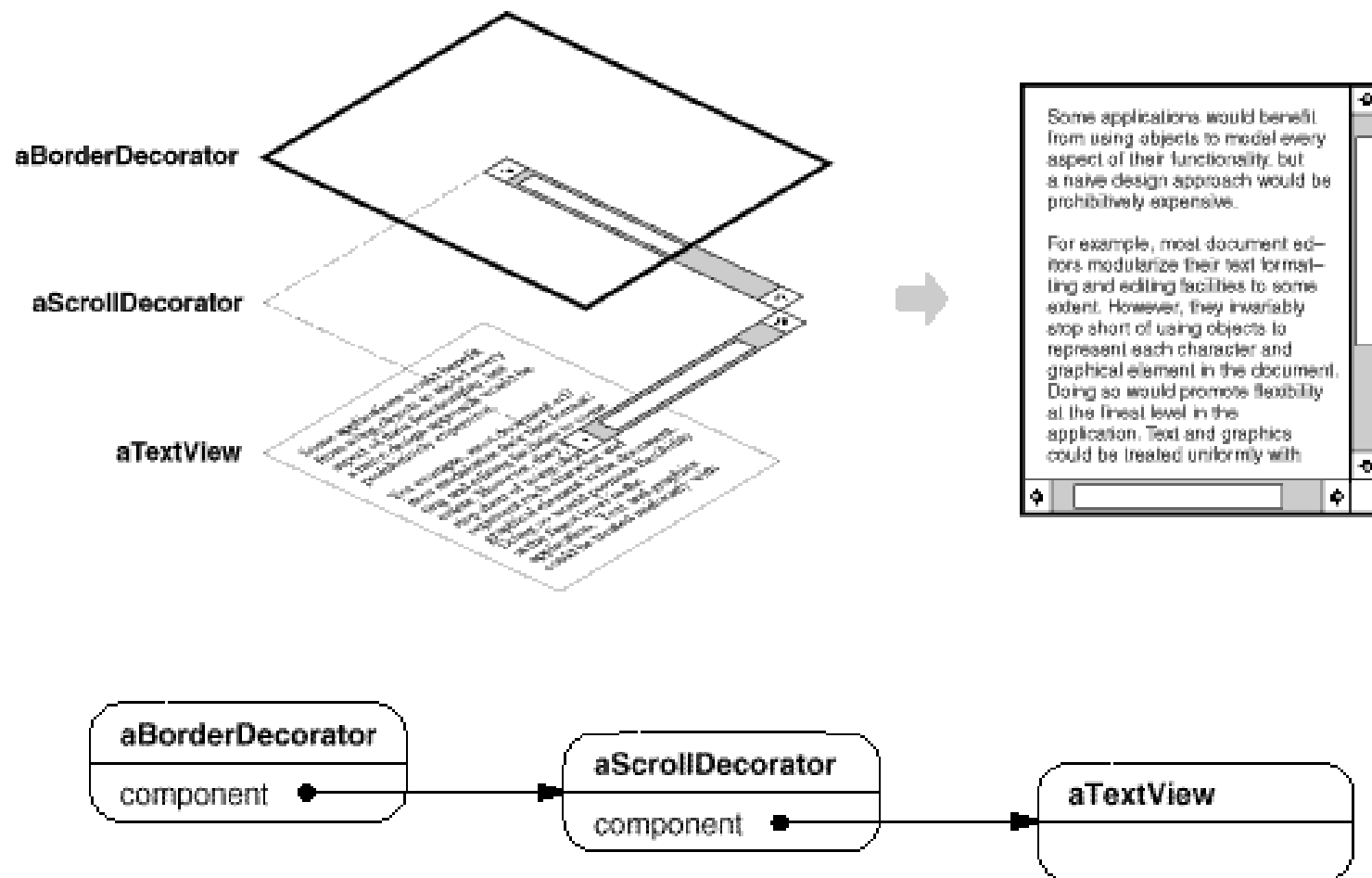


# Decorator

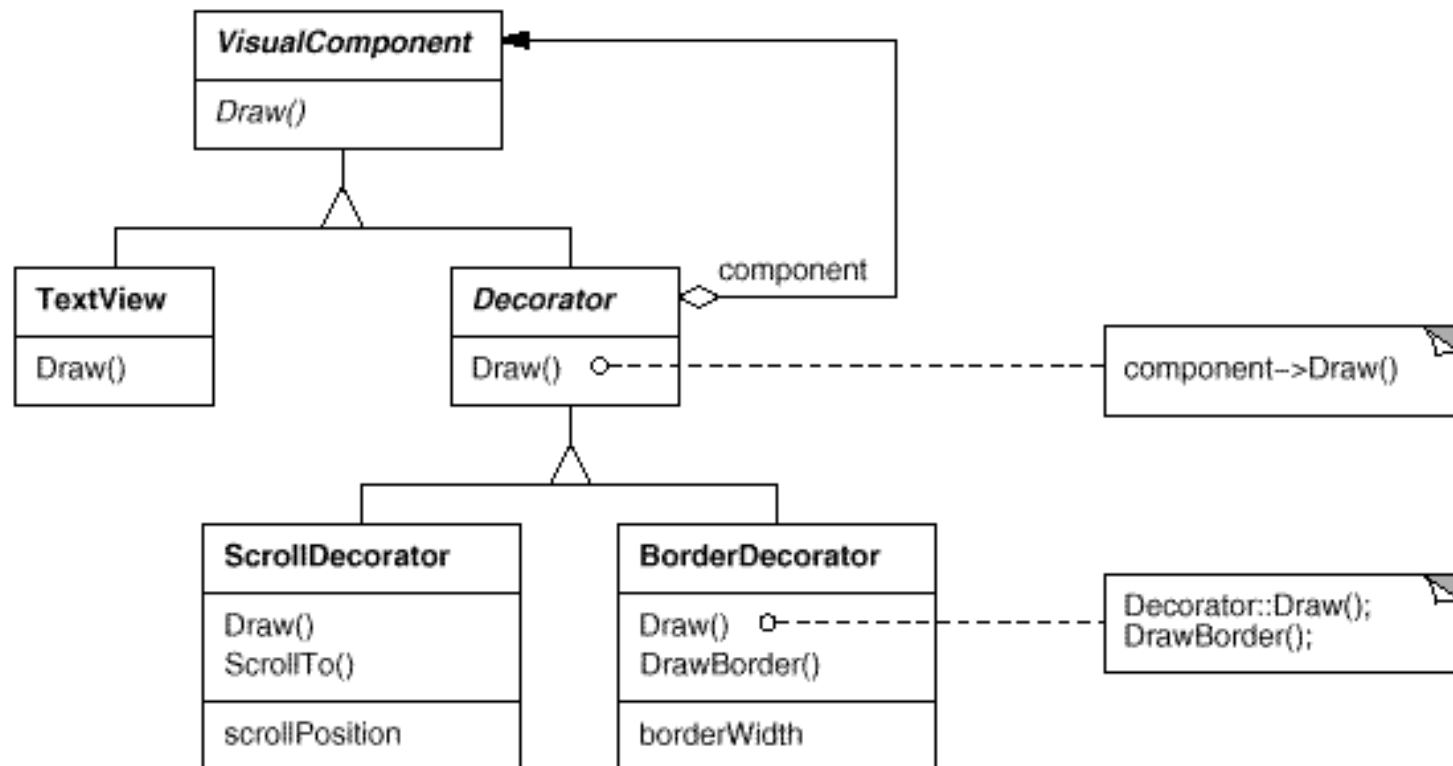
- *Intention* :
  - Attacher dynamiquement de nouvelles responsabilités à un objet.
  - Alternative souple à l'héritage.
- *Indications* :
  - Pour ajouter dynamiquement de nouvelles responsabilités à un objet de manière transparente.
  - Pour des responsabilités qui peuvent être retirées.
  - Dans certains cas, pour éviter la prolifération de sous-classes.

# Decorator

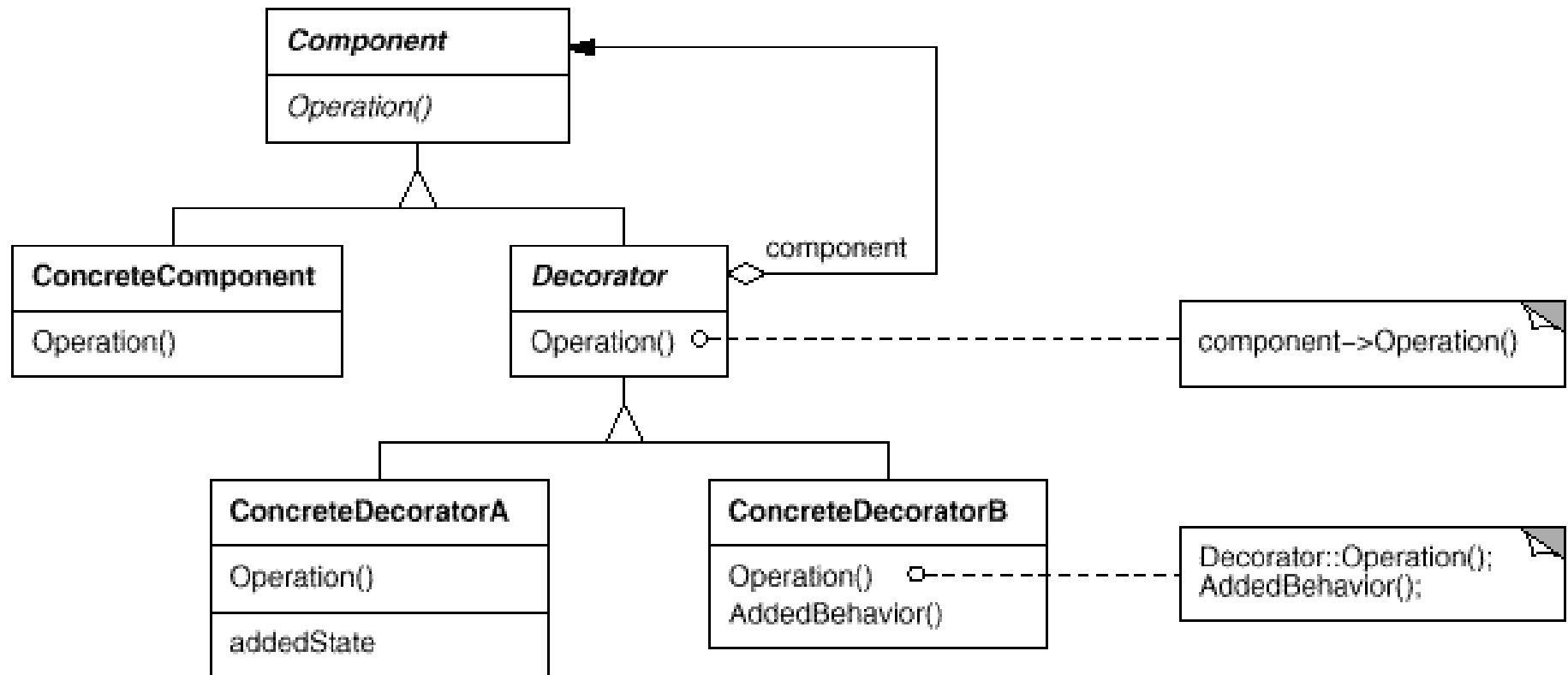
## *Motivation*

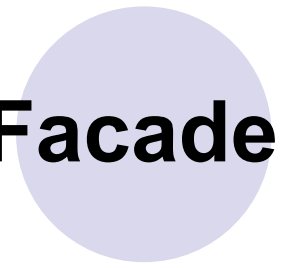
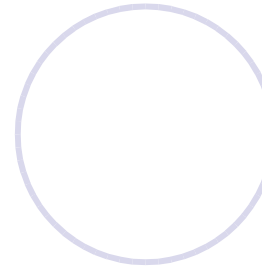
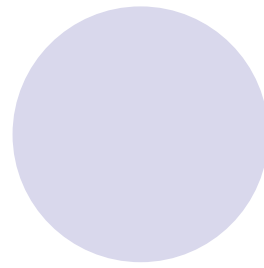
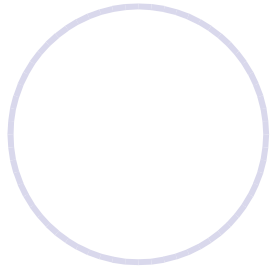
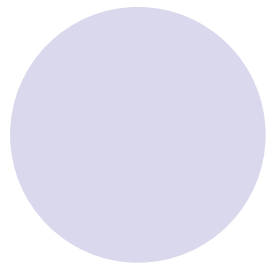


# Decorator *Motivation*



# Decorator Structure





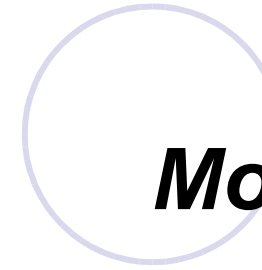
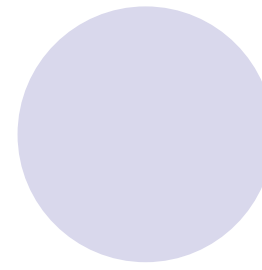
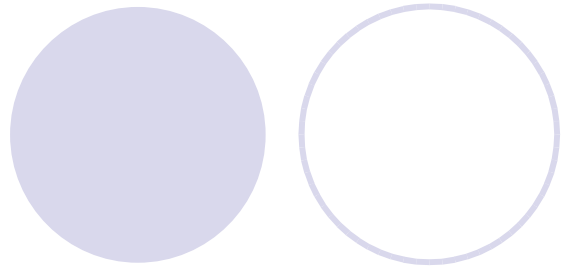
**Facade**

- *Intention* :

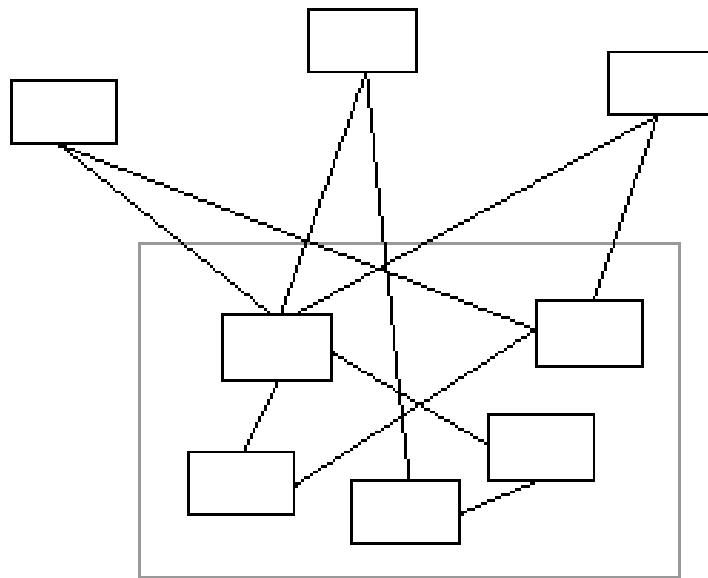
- Fournir une interface unifiée à l'ensemble des interfaces d'un sous-système.
- La façade fournit une interface de plus haut niveau, plus facile à utiliser.

- *Indications* :

- On souhaite disposer d'une interface simple pour un sous-système complexe.
- Pour découpler un sous-système ds clients de manière à favoriser la portabilité.
- Quand on cherche à structurer un sous-système en niveaux.



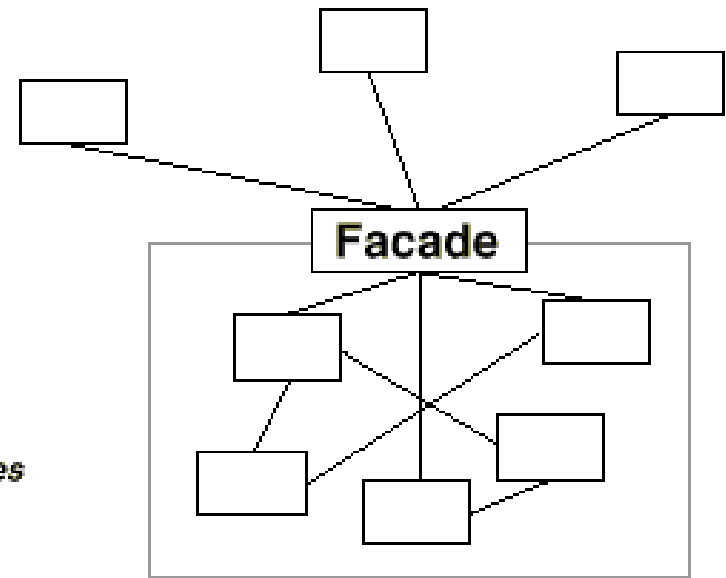
# Facade *Motivation*



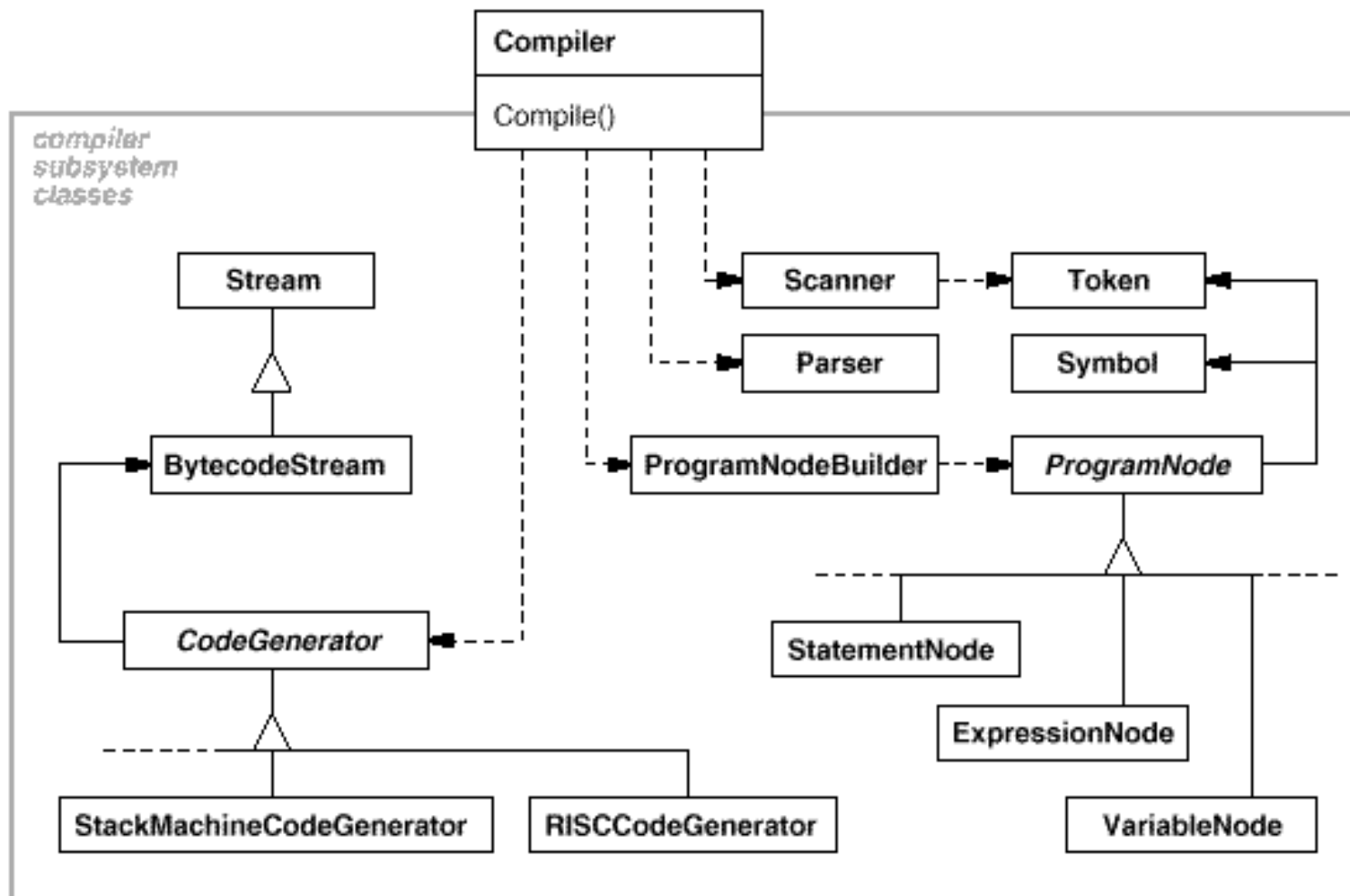
*client classes*



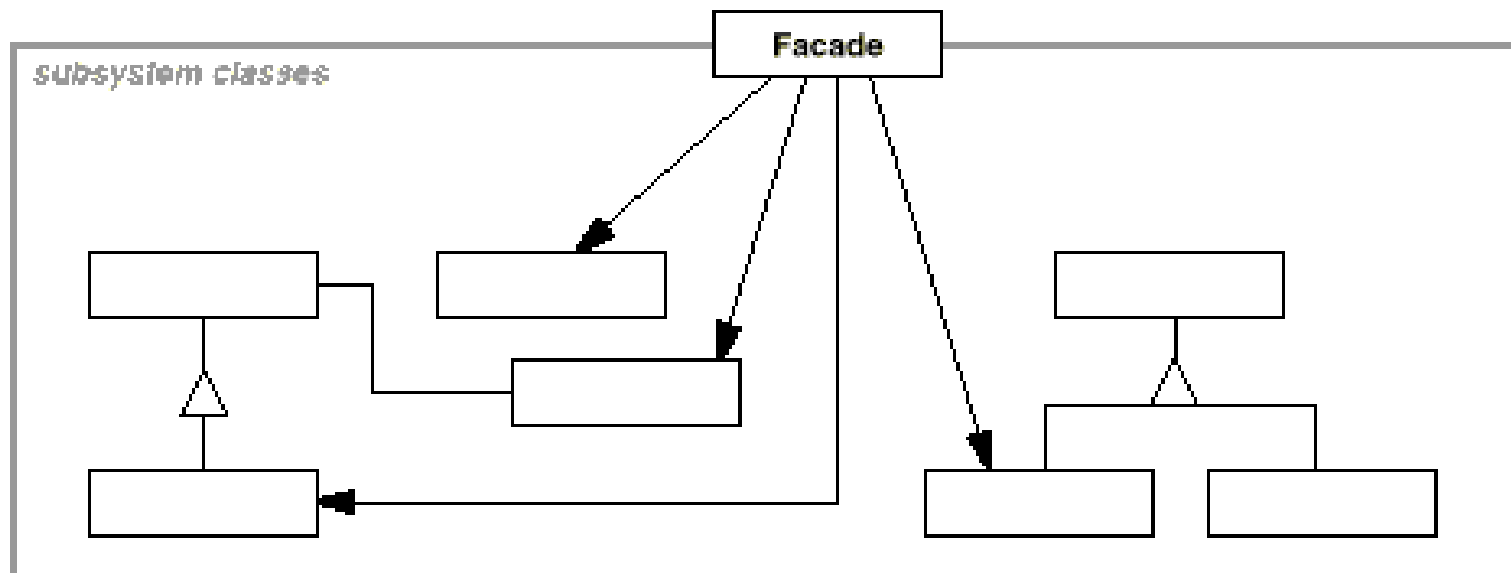
*subsystem classes*



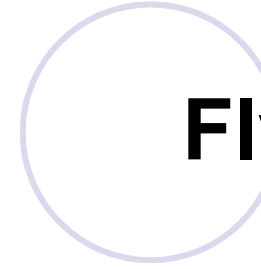
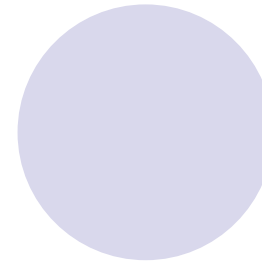
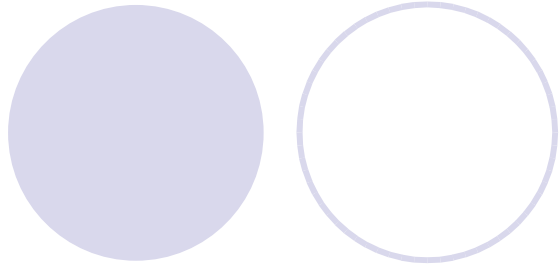
# Facade *Motivation*



# Facade Structure



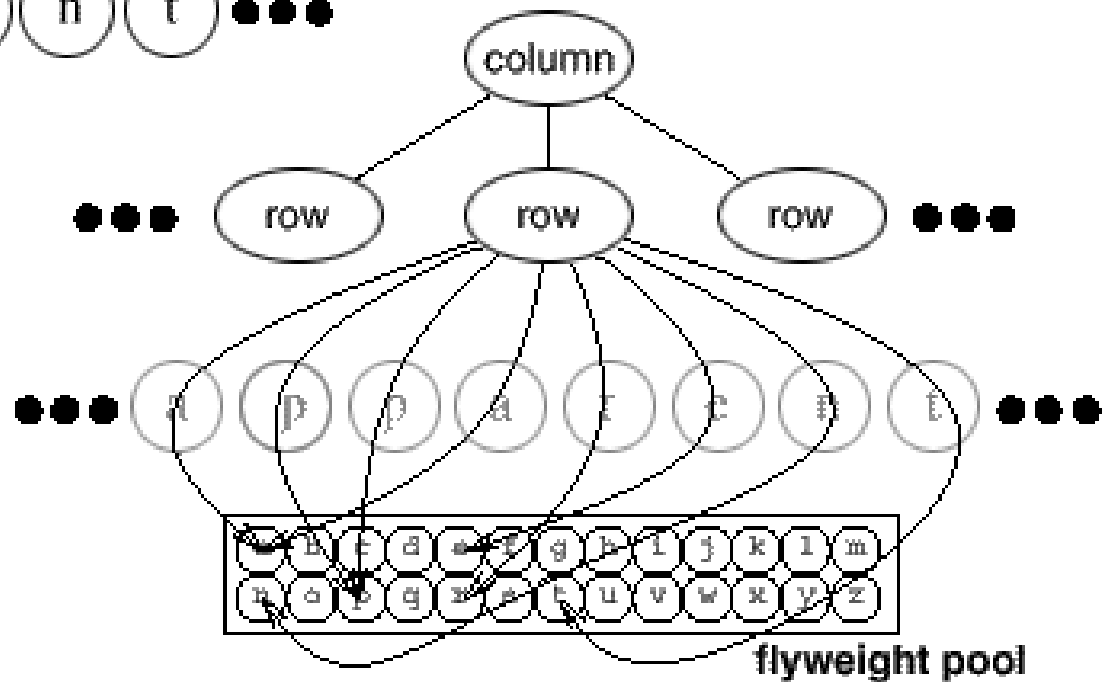
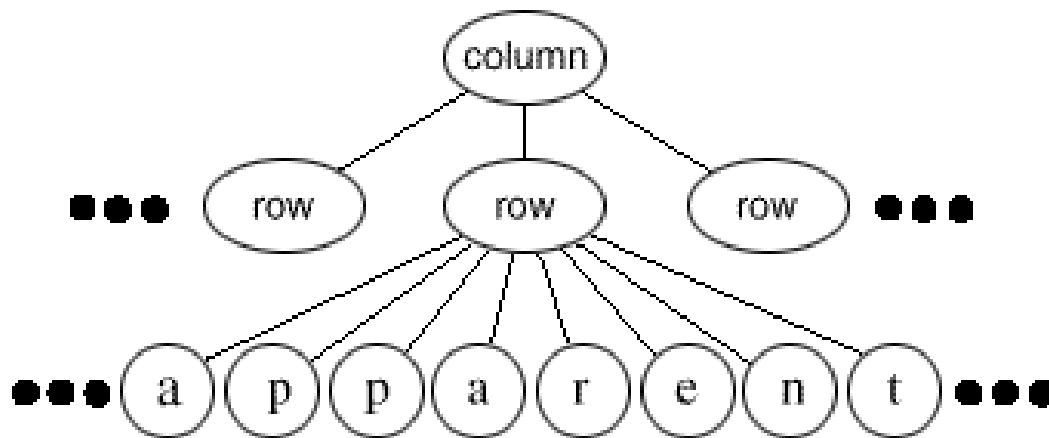




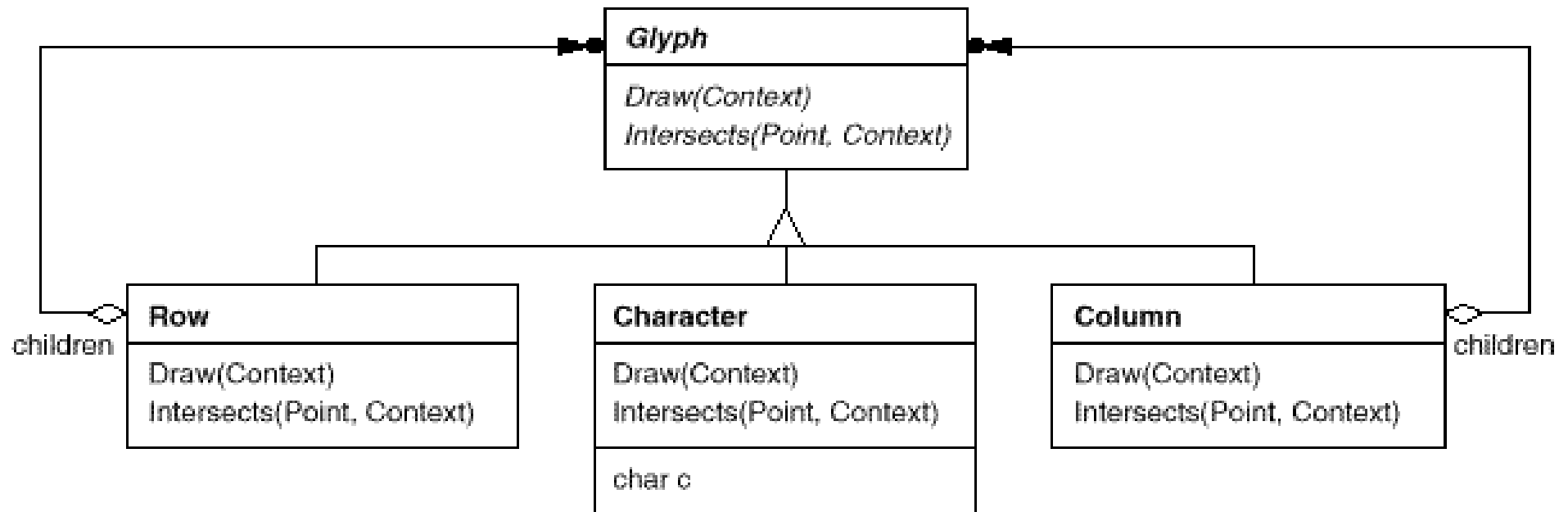
# Flyweight

- *Intention* :
  - Utiliser une technique de partage pour mettre en oeuvre de manière efficace un grand nombre d'objets de faible granularité.
- *Indications* :
  - L'application utilise un grand nombre d'objets.
  - Plusieurs groupes d'objets peuvent être remplacés par un nombre plus faible d'objets partagés.
  - L'application ne dépend pas de l'identité des objets particuliers.

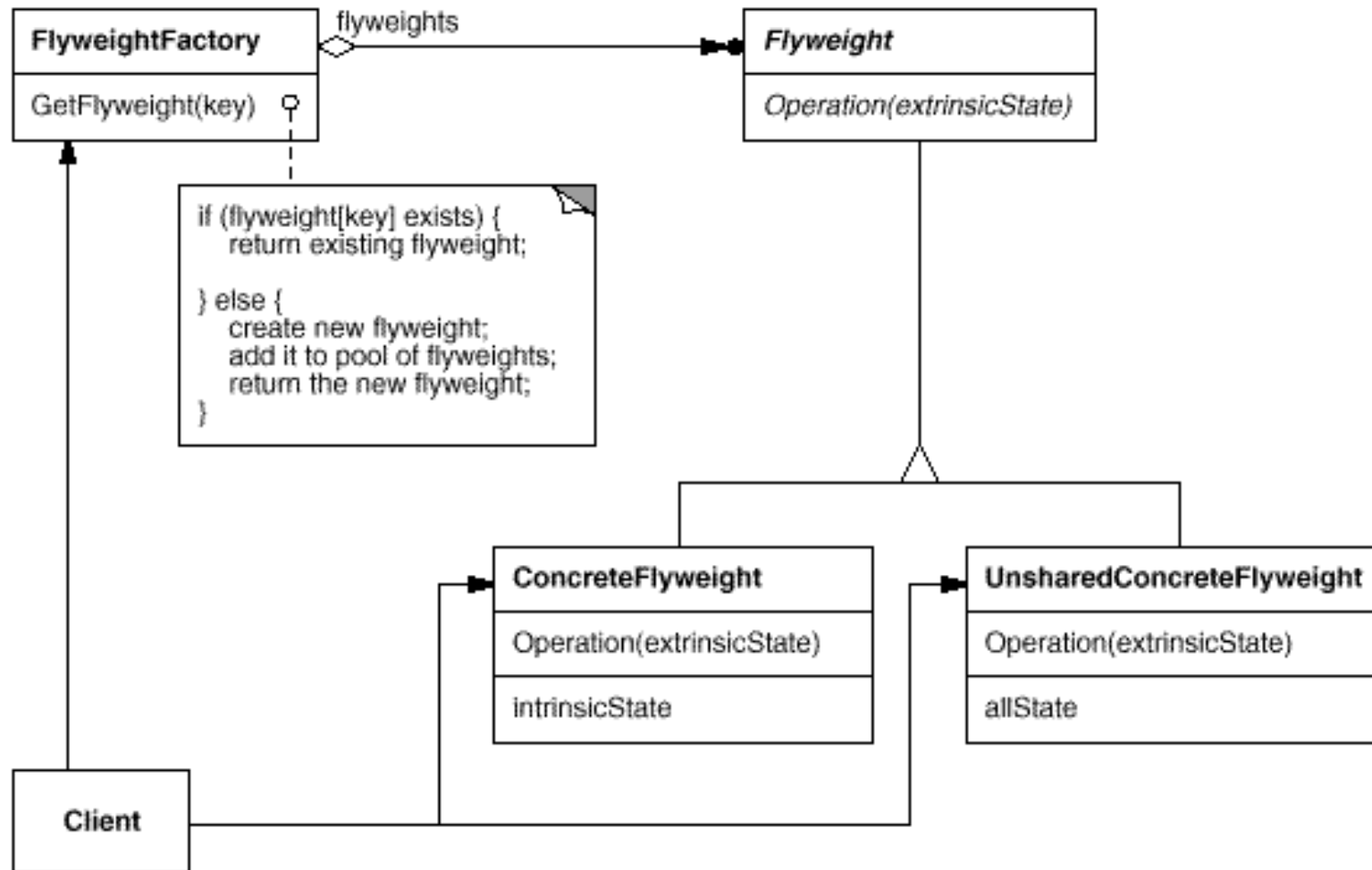
# Flyweight *Motivation*

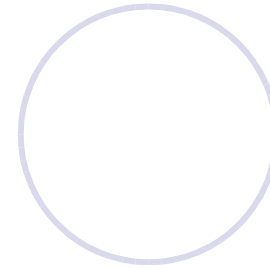
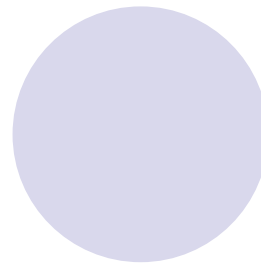
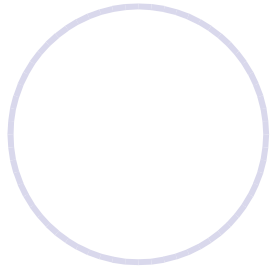
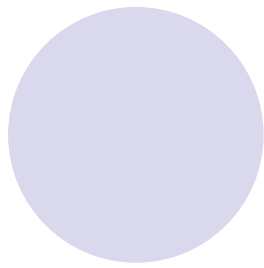


# Flyweight *Motivation*



# Flyweight Structure





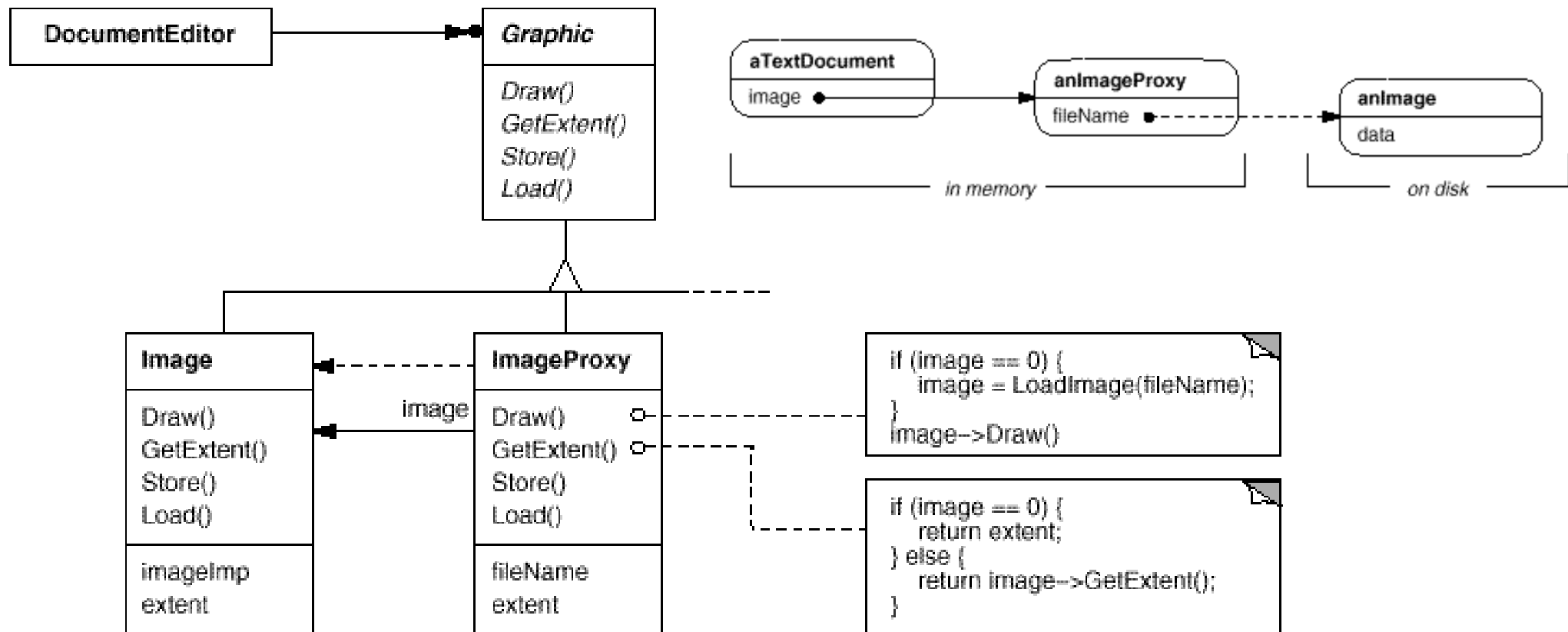
- *Intention* :

- Fournir à un objet tiers un mandataire ou un remplaçant, pour contrôler l'accès à cet objet.

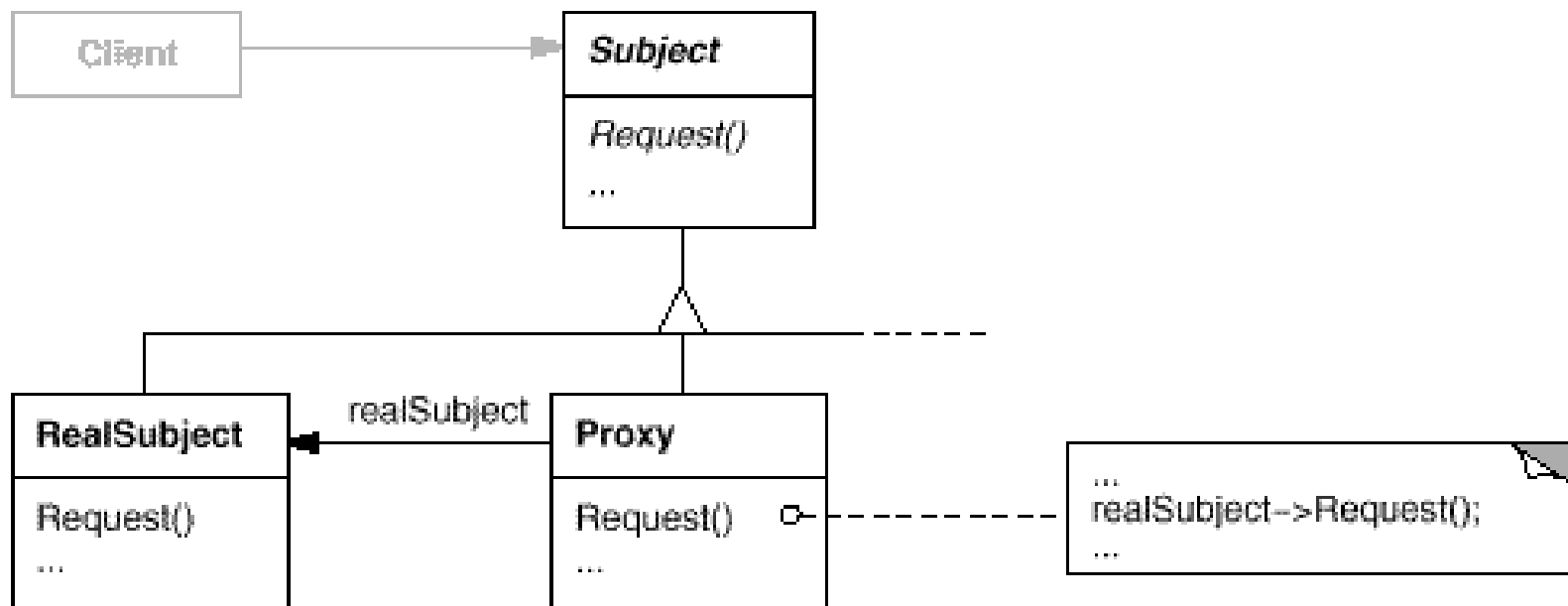
- *Indications* :

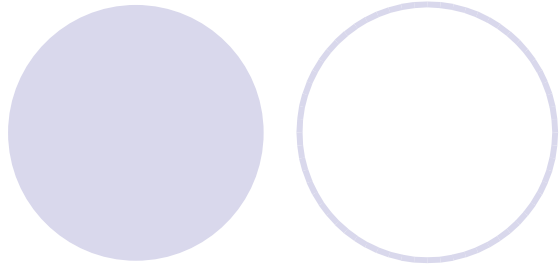
- Procuration à distance (représentant local d'un objet situé dans un espace d'adressage différent).
- Procuration de protection (contrôle de l'accès à l'objet original, en vertu de droits d'accès).
- Référence intelligente (pour réaliser des opérations supplémentaires lors des accès).

# Proxy Motivation



# Proxy Structure





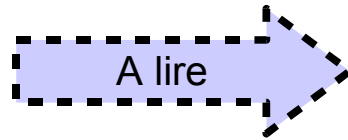
# Behavioural patterns

- Formes de comportement pour décrire :
  - des algorithmes
  - des comportements entre objets
  - des formes de communication entre objet

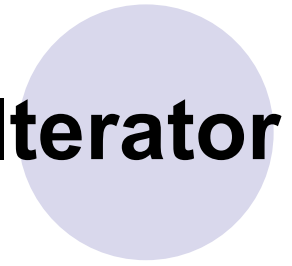
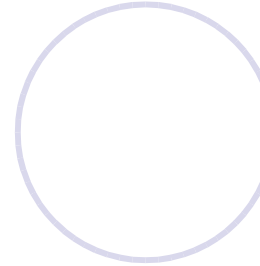
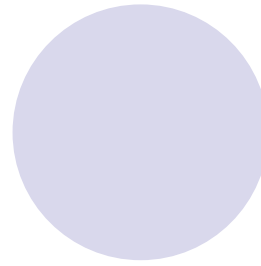
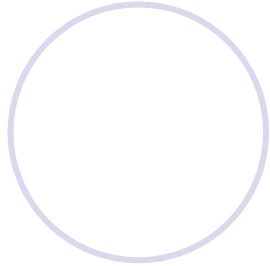
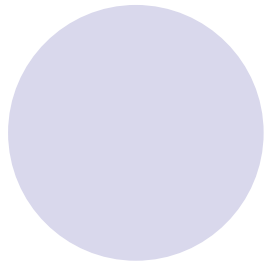


# Exemples de Behavioural patterns

- *Chain of Responsibility*
- *Command*
- *Interpreter*
- *Iterator*
- *Mediator*
- *Memento*
- *Observer*
- *State*
- *Strategy*
- *Template Method*
- *Visitor*



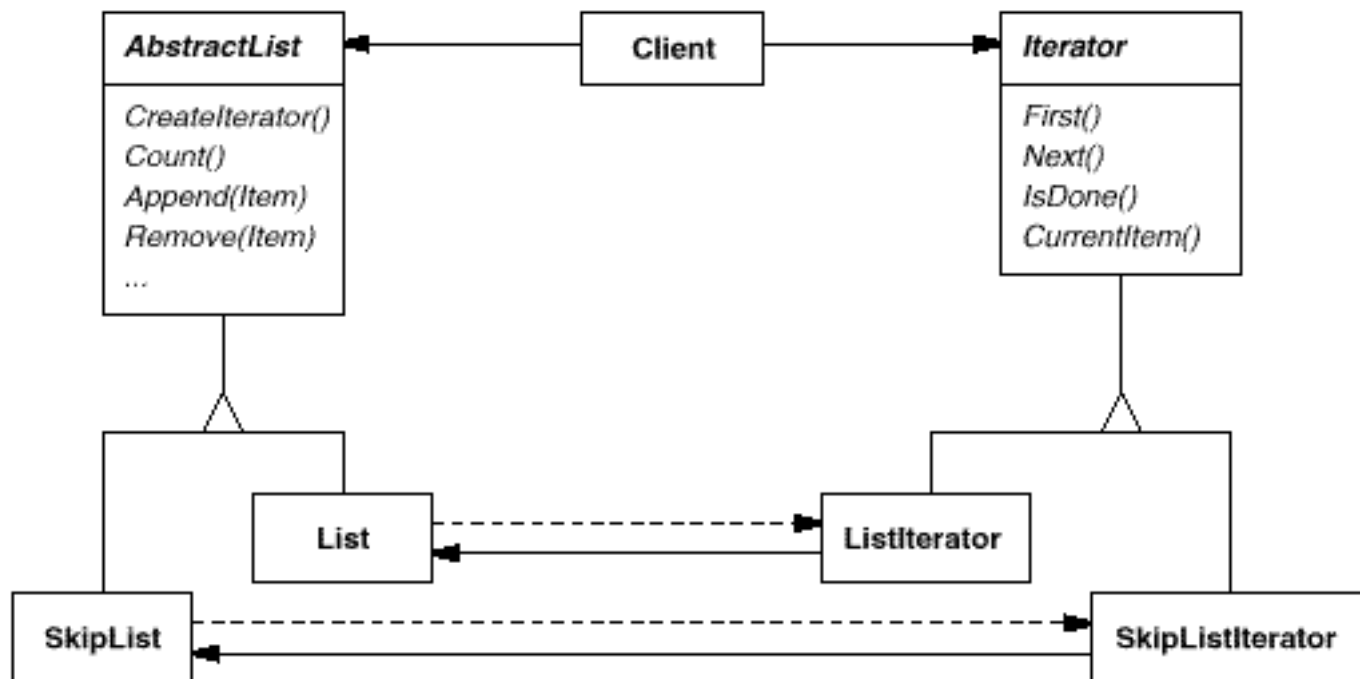
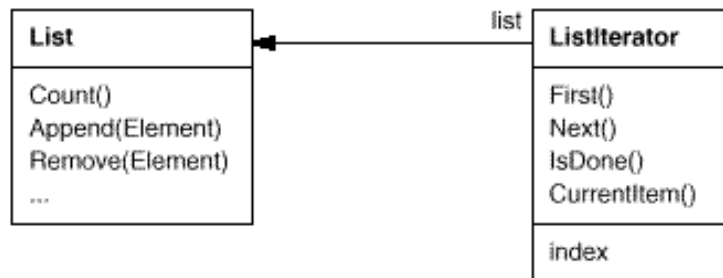
- « **Design patterns. Catalogue des modèles de conception réutilisables** »
  - *Richard Helm, Ralph Johnson, John Vlissides, Eric Gamma*
  - *Vuibert informatique*
- « **UML 2 et les Design Patterns** »
  - *Craig Larman, M.C. Baland, L. Carité, E. Burr*
  - *Pearson Education*



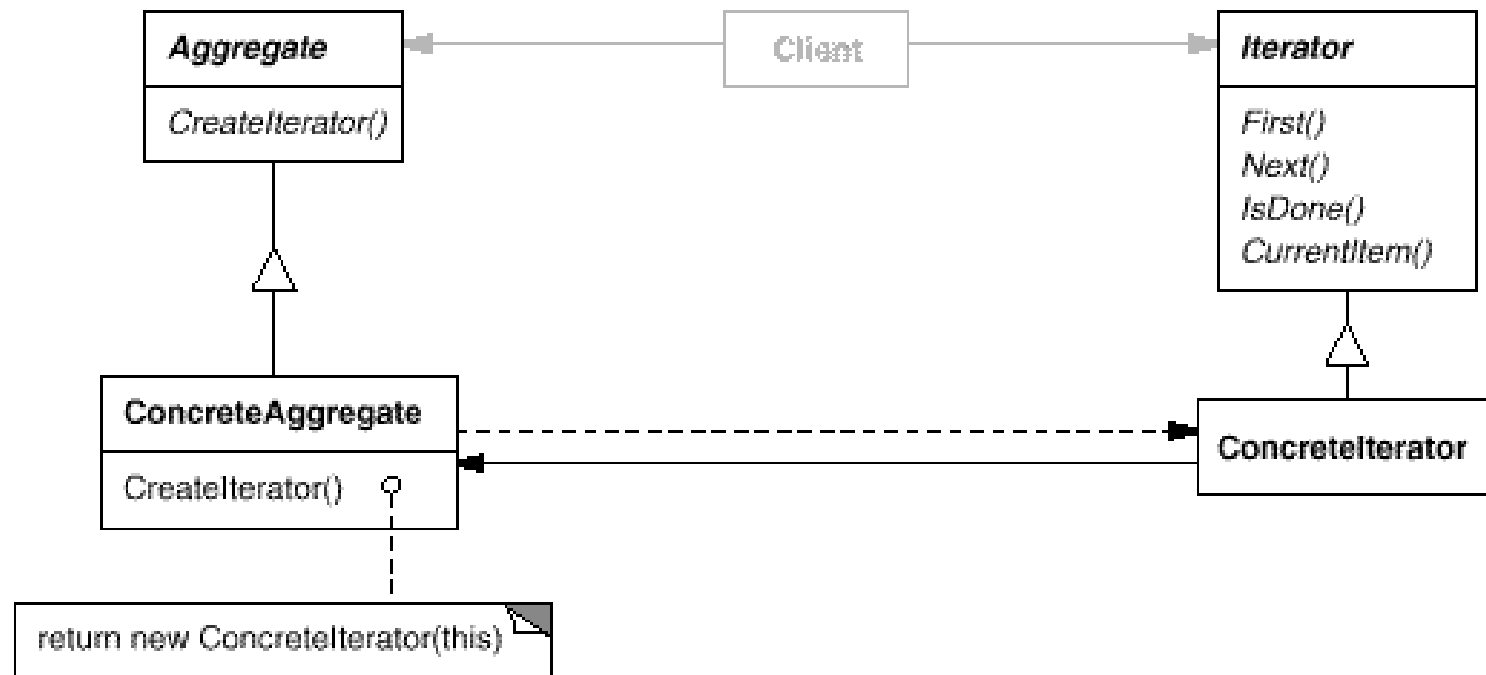
**Iterator**

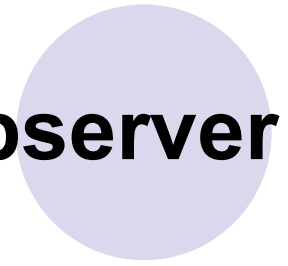
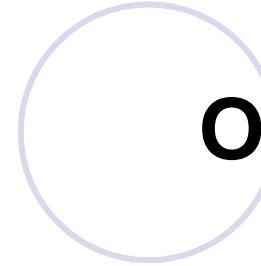
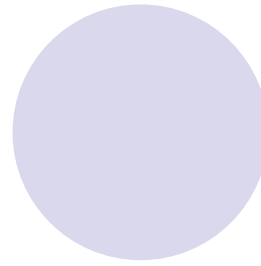
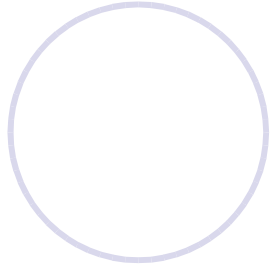
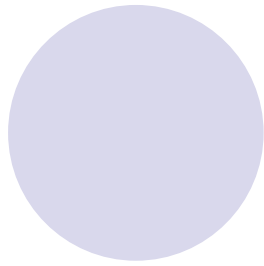
- *Intention* :
  - Fournir un moyen d'accès séquentiel aux éléments d'un agrégat d'objets, sans rendre publique sa représentation interne.
- *Indications* :
  - Pour accéder au contenu d'un objet d'un agrégat sans en révéler la représentation interne.
  - Pour gérer simultanément plusieurs parcours dans un agrégat d'objets.
  - Pour fournir une interface uniforme pour les parcours au travers de diverses structures d'agrégats.

# Iterator *Motivation*



# Iterator Structure





**Observer**

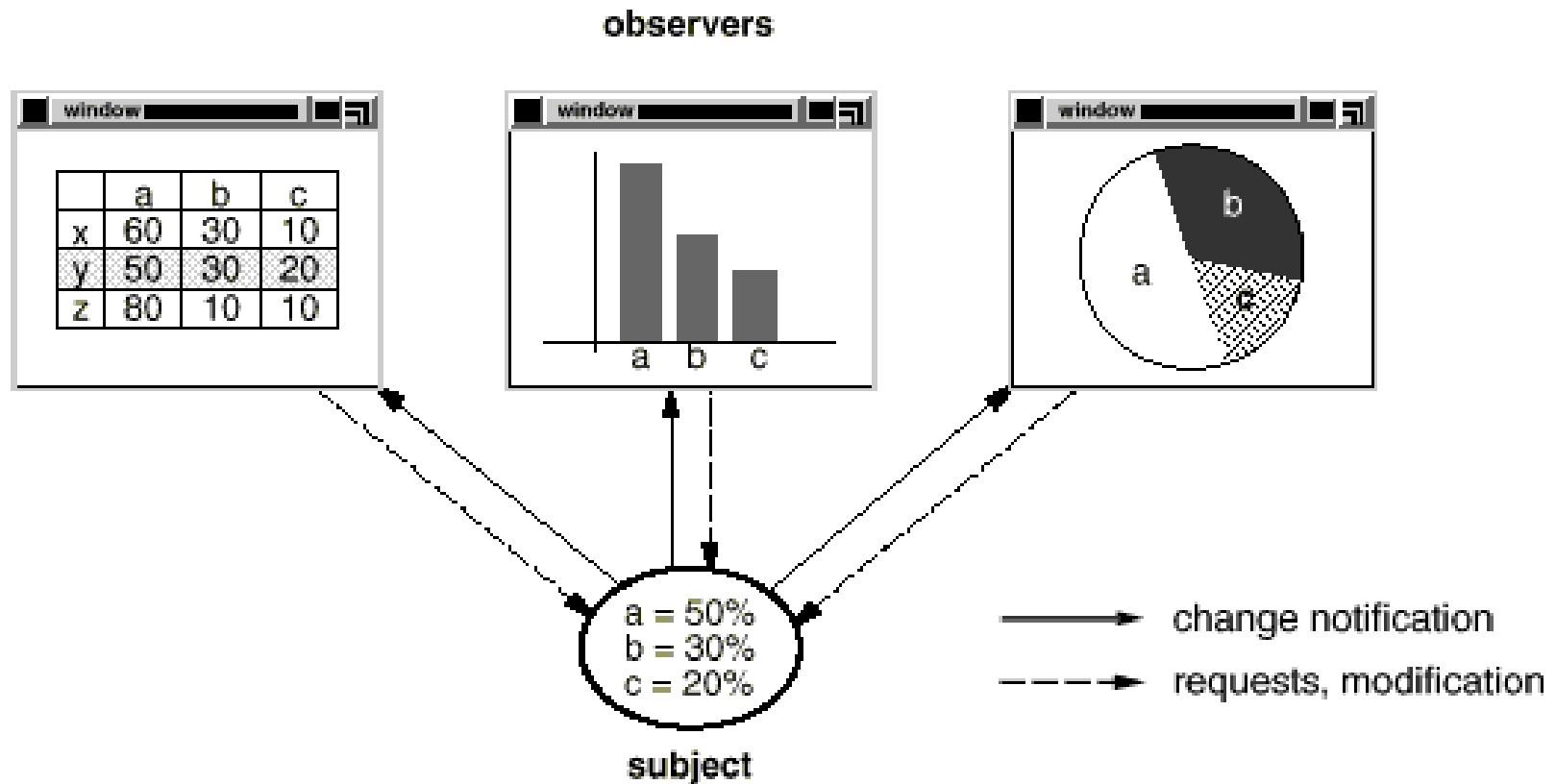
- *Intention* :

- Définir une dépendance de type « un à plusieurs » de façon à ce que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour.

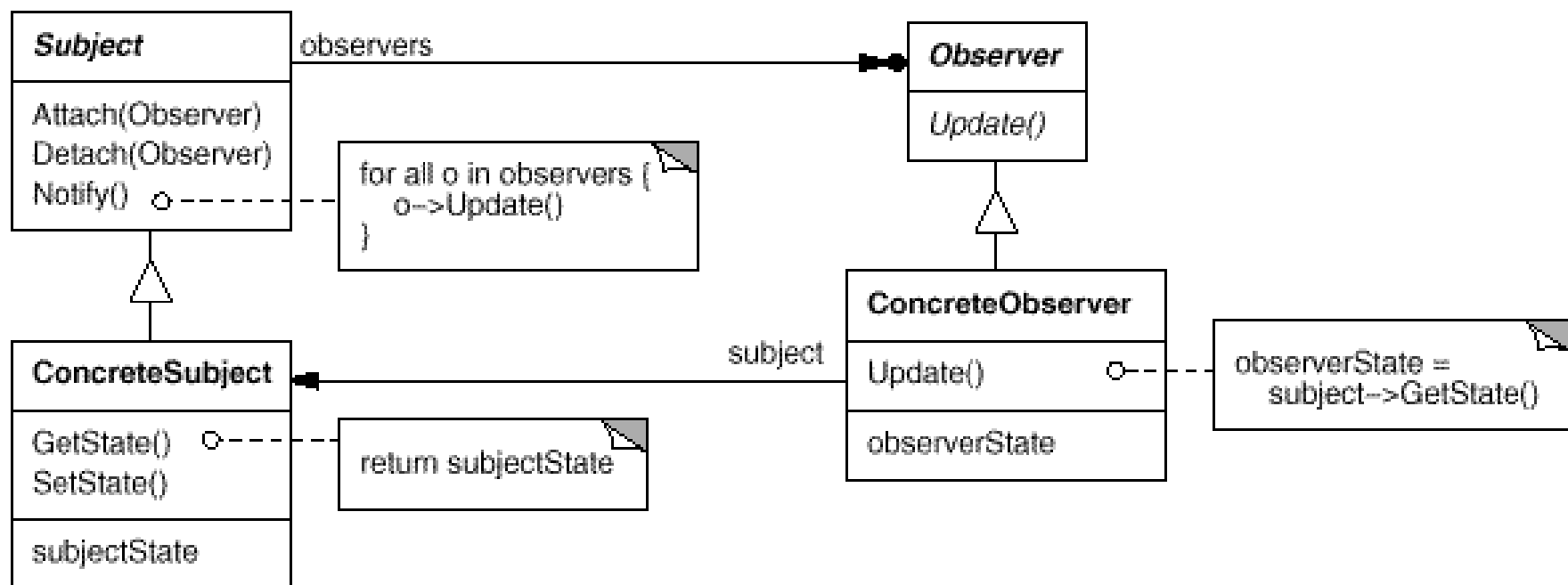
- *Indications* :

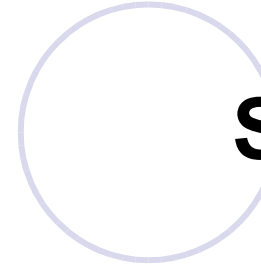
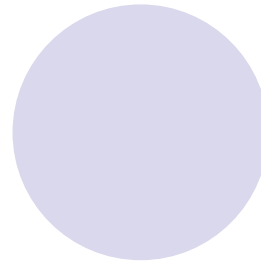
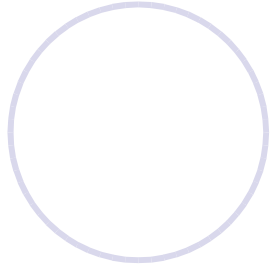
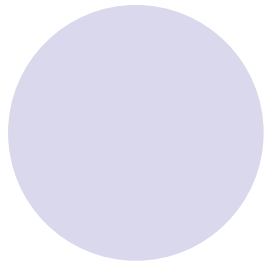
- Quand un concept a plusieurs représentations, l'une dépendant de l'autre.
- Quand la modification d'un objet nécessite la modification d'autres objets, sans qu'on en connaisse le nombre exact.
- Quand un objet doit notifier d'autres objets avec lesquels il n'est pas fortement couplé.

# Observer *Motivation*



# Observer Structure





**Strategy**

- *Intention* :

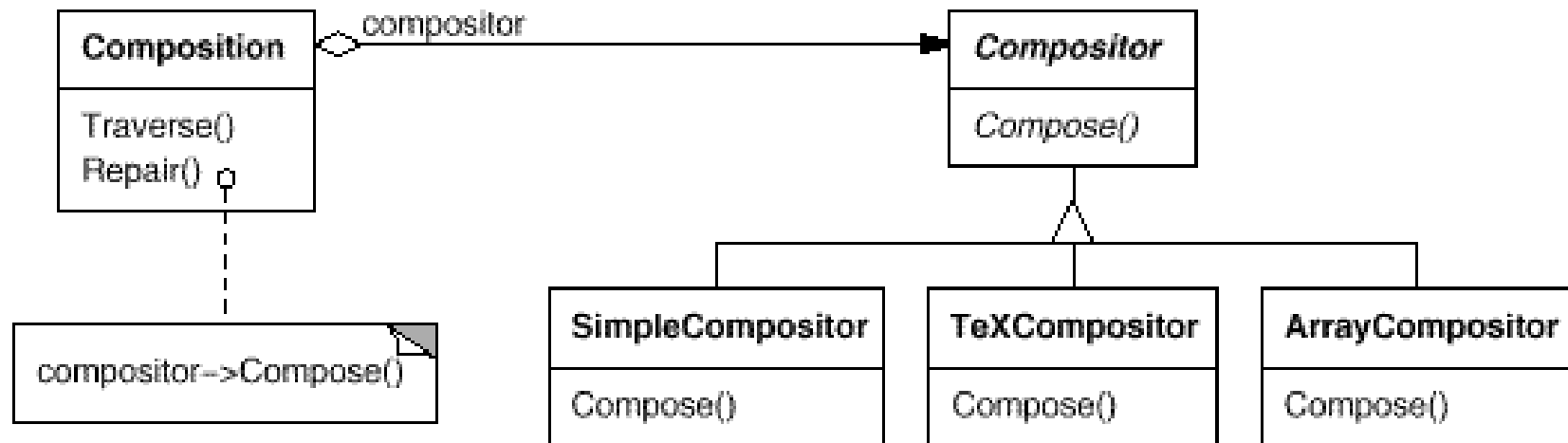
- Définir une famille d'algorithmes en encapsulant chacun d'eux et en les rendant interchangeables.

- *Indications* :

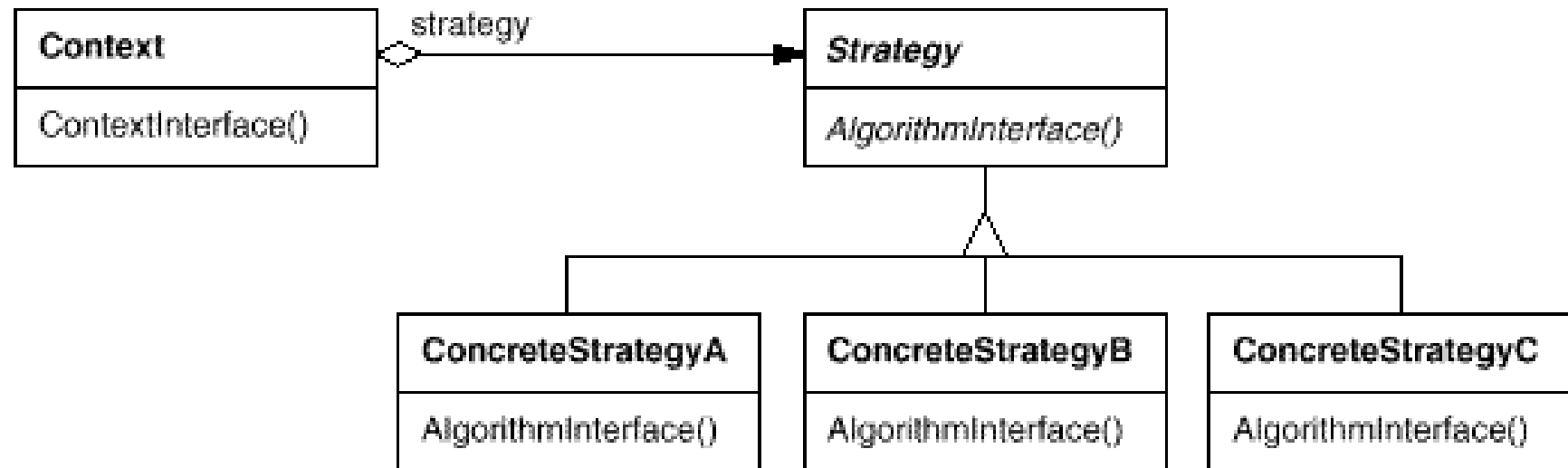
- Plusieurs classes apparentées ne diffèrent que par leur comportement.
- On a besoin de plusieurs variantes d'un algorithme.
- Un algorithme utilise des données que les clients n'ont pas à connaître.
- Une classe définit un ensemble de comportements qui figurent dans des opérations sous la forme de déclarations conditionnelles multiples.



# Strategy *Motivation*



# Strategy Structure

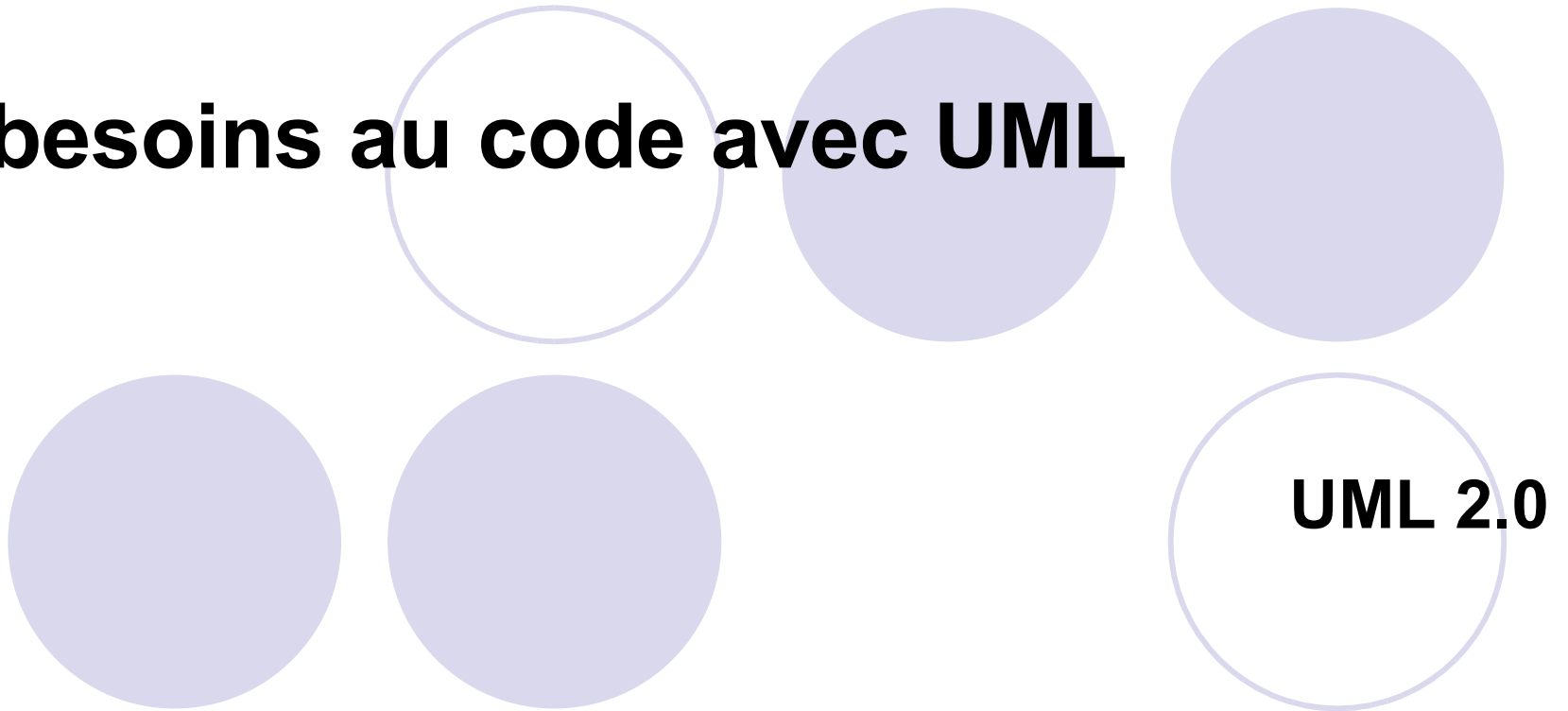


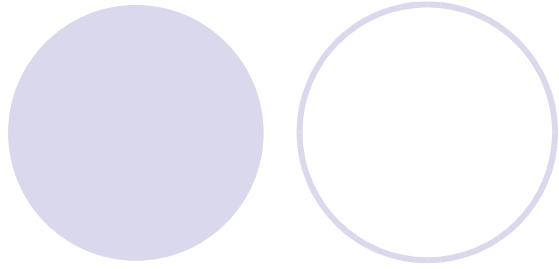


# Utilisation des design patterns

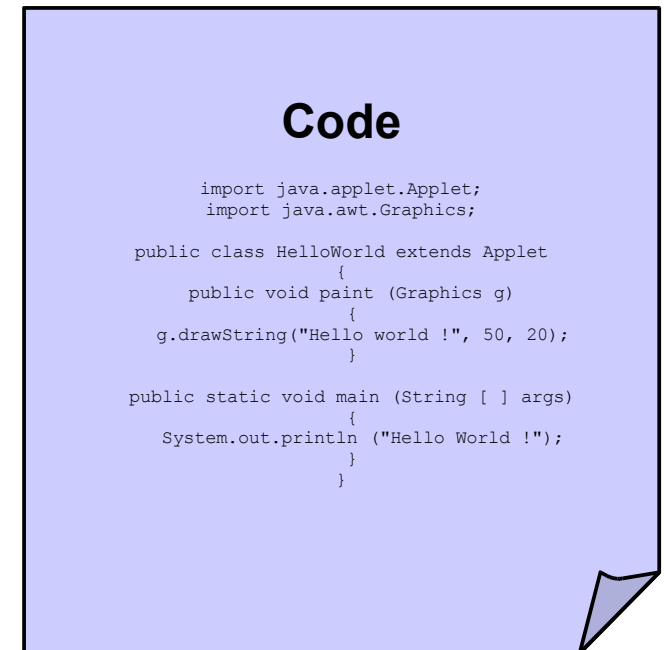
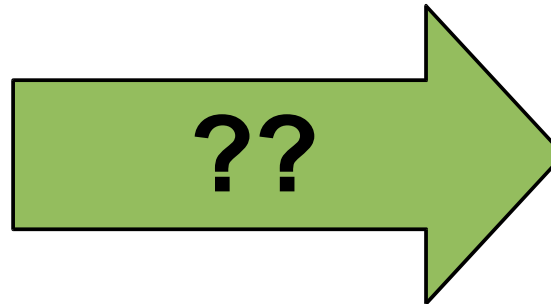
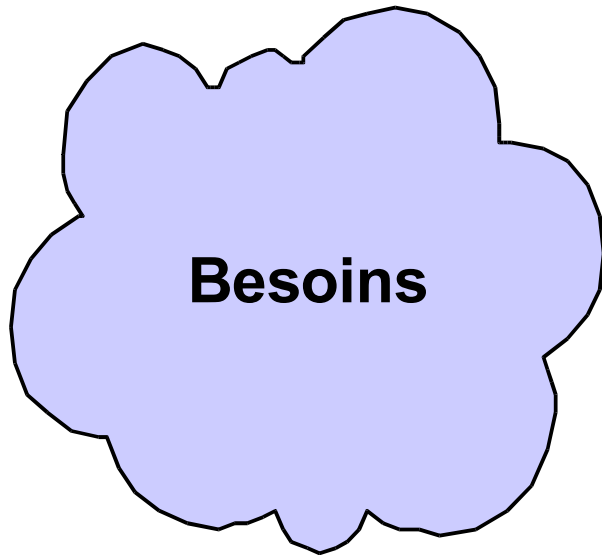
- Les design patterns ne sont que des éléments de conception
  - On les combine pour produire une conception de qualité
    - Trouver les bons objets
    - Mieux réutiliser, concevoir pour l'évolution
- On peut aussi produire de nouveaux design patterns
  - Les DP du GOF ne sont pas les seuls, par exemple :
    - Architecture 3-tiers
    - Modèle Vue Contrôleur
      - Combinaison de Composite, Observer et Strategy
  - On trouve facilement sur Internet des bibliothèques de patterns

# **Des besoins au code avec UML**





# Nécessité d'une méthode





# Processus de développement

- Ensemble d'étapes partiellement ordonnées, qui concourent à l'obtention d'un système logiciel ou à l'évolution d'un système existant.
- Objectif : produire des logiciels
  - De qualité (qui répondent aux besoins de leurs utilisateurs)
  - Dans des temps et des coûts prévisibles
- A chaque étape, on produit
  - Des modèles
  - De la documentation
  - Du code

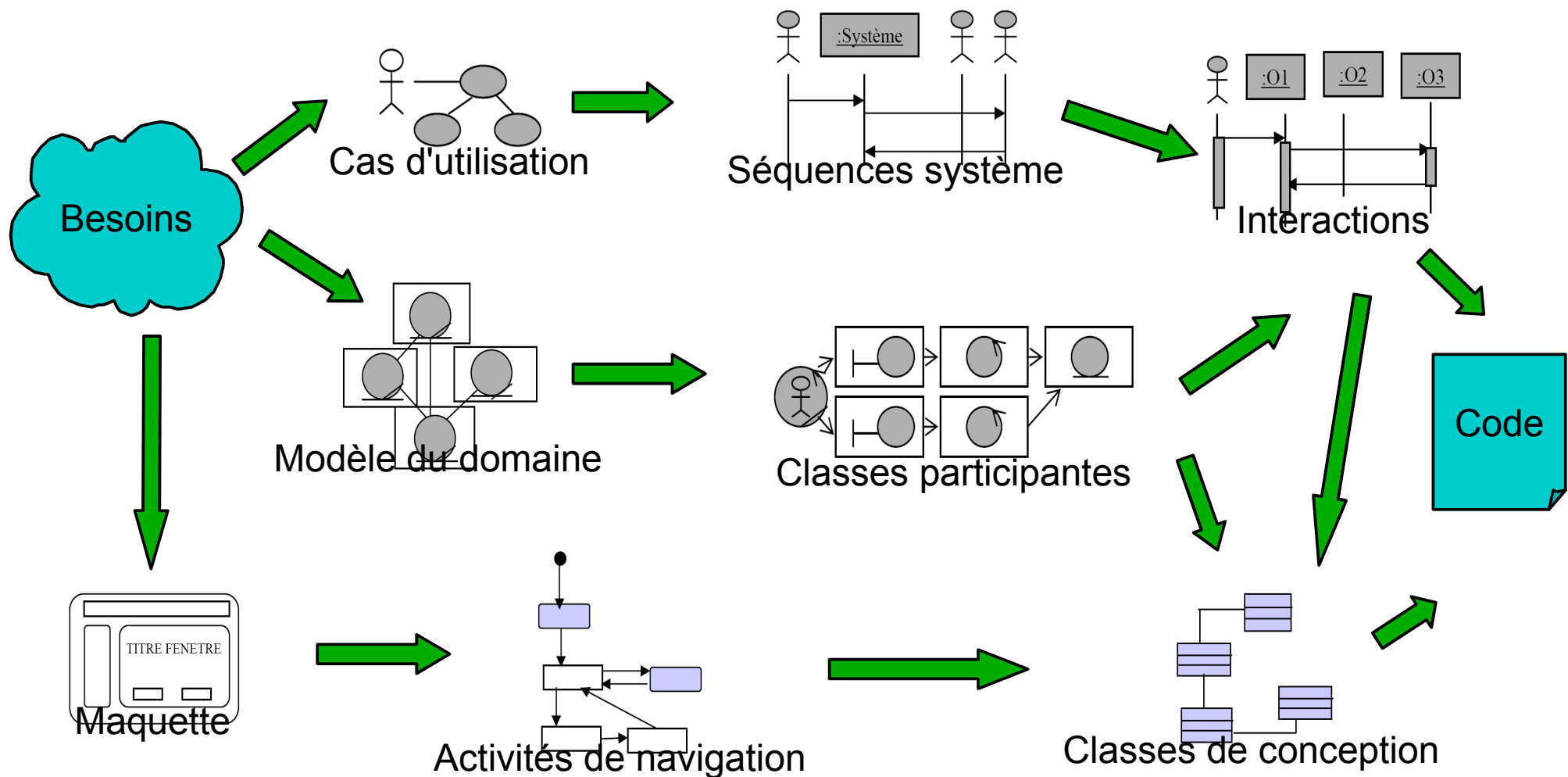


# Méthode = Démarche + Langage

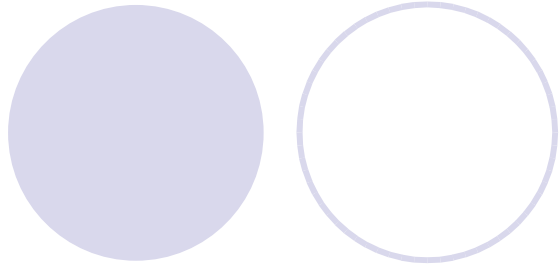
- La méthode MERISE fournit
  - Un langage de représentation graphique (MCD, MPD, MOT, MCT...)
  - Une démarche à adopter pour développer un logiciel
- UML n'est qu'un langage
  - Ne préjuge pas de la démarche employée
- Méthodes s'appuyant sur UML
  - RUP (Rational Unified Process)
  - XP (eXtreme Programming)

# Méthode minimale

- Proposition d'une méthode à mi-chemin entre RUP et XP
- Résoudre 80% des problèmes avec 20% d'UML







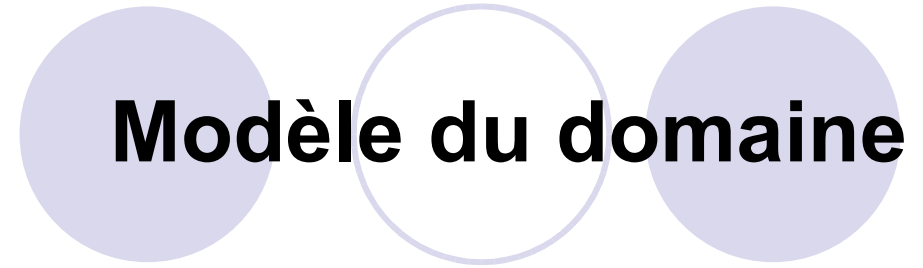
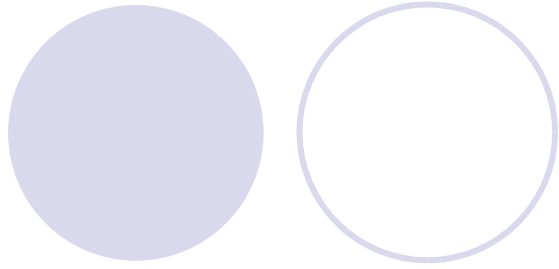
## Cas d'utilisation

- Démarche pour aboutir au diagramme de cas d'utilisation :
  - Identifier les acteurs
  - Identifier les cas d'utilisation
  - Structurer les cas d'utilisation en packages
  - Ajouter les relations entre cas d'utilisation
  - Classer les cas d'utilisation



# Exemple de classement

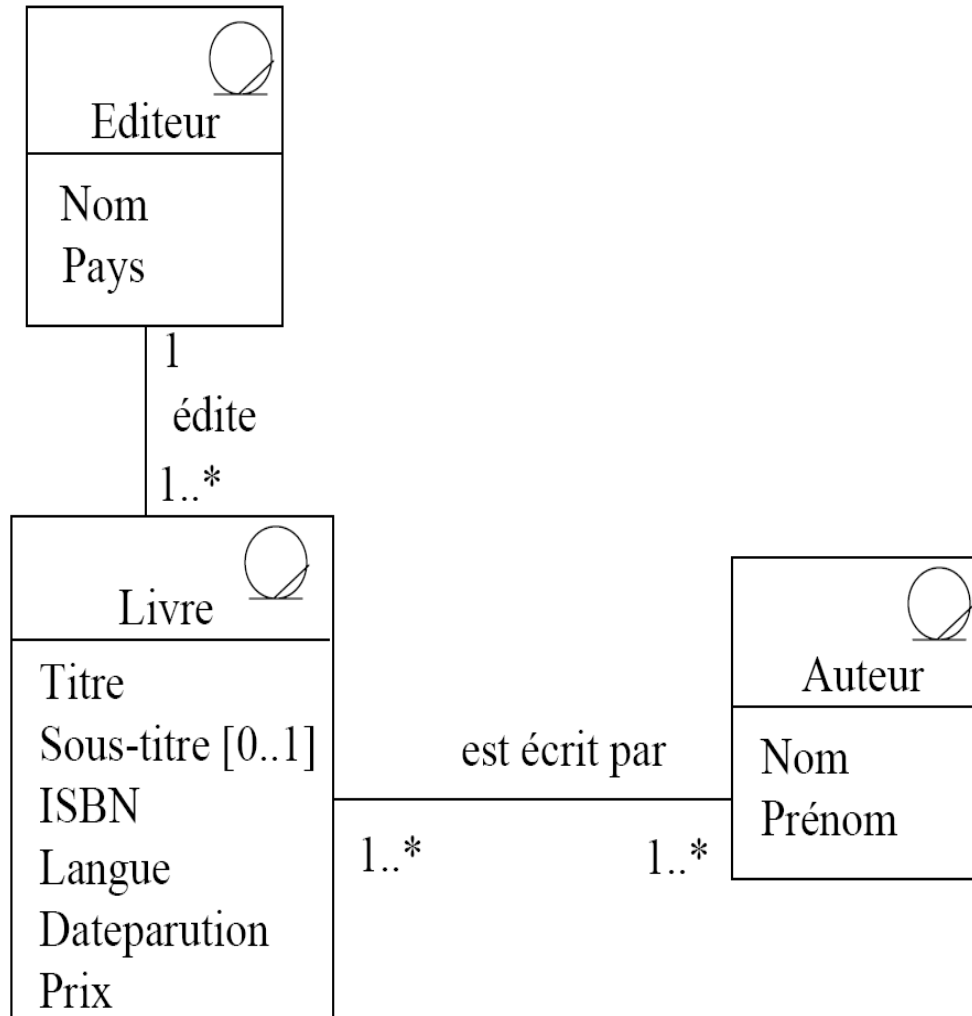
| <b><i>Cas d'utilisation</i></b>        | <b><i>Priorité</i></b> | <b><i>Risque</i></b> |
|----------------------------------------|------------------------|----------------------|
| Rechercher des ouvrages                | Haute                  | Moyenne              |
| Gérer son panier                       | Haute                  | Bas                  |
| Effectuer une commande                 | Moyenne                | Haut                 |
| Consulter ses commandes en cours       | Basse                  | Moyen                |
| Consulter l'aide en ligne              | Basse                  | Bas                  |
| Maintenir le catalogue                 | Haute                  | Haut                 |
| Maintenir les informations éditoriales | Moyenne                | Bas                  |



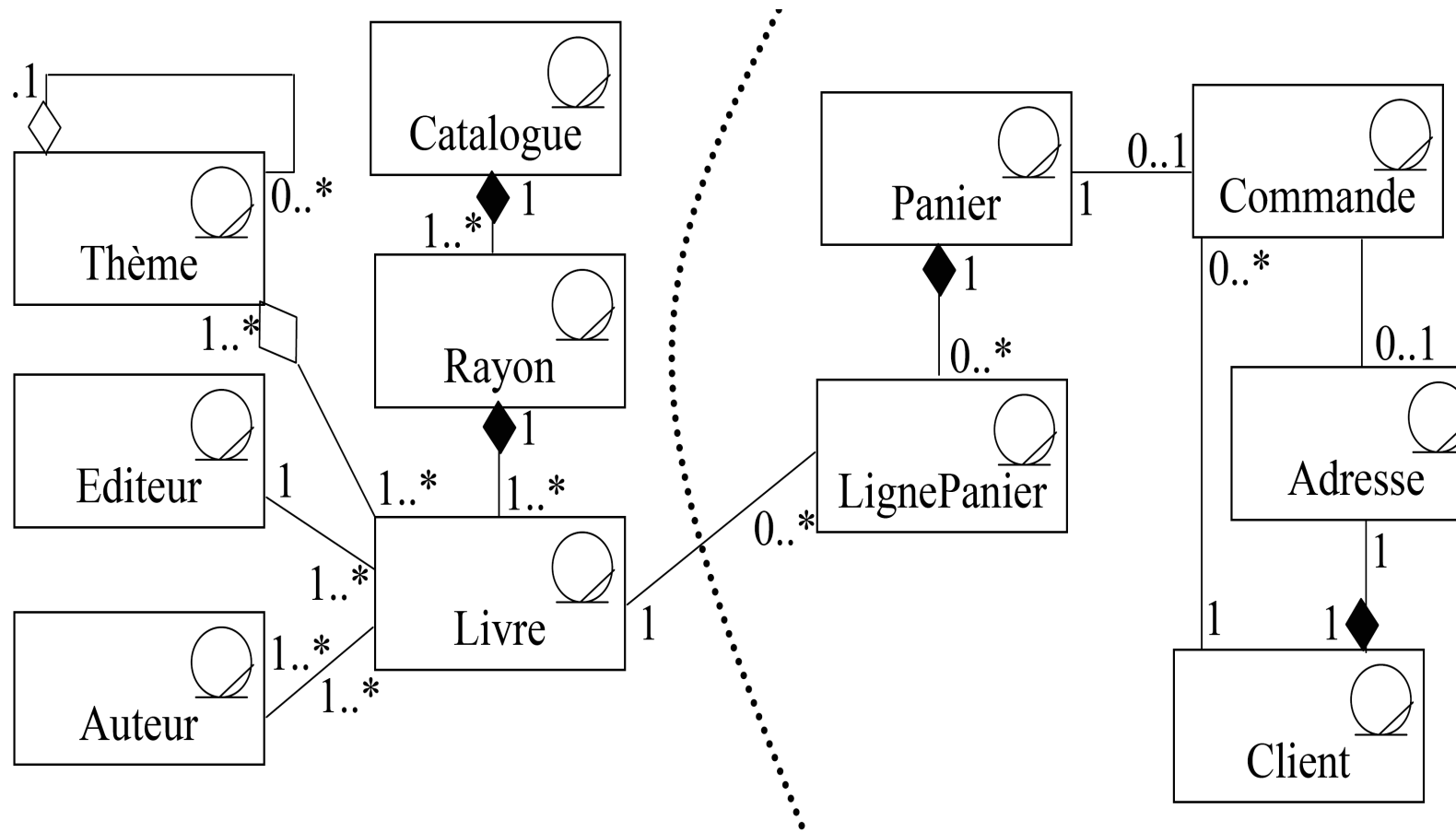
## Modèle du domaine

- Le modèle du domaine est constitué d'un ensemble de classes dans lesquelles aucune opération n'est définie
- Etapes de la démarche :
  - Identifier les concepts du domaine
  - Ajouter les associations et les attributs
  - Généraliser les concepts
  - Structurer en packages : structuration selon les principes de cohérence et d'indépendance.
- Les concepts du domaine peuvent être identifiés directement à partir de la connaissance du domaine ou par interview des experts métier.

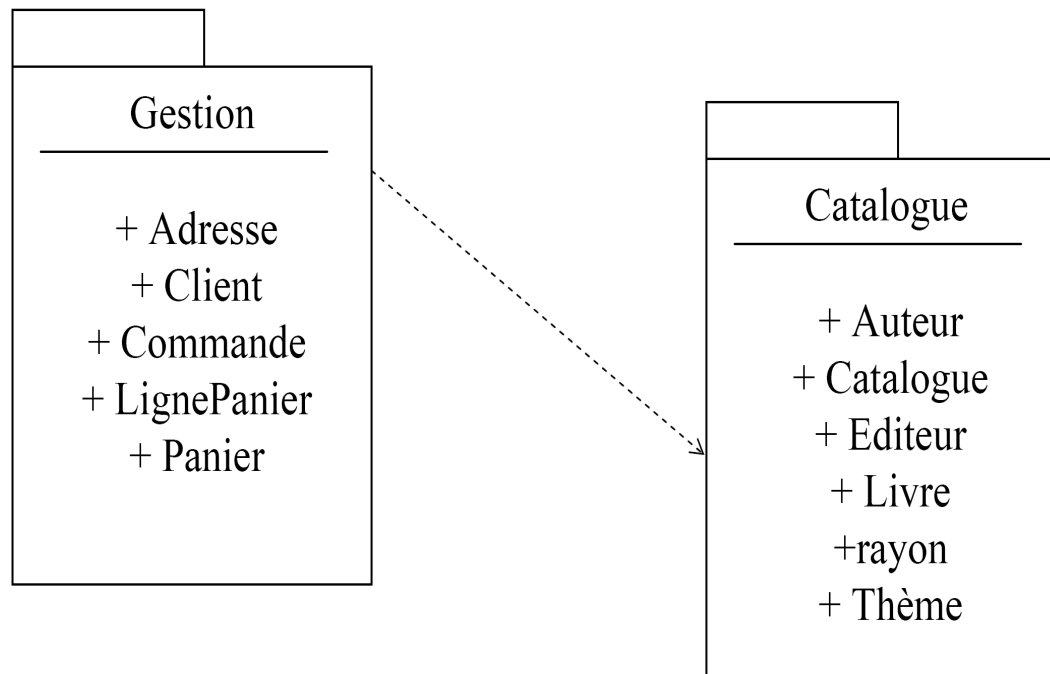
# Exemple de modèle du domaine



# Structuration en packages



# Définition synthétique des packages

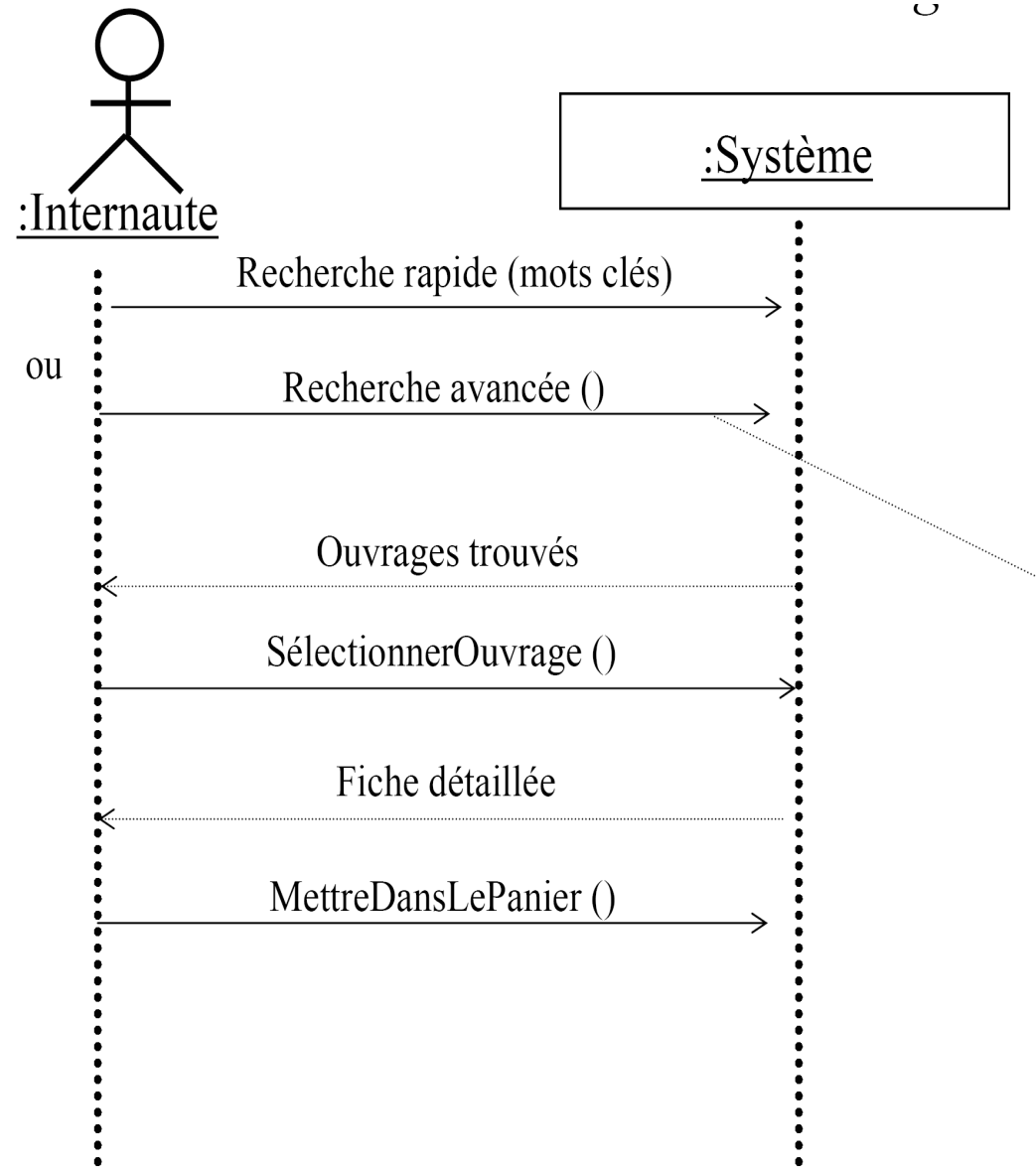




# Diagramme de séquence système

- Formalisation des descriptions textuelles des cas d'utilisation
  - Mise à jour des cas d'utilisation
  - Construction des diagrammes de séquence système
  - Spécification des opérations système
- Le système est ici considéré comme un tout
  - On s'intéresse à ses interactions avec les acteurs

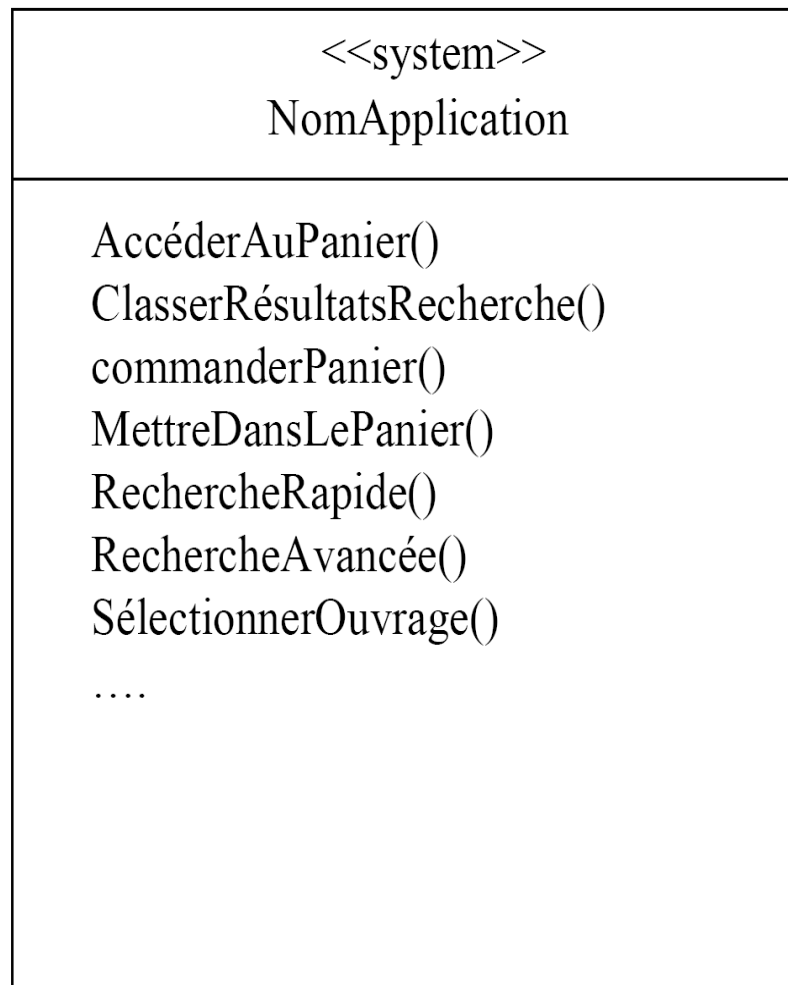
# Exemple de diagramme de séquence système

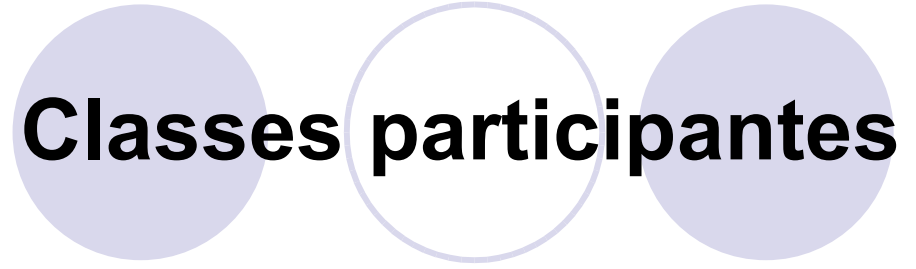
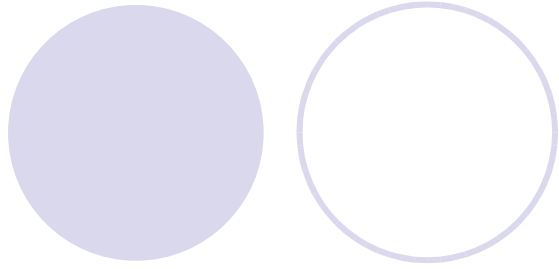






# Opérations système





## Classes participantes

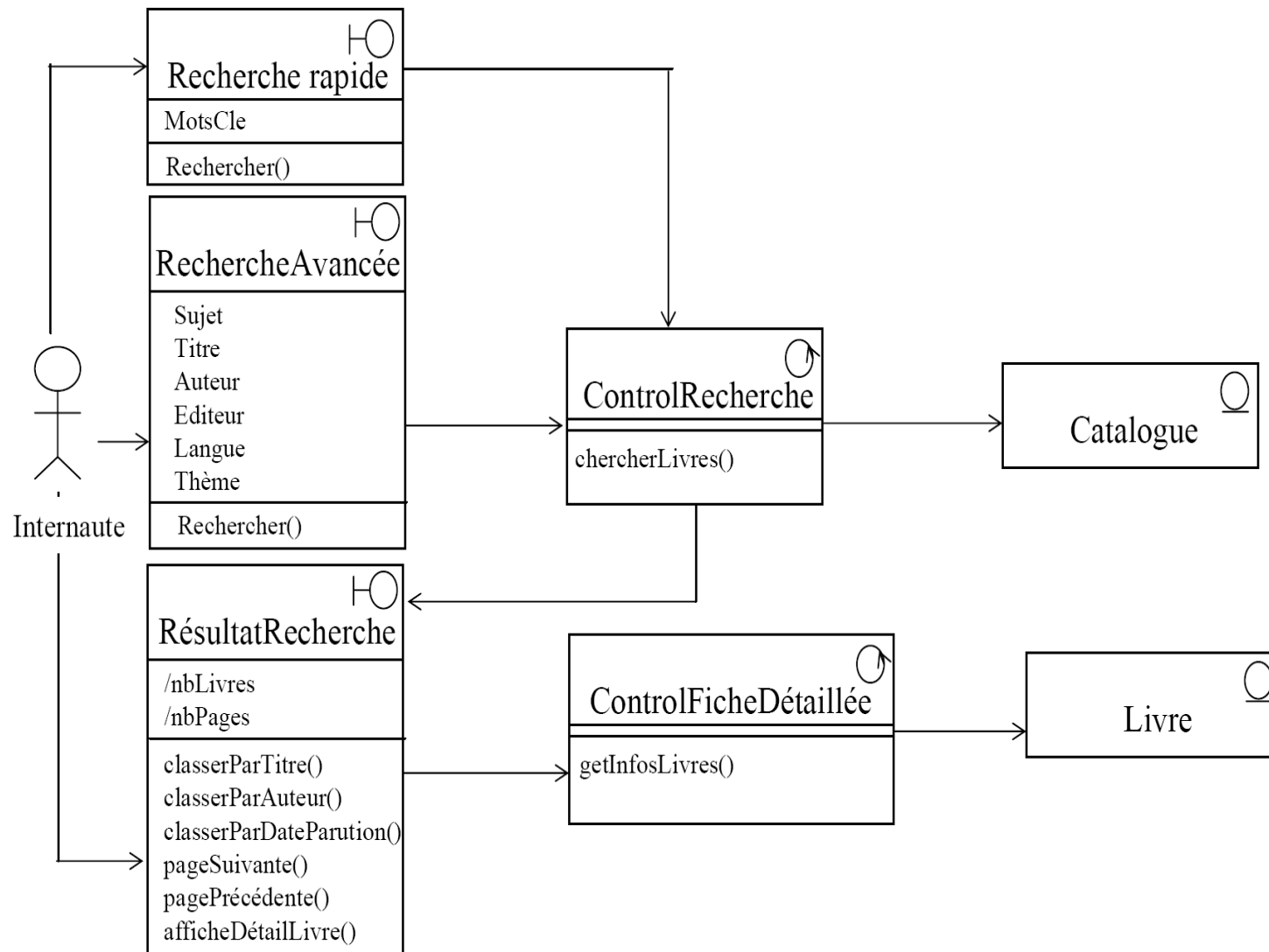
- Réalisation des cas d'utilisation par les *classes d'analyse*
- Typologie des classes d'analyse
  - Les *classes dialogue* permettent les interactions entre les utilisateurs et l'application.
  - Les *classes contrôle* contiennent la cinématique de l'application et font la transition entre les classes dialogues et les classes métiers.
  - Les *classes métier* ou entités représentent les objets métier. Elles proviennent directement du modèle du domaine (mais peuvent être complétées en fonction des cas d'utilisation).



# Diagramme de classes participantes

- Diagramme de classes UML décrivant les principales classes d'analyse dans lequel on ajoute les acteurs
- Seules classes dialogue ont des *opérations* (actions de l'utilisateur sur l'IHM)
- *Associations* :
  - Les dialogues ne peuvent être reliés qu'aux contrôles ou à d'autres dialogues (en général, associations unidirectionnelles)
  - Les classes métier ne peuvent être reliées qu'aux contrôles ou à d'autres classes métier.
  - Les contrôles ont accès à tous les types de classes.

# Exemples de classes participantes

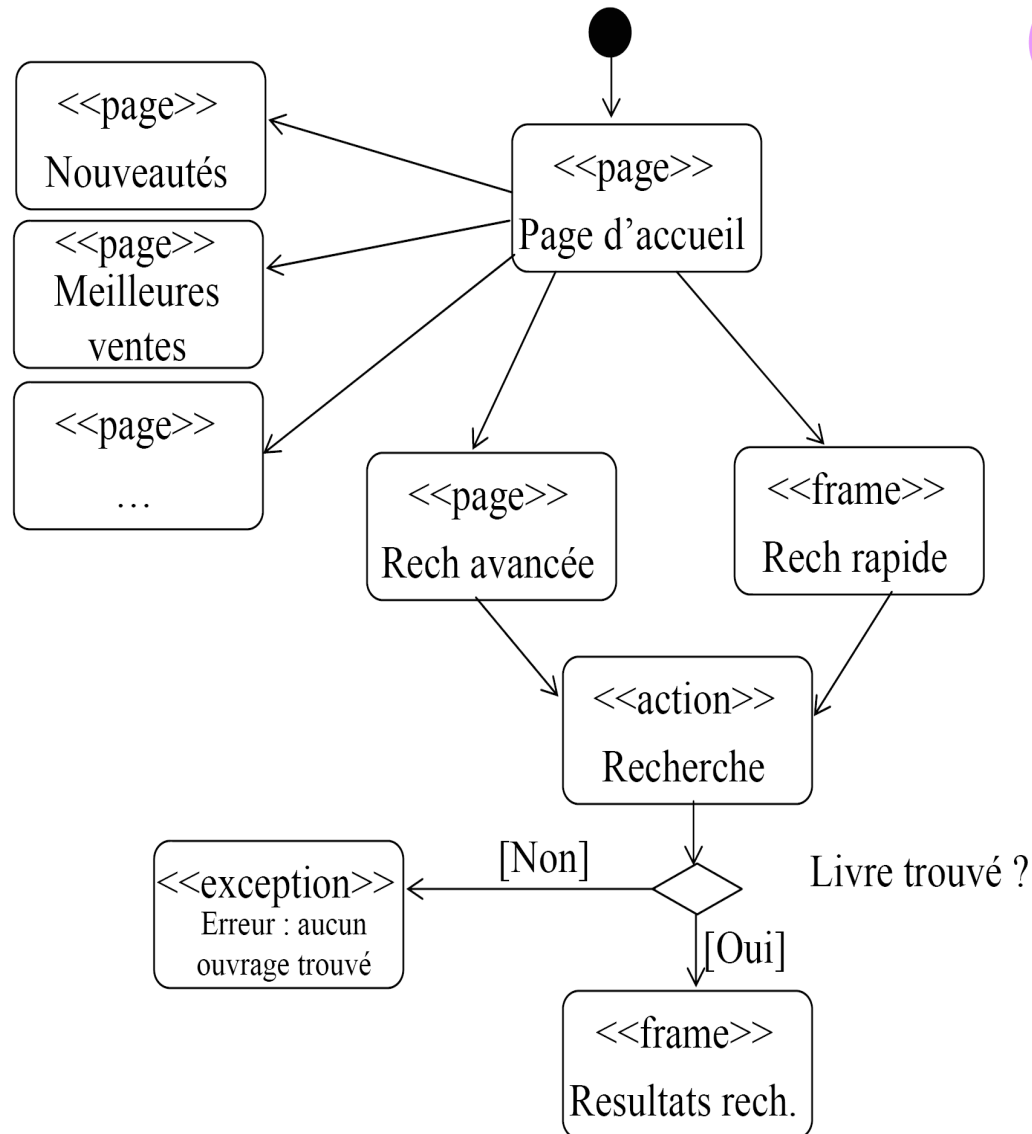


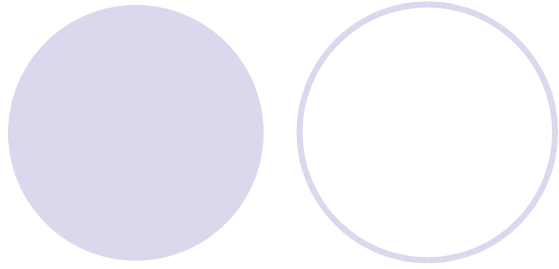


# Diagramme d'activités de navigation

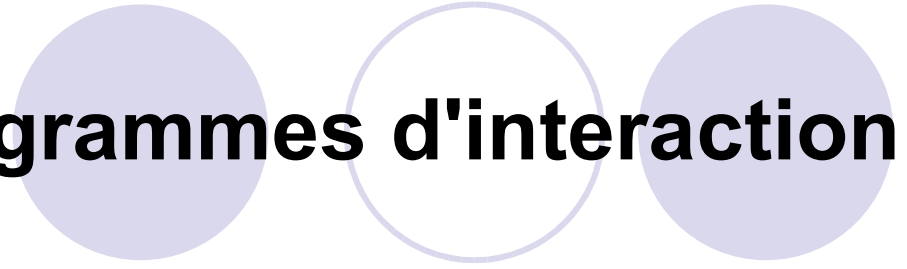
- Modélisation de l'IHM avec des diagrammes d'activité
- Exploitation des maquettes de manière à représenter l'ensemble des chemins possibles entre les principaux écrans proposés à l'utilisateur.

# Exemple de diagramme d'activités



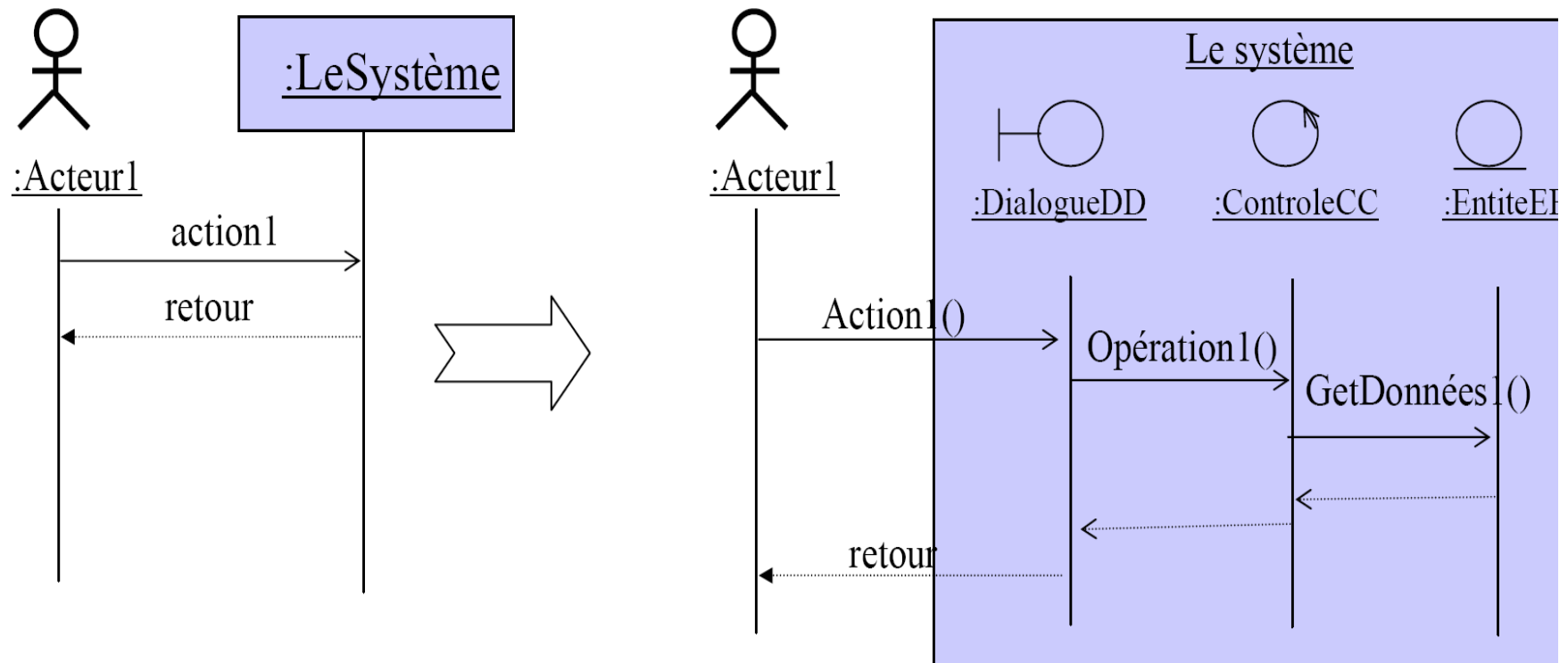


# Diagrammes d'interaction



- Diagrammes de séquence système
  - Système vu comme une boîte noire
- Diagrammes d'interaction pour la conception préliminaire
  - Le système est un ensemble d'objets en collaboration
- Utilisation d'objets des *classes participantes*

# De l'analyse à la conception préliminaire







# Diagramme des classes de conception

- Enrichissement du diagramme de classes pour
  - Prendre en compte l'architecture logicielle hôte
  - Modéliser les opération privées des différentes classes
  - Finaliser le modèle des classes avant l'implémentation
- Illustration des diagrammes de classes par des diagrammes de séquence détaillés.
  - Interactions entre classes
  - Spécification d'algorithmes

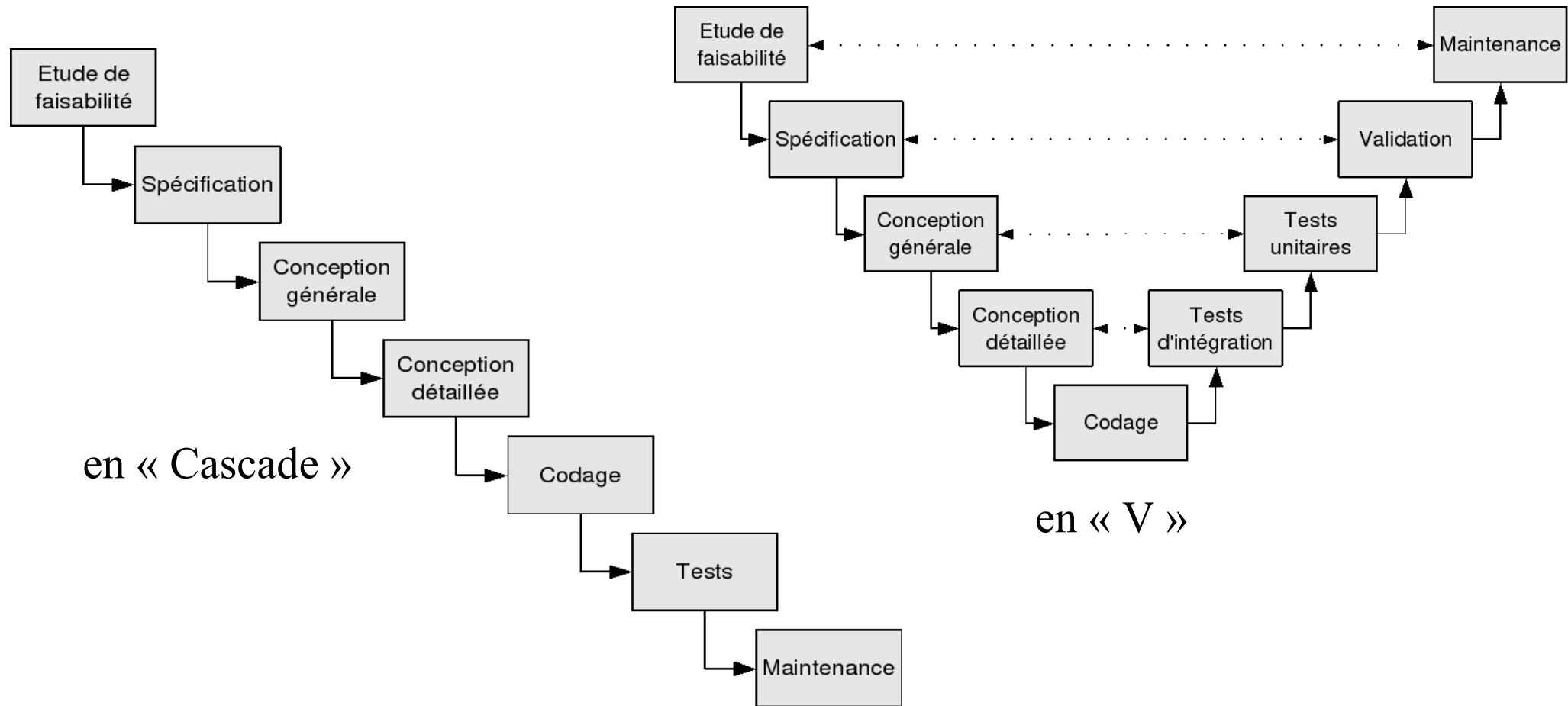
# **Rational Unified Process**



The diagram consists of six circles arranged in two rows of three. The top row has a hollow circle on the left, a solid light purple circle in the middle, and a solid light purple circle on the right. The bottom row has a solid light purple circle on the left, a solid light purple circle in the middle, and a hollow circle on the right. The text 'Rational Unified Process' is centered over the top row, and 'UML 2.0' is centered over the bottom right circle.

**UML 2.0**

# Modèles de cycles de vie linéaire

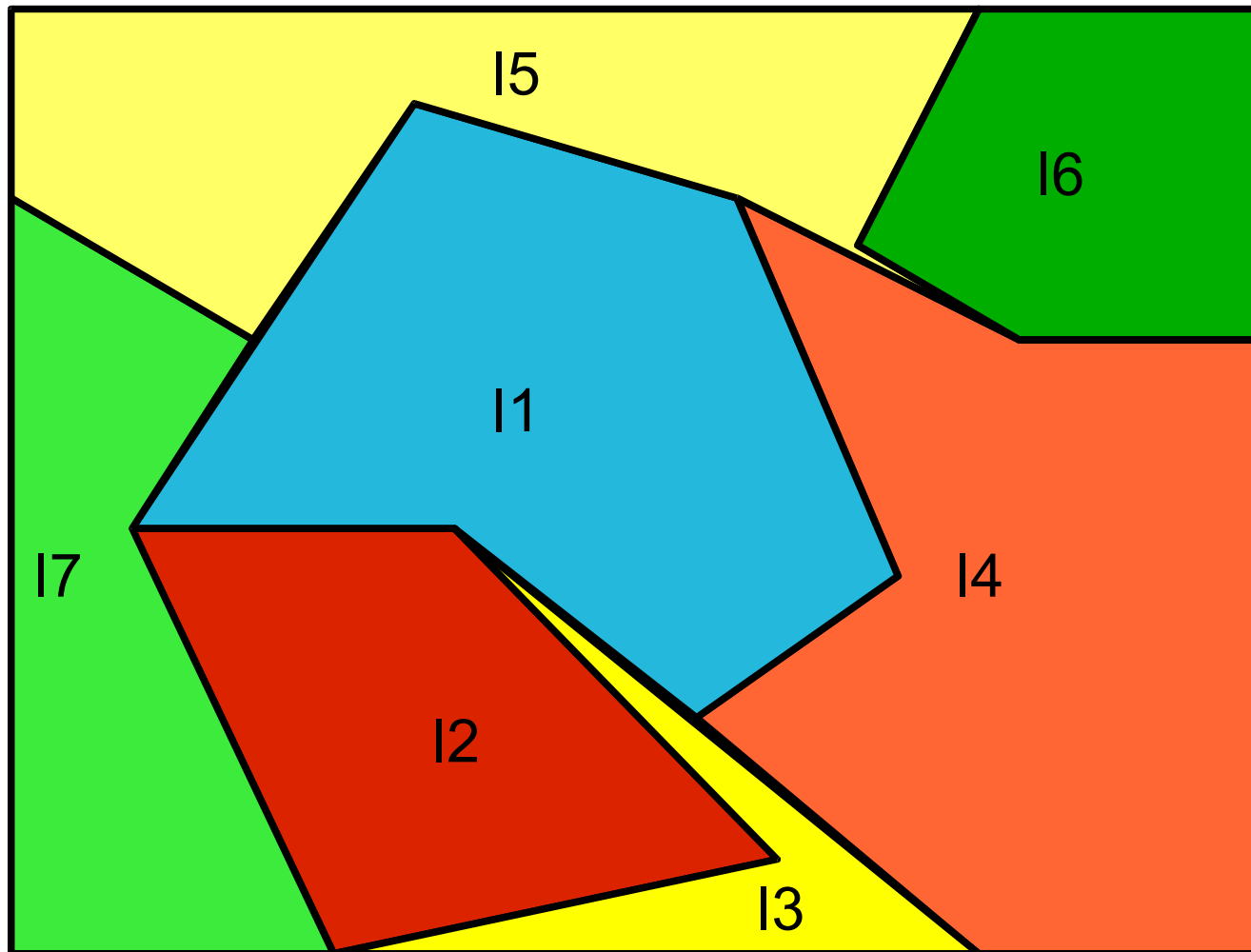




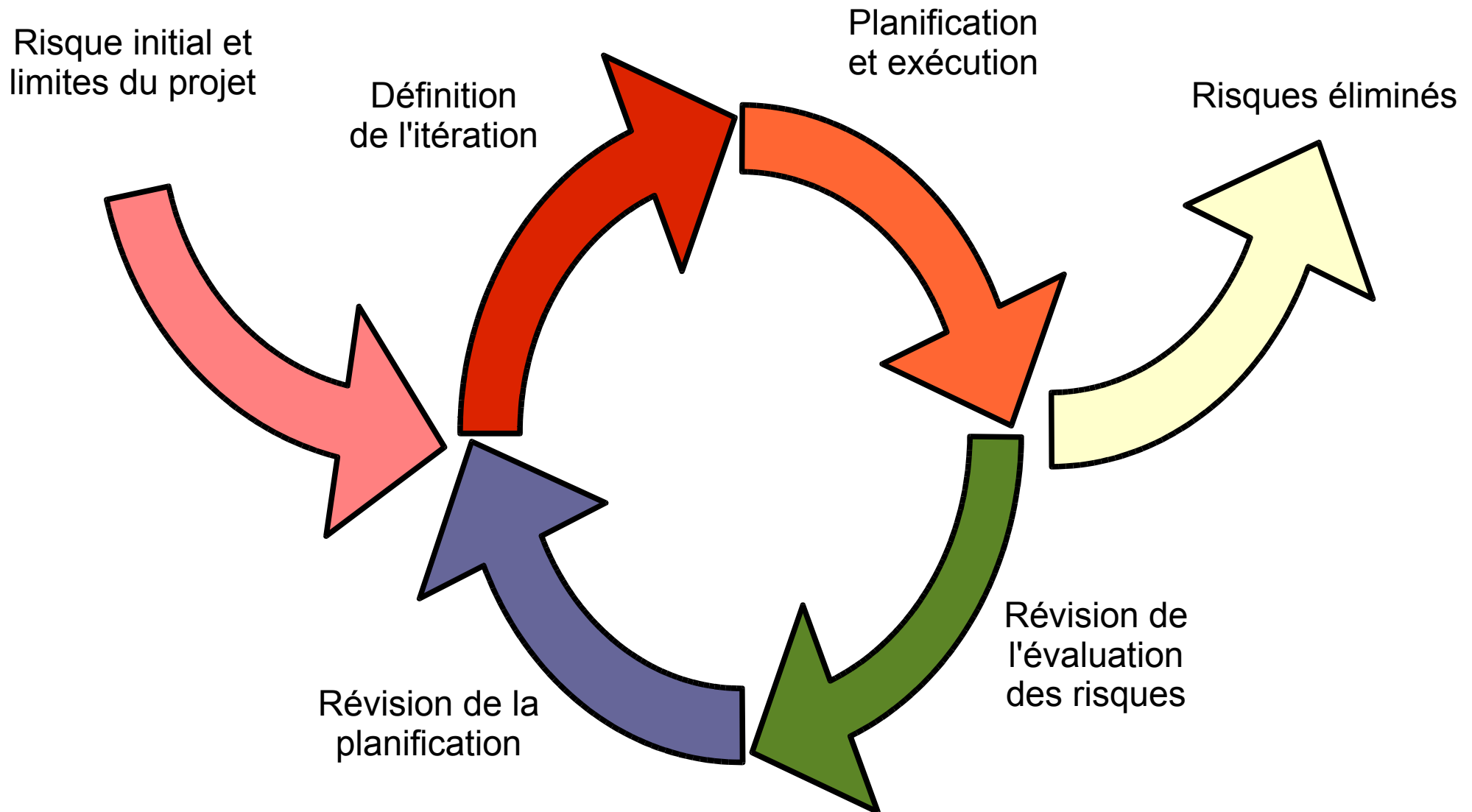
# Problèmes des cycles linéaires

- Risques élevés et non contrôlés
  - Identification *tardive* des problèmes
  - Preuve *tardive* de bon fonctionnement
- Améliorations : construction *itérative* du système
  - Chaque itération produit un nouvel *incrément*
  - Chaque nouvel incrément a pour but de maîtriser une partie des risques et apporte une preuve tangible de faisabilité ou d'adéquation
    - Enrichissement d'une série de prototypes ; les versions livrées correspondent à une étape de la chaîne des prototypes

# Production it rative d'incr ments



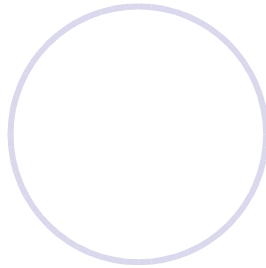
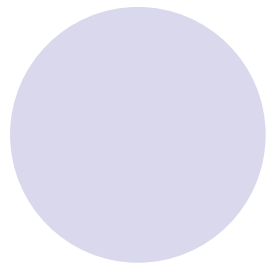
# Elimination des risques à chaque itération



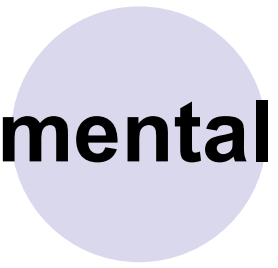
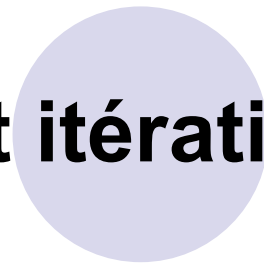


# Rational Unified Process

- RUP est une démarche de développement qui est souvent utilisé conjointement au langage UML
- RUP est
  - Piloté par les cas d'utilisation
  - Centré sur l'architecture
  - Itératif et incrémental



# **RUP est itératif et incrémental**



- Chaque itération prend en compte un certain nombre de cas d'utilisation
- Les risques majeurs sont traités en priorité
- Chaque itération donne lieu à un incrément et produit une nouvelle version exécutable



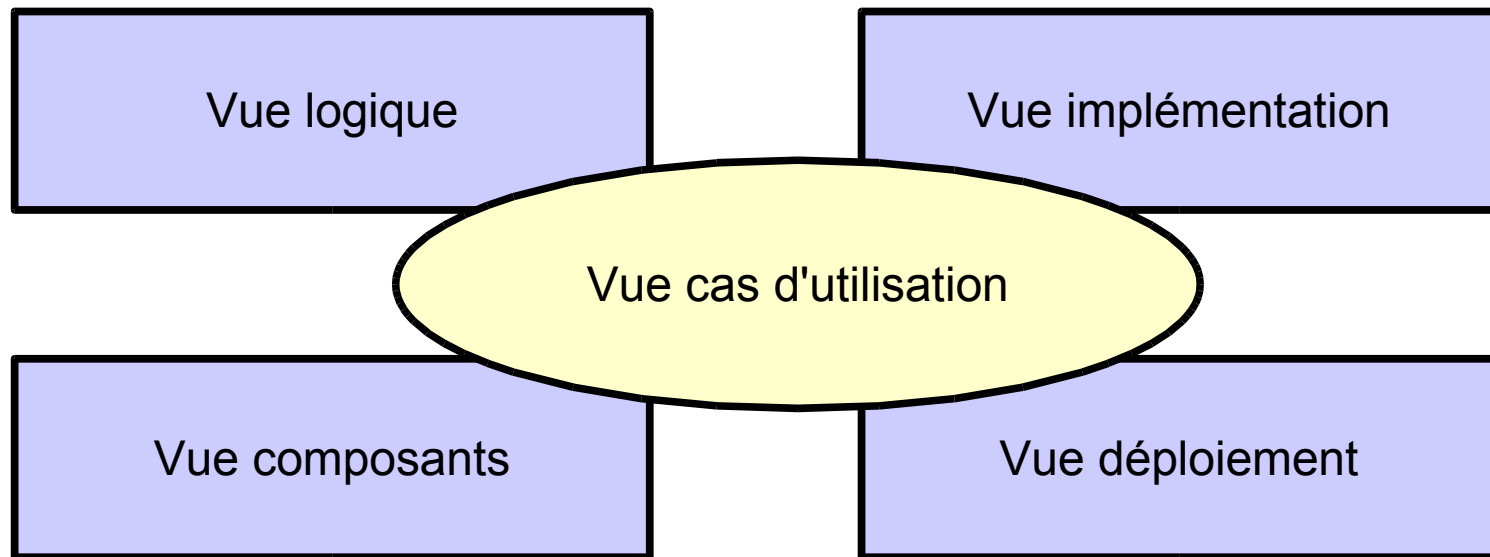


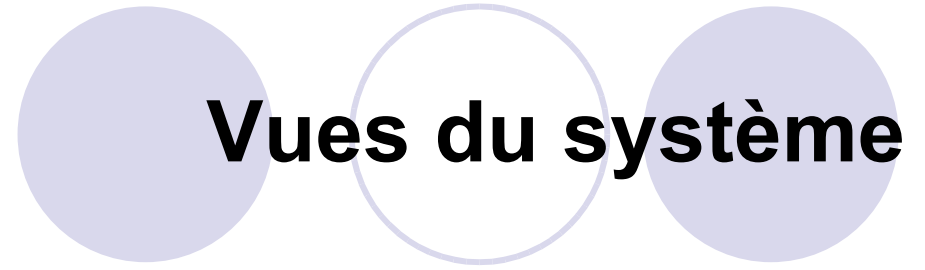
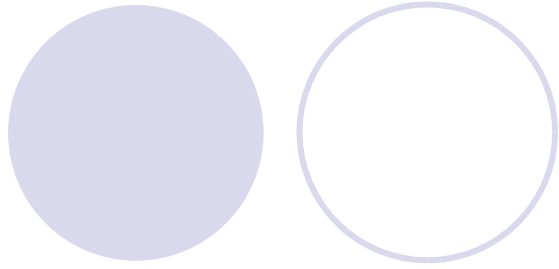
# RUP est piloté par les cas d'utilisation

- La principale qualité d'un logiciel est son *utilité*
  - Adéquation avec les besoins des utilisateurs
- Le développement d'un logiciel doit être centré sur l'utilisateur
- Les cas d'utilisation permettent d'exprimer ces besoins
  - Détection et description des besoins fonctionnels
  - Organisation des besoins fonctionnels

# RUP est centré sur l'architecture

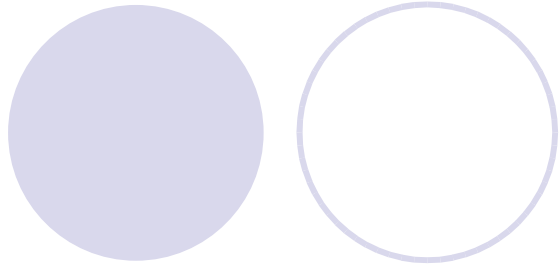
- Modélisation de différentes perspectives indépendantes et complémentaires
- Architecture en couches et vues de Krutchen





## Vues du système

- Vue cas d'utilisation
  - Description du système comme un ensemble de transactions du point de vue de l'utilisateur
- Vue logique
  - Créée lors de la phase d'élaboration et raffinée lors de la phase de construction



## Vues du système

- Vue composants
  - Description de l'architecture logicielle
- Vue déploiement
  - Description de l'architecture matérielle du système
- Vue implémentation
  - Description des algorithmes, code source

# Organisation en phases du développement

- Initialisation
  - Définition du problème
- Elaboration
  - Planification des activités, affectation des ressources, analyse
- Construction
  - Développement du logiciel par incréments successifs
- Transition
  - Recettage et déploiement



# Objectifs de la phase d'initialisation

- Définition du cadre du projet, son concept, et inventaire du contenu
- Elaboration des cas d'utilisation critiques ayant le plus d'influence sur l'architecture et la conception
- Réalisation d'un ou de plusieurs prototypes démontrant les fonctionnalités décrites par les cas d'utilisation principaux
- Estimation détaillée de la charge de travail, du coût et du planning général ainsi que de la phase suivante d'élaboration  
Estimation des risques



# Activités de la phase d'initialisation

- Formulation du cadre du projet, des besoins, des contraintes et des critères d'acceptation
- Planification et préparation de la justification économique du projet et évaluation des alternatives en termes de gestion des risques, ressources, planification
- Synthèse des architectures candidates, évaluation des coûts



# **Livrables de la phase d'initialisation**

- Un document de vision présentant les besoins de base, les contraintes et fonctionnalités principales
- Une première version du modèle de cas d'utilisation
- Un glossaire de projet
- Un document de justification économique incluant le contexte général de réalisation, les facteurs de succès et la prévision financière
- Une évaluation des risques
- Un plan de projet présentant phases et itérations
- Un ou plusieurs prototypes





# Critères d'évaluation de la phase d'initialisation

- Un consensus sur
  - la planification,
  - les coûts
  - la définition de l'ensemble des projets des parties concernées
- La compréhension commune des besoins



# Objectifs de la phase d'élaboration

- Définir, valider et arrêter l'architecture
- Démontrer l'efficacité de cette architecture à répondre à notre besoin
- Planifier la phase de construction



# Activités de la phase d'élaboration

- Elaboration de la vision générale du système, les cas d'utilisation principaux sont compris et validés
- Le processus de projet, l'infrastructure, les outils et l'environnement de développement sont établis et mis en place
- Elaboration de l'architecture et sélection des composants



# **Livrables de la phase d'élaboration**

- Le modèle de cas d'utilisation est produit au moins à 80 %
- La liste des exigences et contraintes non fonctionnelles identifiées
- Une description de l'architecture
- Un exécutable permettant de valider l'architecture du logiciel à travers certaines fonctionnalités complexes
- La liste des risques revue et la mise à jour de la justification économique du projet
- Le plan de réalisation, y compris un plan de développement présentant les phases, les itérations et les critères d'évaluation

# Critères d'évaluation de la phase d'élaboration

- La stabilité de la vision du produit final
- La stabilité de l'architecture
- La prise en charge des risques principaux est adressée par le(s) prototype(s)
- La définition et le détail du plan de projet pour la phase de construction
- Un consensus, par toutes les parties prenantes, sur la réactualisation de la planification, des coûts et de la définition de projet



# Objectifs de la phase de construction

- La minimisation des coûts de développement par
  - l'optimisation des ressources
  - la minimisation des travaux non nécessaires
- Le maintien de la qualité
- Réalisation des versions exécutables



# Activités de la phase de construction

- La gestion et le contrôle des ressources et l'optimisation du processus de projet
- Evaluation des versions produites en regard des critères d'acceptation définis



# **Livrables de la phase de construction**

- Les versions exécutables du logiciel correspondant à l'enrichissement itération par itération des fonctionnalités
- Les manuels d'utilisation réalisés en parallèle à la livraison incrémentale des exécutables
- Une description des versions produites





# Critères d'évaluation de la phase de construction

- La stabilité et la qualité des exécutables
- La préparation des parties prenantes
- La situation financière du projet en regard du budget initial



# Objectifs de la phase de transition

- Le déploiement du logiciel dans l'environnement d'exploitation des utilisateurs
- La prise en charge des problèmes liés à la transition
- Atteindre un niveau de stabilité tel que l'utilisateur est indépendant
- Atteindre un niveau de stabilité et qualité tel que les parties prenantes considèrent le projet comme terminé



# Activités de la phase de transition

- Activités de « packaging » du logiciel pour le mettre à disposition des utilisateurs et de l'équipe d'exploitation
- Correction des erreurs résiduelles et amélioration de la performance et du champ d'utilisation
- Evaluation du produit final en regard des critères d'acceptation définis



# **Livrables de la phase de transition**

- La version finale du logiciel
- Les manuels d'utilisation

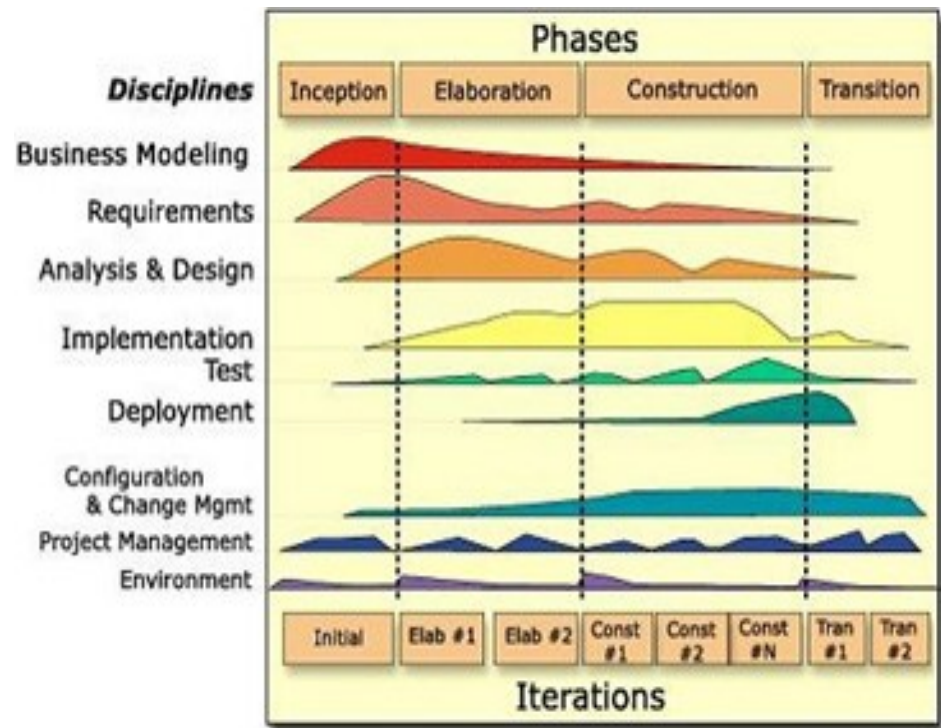


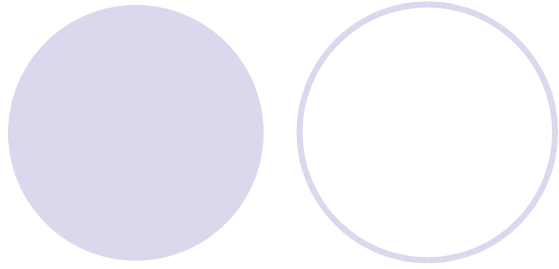
# Critères d'évaluation de la phase de transition

- La satisfaction des utilisateurs
- La situation financière du projet en regard du budget initial

# Organisation en activités de développement

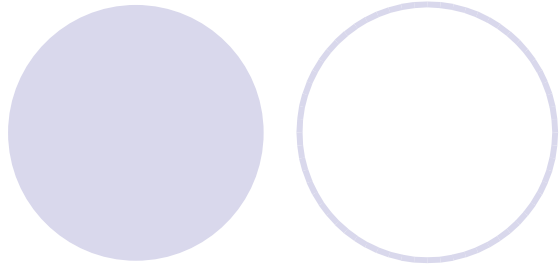
- Chaque phase comprend plusieurs itérations
- Chaque itération comprend plusieurs activités
  - Modélisation métier
  - *Expression des besoins*
  - *Analyse*
  - *Conception*
  - *Implémentation*
  - *Test*
  - Déploiement



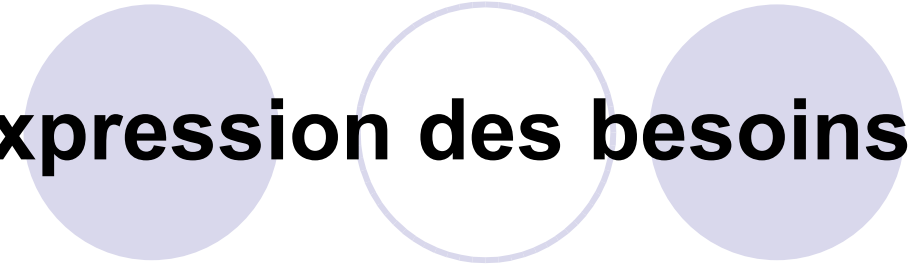


## Modélisation métier

- Mieux comprendre la structure et la dynamique de l'organisation.
  - Proposer la meilleure solution dans le contexte de l'organisation cliente.
  - Réalisation d'un glossaire des termes métiers.
  - Cartographie des processus métier de l'organisation cliente.
- Activité coûteuse mais qui permet d'accélérer la compréhension d'un problème



# Expression des besoins



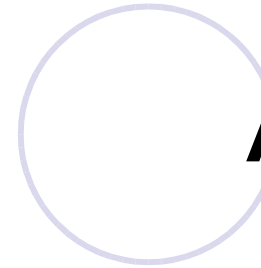
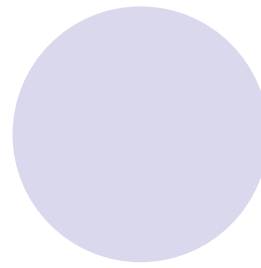
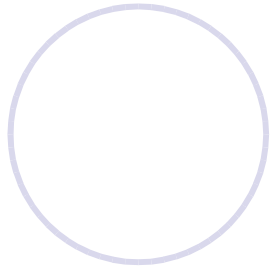
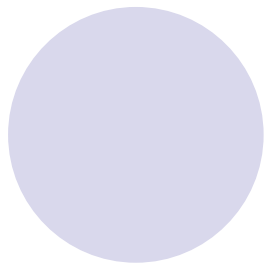
- Cibler les besoins des utilisateurs et du clients grâce à une
- série d'interviews.
  - L'ensemble des parties prenantes du projet, maîtrise d'oeuvre et maîtrise d'ouvrage, est acteur de cette activité.
- L'activité de recueil et d'expression des besoins débouche sur ce que doit faire le système (question « *QUOI ?* »)





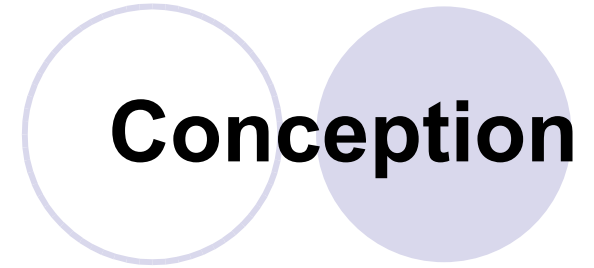
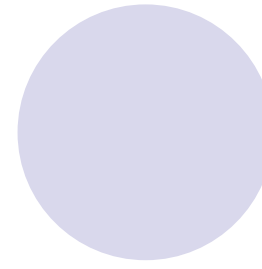
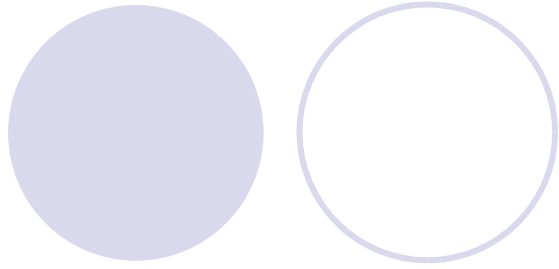
# Expression des besoins

- Utilisation des cas d'utilisation pour
  - Schématiser les besoins
  - Structurer les documents de spécifications fonctionnelles.
- Les cas d'utilisation sont décomposés en scénarios d'usage du système, dans lesquels l'utilisateur « raconte » ce qu'il fait grâce au système et ses interactions avec le système.
- Un maquettage est réalisable pour mieux « immerger » l'utilisateur dans le futur système.
- Une fois posées les limites fonctionnelles, le projet est planifié et une prévision des coûts est réalisée.

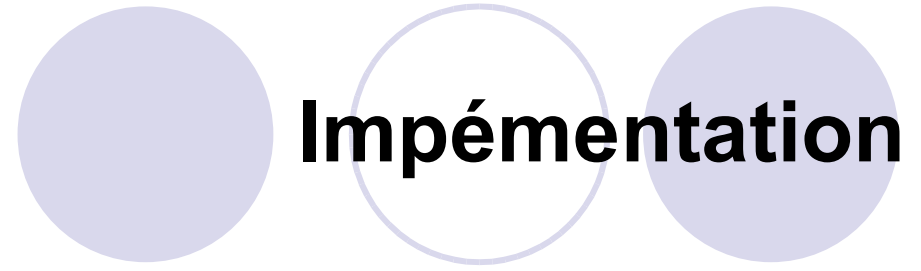
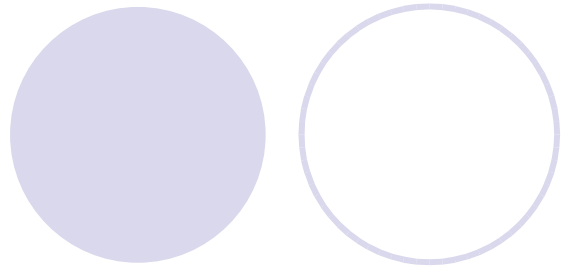


**Analyse**

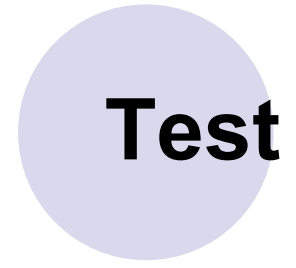
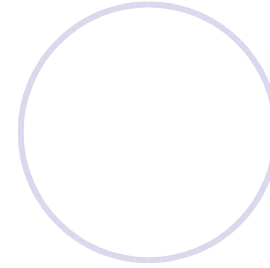
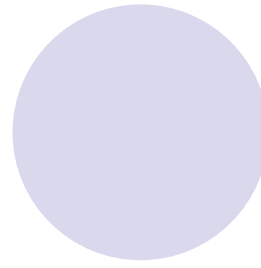
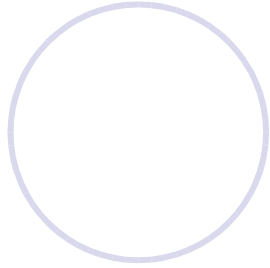
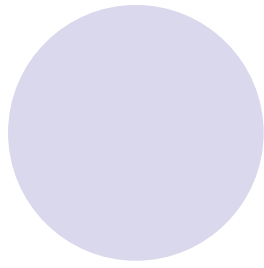
- Transformer les besoins utilisateurs en modèles UML
  - Analyse objet servant de base à une réflexion sur les mécanismes internes du système
- Principaux livrables
  - Modèles d'analyse, neutre vis à vis d'une technologie.
  - Livre une spécification plus précise des besoins
- Peut envisagé comme une première ébauche du modèle de conception



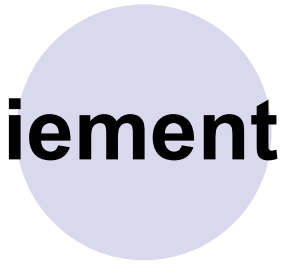
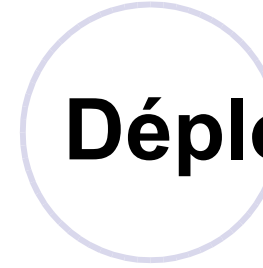
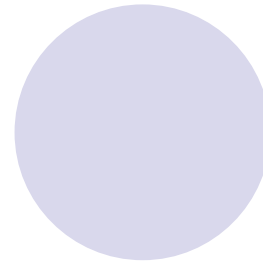
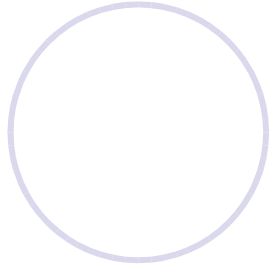
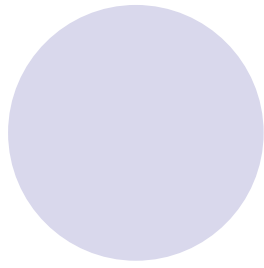
- La conception tient compte des
  - Exigences non fonctionnelles
  - Choix technologiques.
- Le système est analysé et on produit
  - Une proposition d'architecture.
  - Un découpage en composants.



- Concevoir le système par composants.
  - Le système est développé par morceaux dépendant les uns des autres.
  - Optimisation de l'utilisation des ressources selon leurs expertises.
- Les découpages fonctionnel et en couches sont indispensable pour cette activité.
- Il est tout à fait envisageable de retoucher les modèles d'analyse et de conception à ce stade.



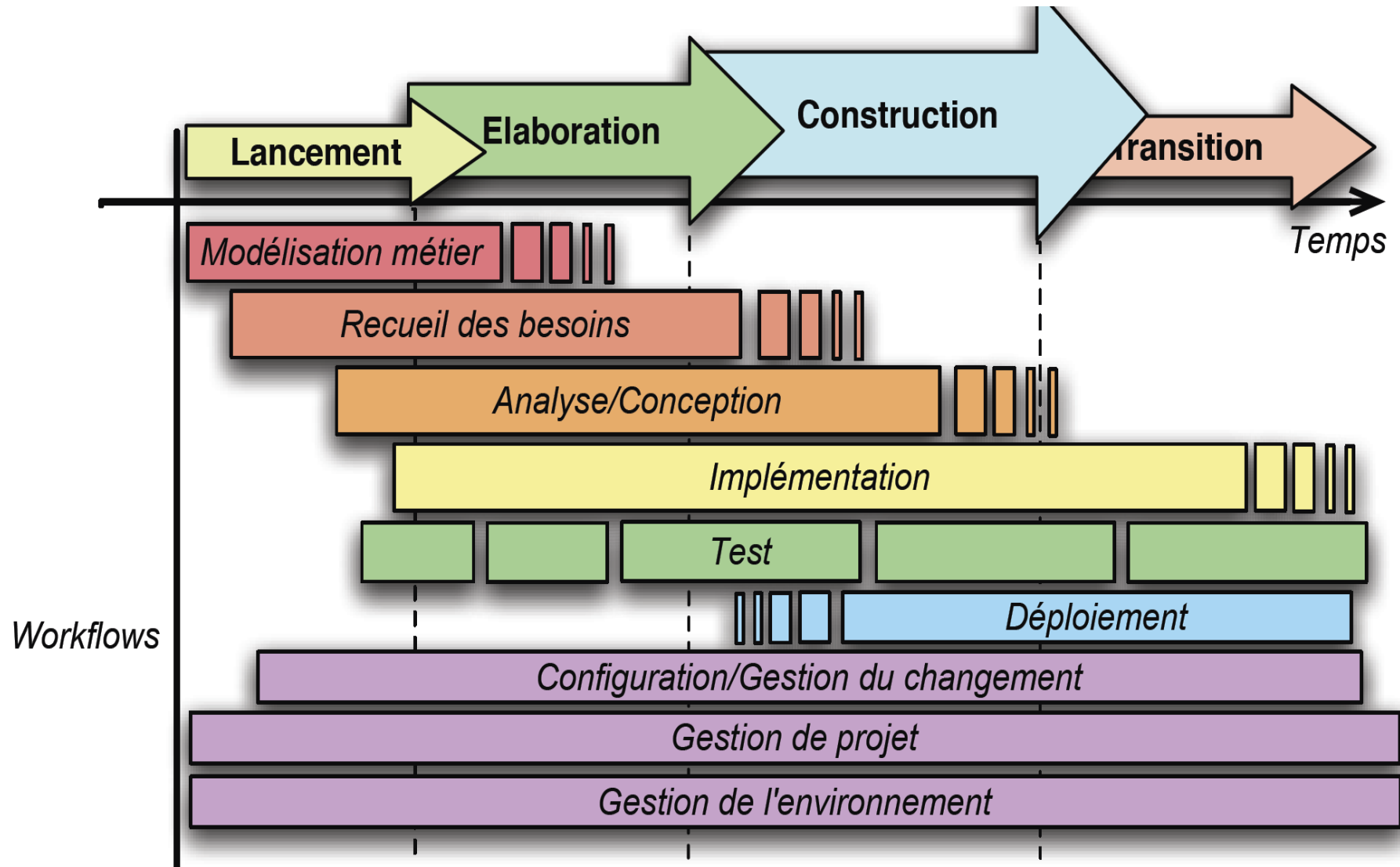
- Vérifier des résultats de l'implémentation en testant la construction.
  - Tests unitaires : tests composants par composants
  - Tests d'intégration : tests de l'interaction de composants préalablement testés individuellement
- Méthode :
  - Planification pour chaque itération
  - Implémentation des tests en créant des cas de tests
  - Exécuter les tests
  - Prendre en compte le résultat de chacun.



**Déploiement**

- Déployer les développements une fois réalisés.
  - Peut être réalisé très tôt dans le processus dans une sousactivité de prototypage dont l'objectif est de valider
    - l'architecture physique
    - les choix technologiques.

# Importance des activités dans chaque phase



# Principaux diagrammes UML par activité

- Expression des besoins et modélisation métier
  - Modèles métier, domaine, cas d'utilisation
  - Diagramme de séquences
- Analyse
  - Modèles métier, cas d'utilisation
  - Diagramme des classes, de séquences et de déploiement
- Conception
  - Diagramme des classes, de séquences
  - Diagramme Etat/transition
  - Diagramme de déploiement et de composant



**eXtreme Programming**



The diagram consists of five circles arranged in two rows. The top row has three circles, and the bottom row has two circles. The top-left circle is white with a light purple outline. The top-middle and top-right circles are solid light purple. The bottom-left and bottom-middle circles are solid light purple. The bottom-right circle is white with a light purple outline and contains the text 'UML 2.0'. The text 'eXtreme Programming' is positioned to the left of the top row of circles.

**UML 2.0**



# Méthodes agiles

*« Quelles activités pouvons nous abandonner tout en produisant des logiciels de qualité ? »*

*« Comment mieux travailler avec le client pour nous focaliser sur ses besoins les plus prioritaires et être aussi réactifs que possible ? »*

- Filiation avec le RAD
- Exemples de méthodes agiles
  - *XP* (eXtreme Programming), *DSDM* (Dynamic Software Development Method), *ASD* (Adaptative Software Development), *CCM* (Crystal Clear Methodologies), *SCRUM*, *FDD* (Feature Driven Development)



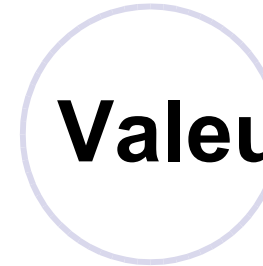
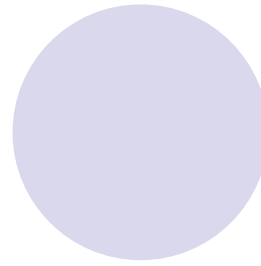
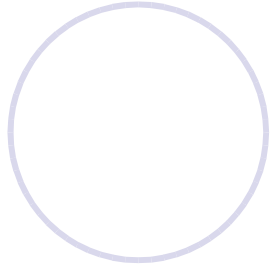
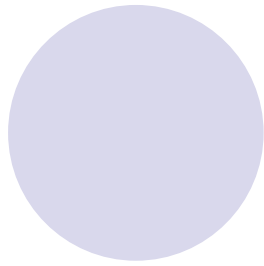
# Priorités des méthodes agiles

- Priorité *aux personnes et aux interactions* sur les procédures de les outils
- Priorité *aux applications fonctionnelles* sur une documentation pléthorique
- Priorité *à la collaboration avec le client* sur la négociation de contrat
- Priorité *à l'acceptation du changement* sur la planification



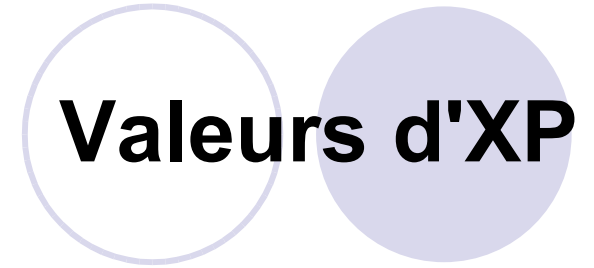
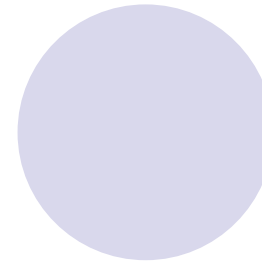
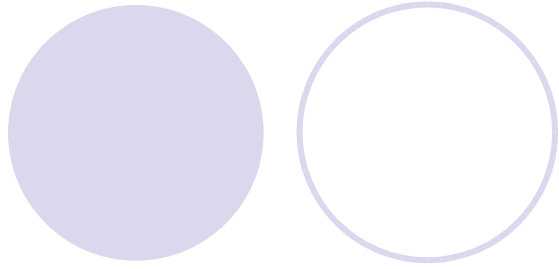
# eXtreme Programming

- Méthode « basée sur des pratiques qui sont autant de boutons poussés au maximum »
- Méthode qui peut sembler naturelle mais concrètement pas facile à appliquer et à maîtriser
  - Réclame beaucoup de discipline et de communication (contrairement à la première impression qui peut faire penser à une ébullition de cerveaux individuels).
  - Aller vite mais sans perdre de vue la rigueur du codage et les fonctions finales de l'application.
- Force de XP : sa simplicité et le fait qu'on va droit à l'essentiel, selon un rythme qui doit rester constant.

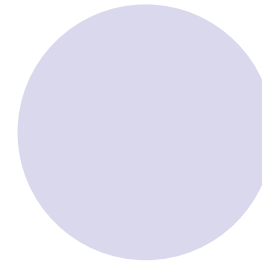
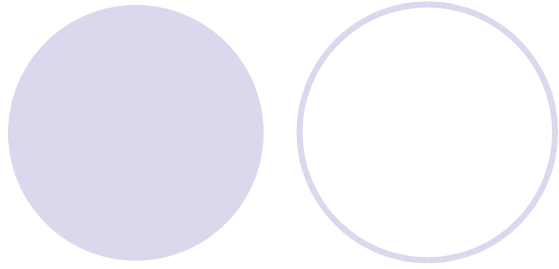


## Valeurs d'XP

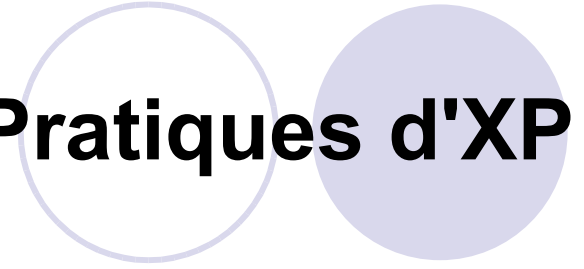
- Communication
  - XP favorise la communication directe, plutôt que le cloisonnement des activités et les échanges de documents formels.
  - Les développeurs travaillent directement avec la maîtrise d'ouvrage
- Feedback
  - Les pratiques XP sont conçues pour donner un maximum de feedback sur le déroulement du projet afin de corriger la trajectoire au plus tôt.



- Simplicité :
  - Du processus
  - Du code
- Courage
  - d'honorer les autres valeurs
  - de maintenir une communication franche et ouverte
  - d'accepter et de traiter de front les mauvaises nouvelles.



# Pratiques d'XP



- 13 pratiques réparties en 3 catégories
  - Gestion de projets
  - Programmation
  - Collaboration



# Pratiques de gestion de projets

- Livraisons fréquentes
  - L'équipe vise la mise en production rapide d'une version minimale du logiciel, puis elle fournit ensuite régulièrement de nouvelles livraisons en tenant compte des retours du client.
- Planification itérative
  - Un plan de développement est préparé au début du projet, puis il est revu et remanié tout au long du développement pour tenir compte de l'expérience acquise par le client et l'équipe de développement.





# Pratiques de gestion de projets

- Client sur site
  - Le client est intégré à l'équipe de développement pour répondre aux questions des développeurs et définir les tests fonctionnels.
- Rythme durable
  - L'équipe adopte un rythme de travail qui lui permet de fournir un travail de qualité tout au long du projet.
  - Jamais plus de 40h de travail par semaine (un développeur fatigué développe mal)



# Pratiques de programmation

- Conception simple
  - On ne développe rien qui ne soit utile tout de suite.
- Remaniement
  - Le code est en permanence réorganisé pour rester aussi clair et simple que possible.



# Pratiques de programmation

- Tests unitaires
  - Les développeurs mettent en place une batterie de tests de nonrégression qui leur permettent de faire des modifications sans crainte.
- Tests de recette
  - Les testeurs mettent en place des tests automatiques qui vérifient que le logiciel répond aux exigences du client.
  - Ces tests permettent des recettes automatiques du logiciel.



# Pratiques de collaboration

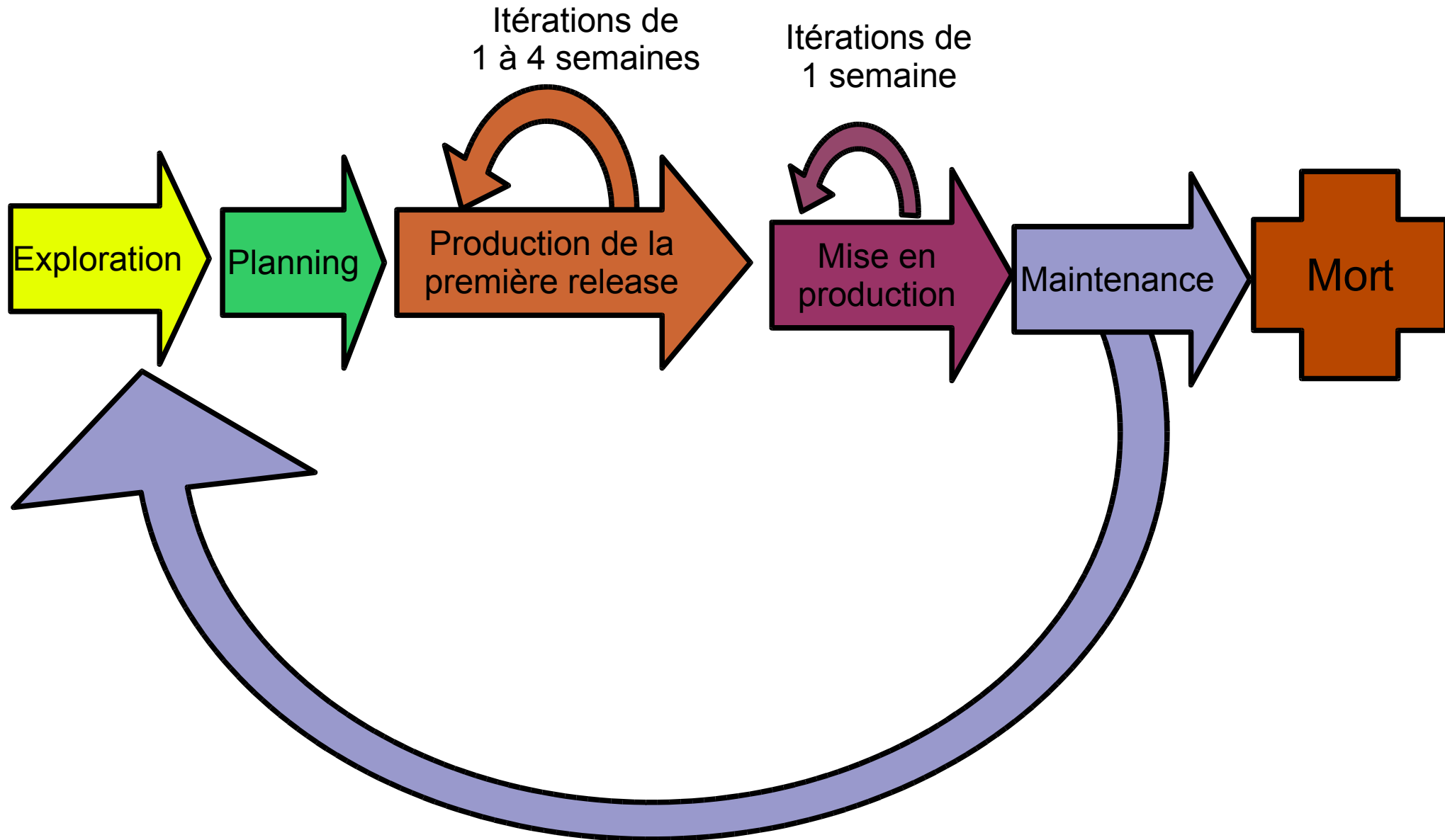
- Responsabilité collective du code
  - Chaque développeur est susceptible de travailler sur n'importe quelle partie de l'application.
- Programmation en binômes
  - Les développeurs travaillent toujours en binômes, ces binômes étant renouvelés fréquemment.

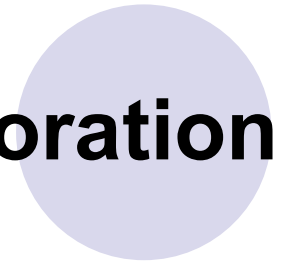
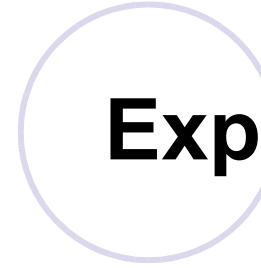
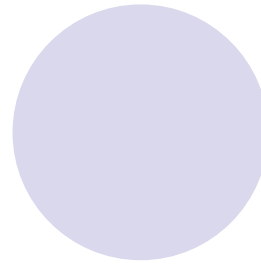
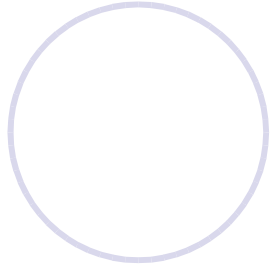
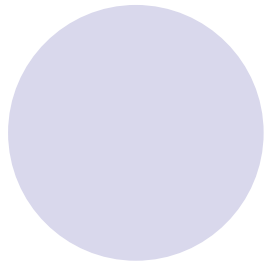


# Pratiques de collaboration

- Règles de codage
  - Les développeurs se plient à des règles de codage strictes définies par l'équipe elle-même.
- Métaphore
  - Les développeurs s'appuient sur une description commune du design.
- Intégration continue
  - L'intégration des nouveaux développements est faite chaque jour.

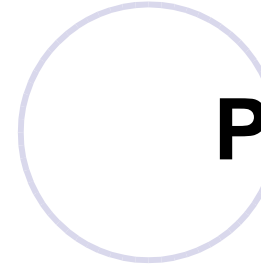
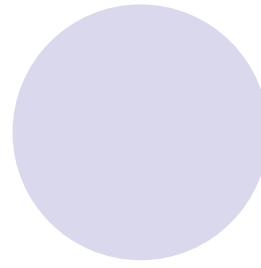
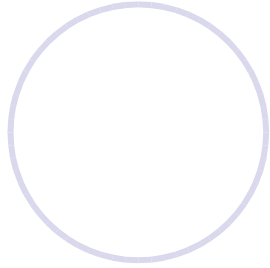
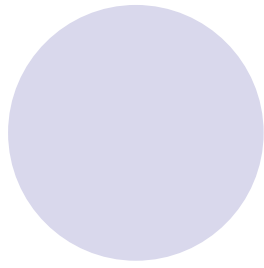
# Cycle de vie XP





## Exploration

- Les développeurs se penchent sur des questions techniques
  - Explorer les différentes possibilités d'architecture pour le système
  - Etudier par exemple les limites au niveau des performances présentées par chacune des solutions possibles
- Le client s'habitue à exprimer ses besoins sous forme de user stories (proches de diagrammes de cas illustrés par des interactions)
  - Les développeurs estiment les temps de développement



# Planning

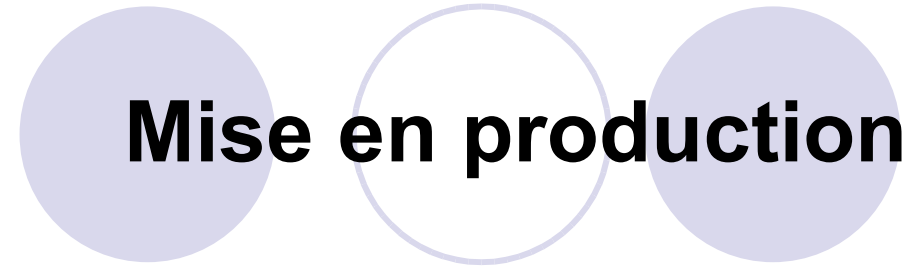
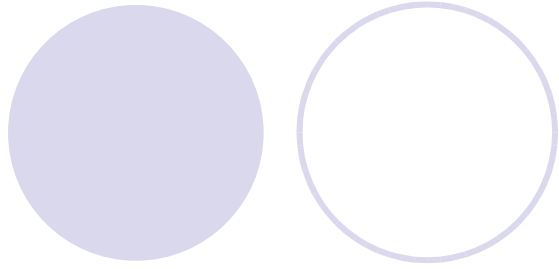
- Planning de la première release :
  - Uniquement les fonctionnalités essentielles
  - Première release à enrichir par la suite
- Durée du planning : 1 ou 2 jours
- Première release au bout de 2 à 6 mois



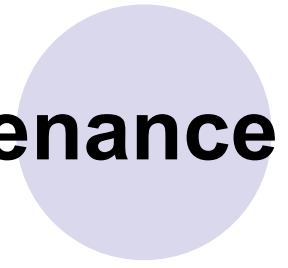
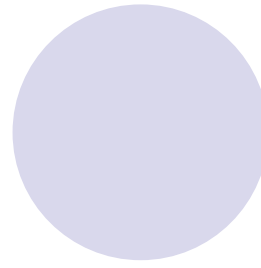
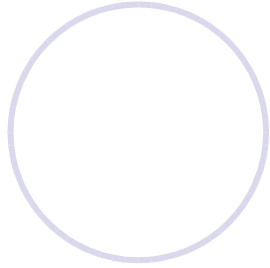
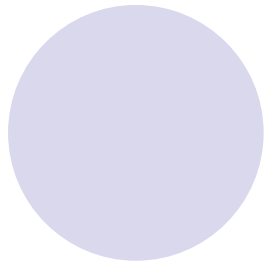


# Itérations jusqu'à la première release

- Développement de la première version de l'application
- Itérations de une à quatre semaines
  - Chaque itération produit un sous ensemble des fonctionnalités principales
  - Le produit de chaque itération subit des tests fonctionnels
  - Itérations courtes pour identifier très tôt des déviations par rapport au planning
- Brèves réunions quotidiennes réunissant toute l'équipe, pour mettre chacun au courant de l'avancement du projet

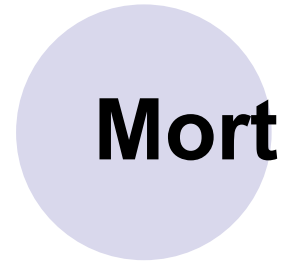
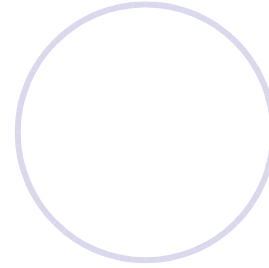
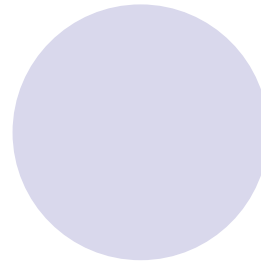
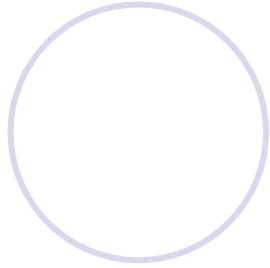
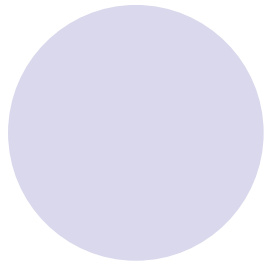


- La mise en production produit un logiciel
  - Offrant toutes les fonctionnalités indispensables
  - Parfaitement fonctionnel
  - Mis à disposition des utilisateurs
- Itérations très courtes
- Tests constants en parallèle du développement
- Les développeurs procèdent à des réglages affinés pour améliorer les performances du logiciel

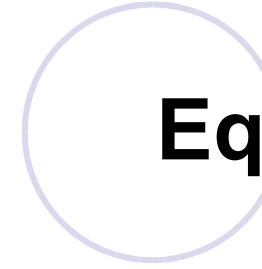
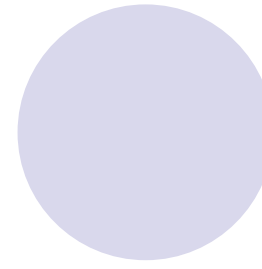
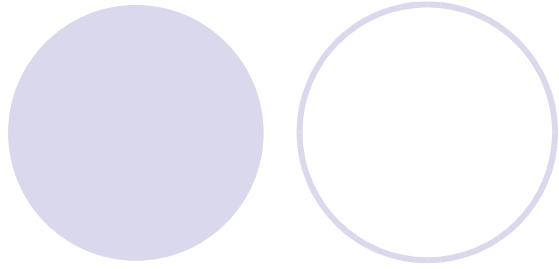


## Maintenance

- Continuer à faire fonctionner le système
- Adjunction de nouvelles fonctionnalités secondaires
  - Pour les fonctionnalités secondaires, on recommence par une rapide exploration
  - L'ajout de fonctionnalités secondaires donne lieu à de nouvelles releases

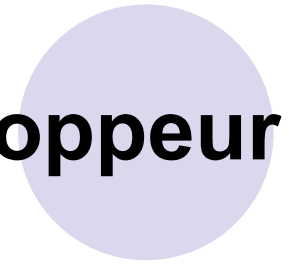
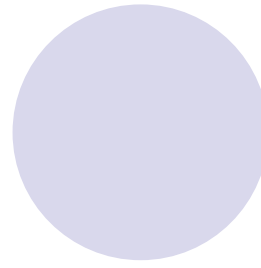
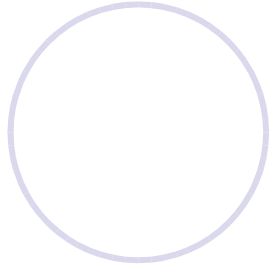
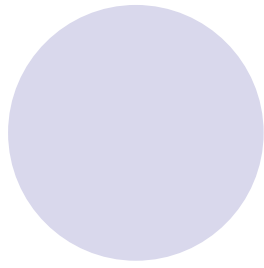


- Quand le client ne parvient plus à spécifier de nouveaux besoins, le projet est dit « mort »
  - Soit que tous les besoins possibles sont remplis
  - Soit que le système ne supporte plus de nouvelles modifications en restant rentable



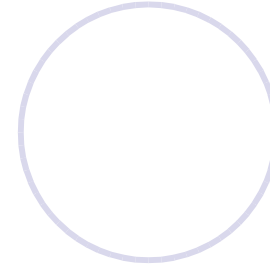
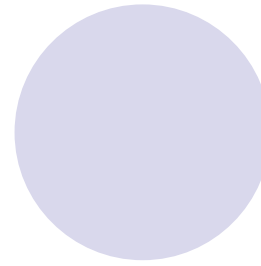
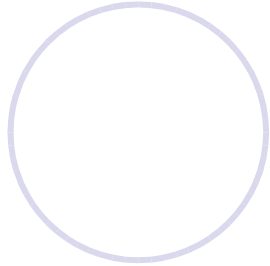
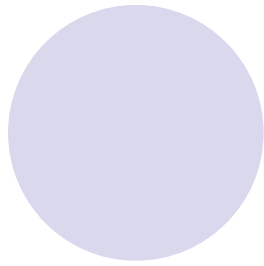
**Equipe XP**

- Pour un travail en équipe, on distingue 6 rôles principaux au sein d'une équipe XP
  - Développeur
  - Client
  - Testeur
  - Tracker
  - Manager
  - Coach

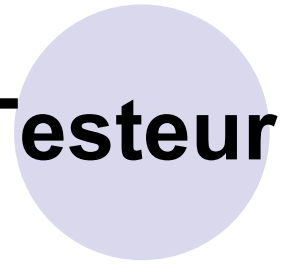
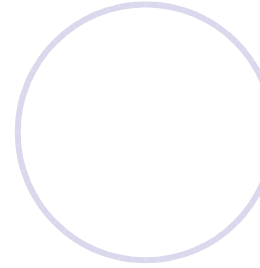
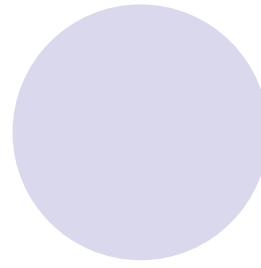
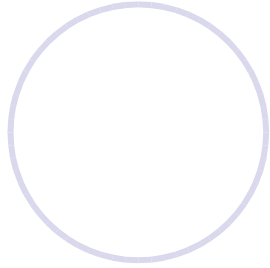
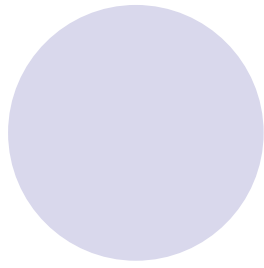


**Développeur**

- Conception et programmation, même combat !
- Participe aux séances de planification, évalue les tâches et leur difficulté
- Définition des test unitaires
- Implémentation des fonctionnalités et des tests unitaires

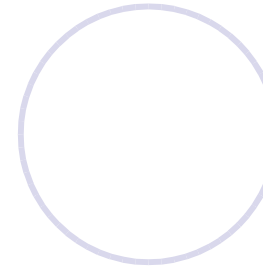
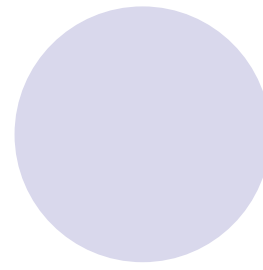
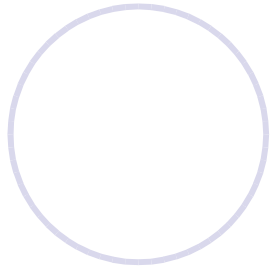
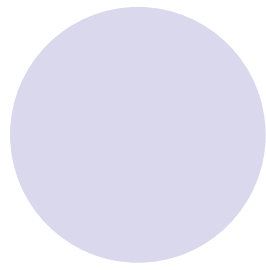


- Écrit, explique et maîtrise les scénarios
- Spécifie les tests fonctionnels de recette
- Définit les priorités



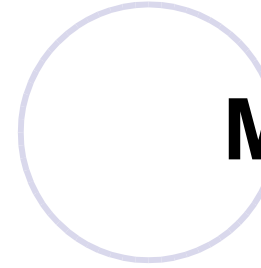
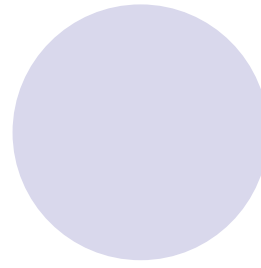
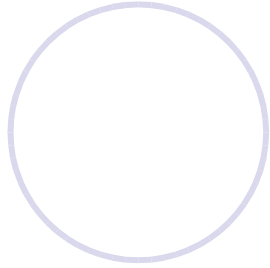
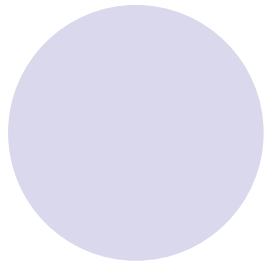
- Écriture des tests de recette automatiques pour valider les scénarios clients
- Peut influencer sur les choix du clients en fonction de la « testatibilité » des scénarios



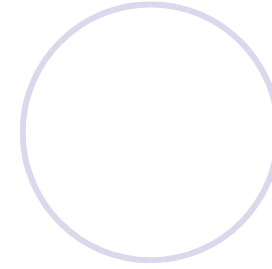
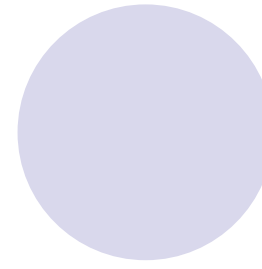
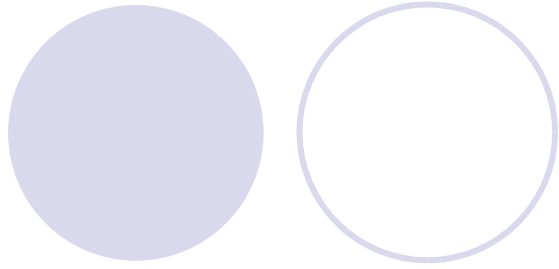


**Tracker**

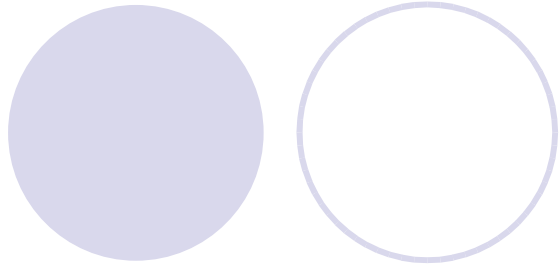
- Suivre le planning pour chaque itération.
- Comprendre les estimations produites par les développeurs concernant leur charges
- Interagir avec les développeurs pour le respect du planning de l'itération courante
- Détection des éventuels retards et rectifications si besoin



- Supérieur hiérarchique des développeurs
  - Responsable du projet
- Vérification de la satisfaction du client
- Contrôle le planning
- Gestion des ressources

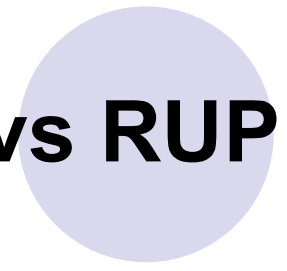
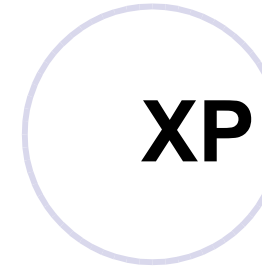
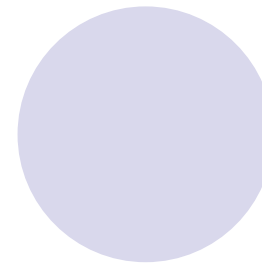
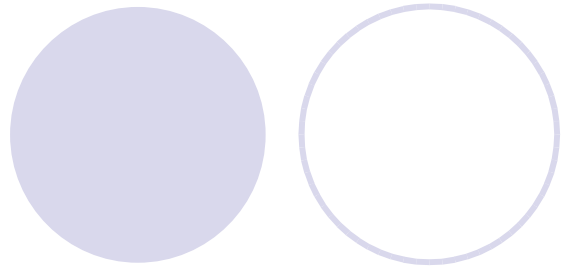


- Garant du processus XP
  - Organise et anime les séances de planifications
  - Favorise la créativité du groupe, n'impose pas ses solutions techniques
  - Coup de gueules...



## **Spécification avec XP**

- Pas de documents d'analyse ou de spécifications détaillées
- Les tests de recette remplacent les spécifications
- Emergence de l'architecture au fur et à mesure du développement



## XP vs RUP

- Inconvénients de XP
  - Focalisation sur l'aspect individuel du développement, au détriment d'une vue globale et des pratiques de management ou de formalisation
  - Manquer de contrôle et de structuration, risques de dérive
- Inconvénients de RUP
  - Fait tout, mais lourd, « usine à gaz »
  - Parfois difficile à mettre en oeuvre de façon spécifique.
- XP pour les petits projets en équipe de 12 max, RUP pour les gros projets qui génèrent beaucoup de documentation

**And now, something completely different...**



**...the Larch.**