

UML et la programmation orientée objet

Historique de la POO

Simula (1966) regroupe données et procédures.

Simula I (1972) formalise les concepts d'objet et de classe. Un programme devient une collection d'objets actifs et autonomes.

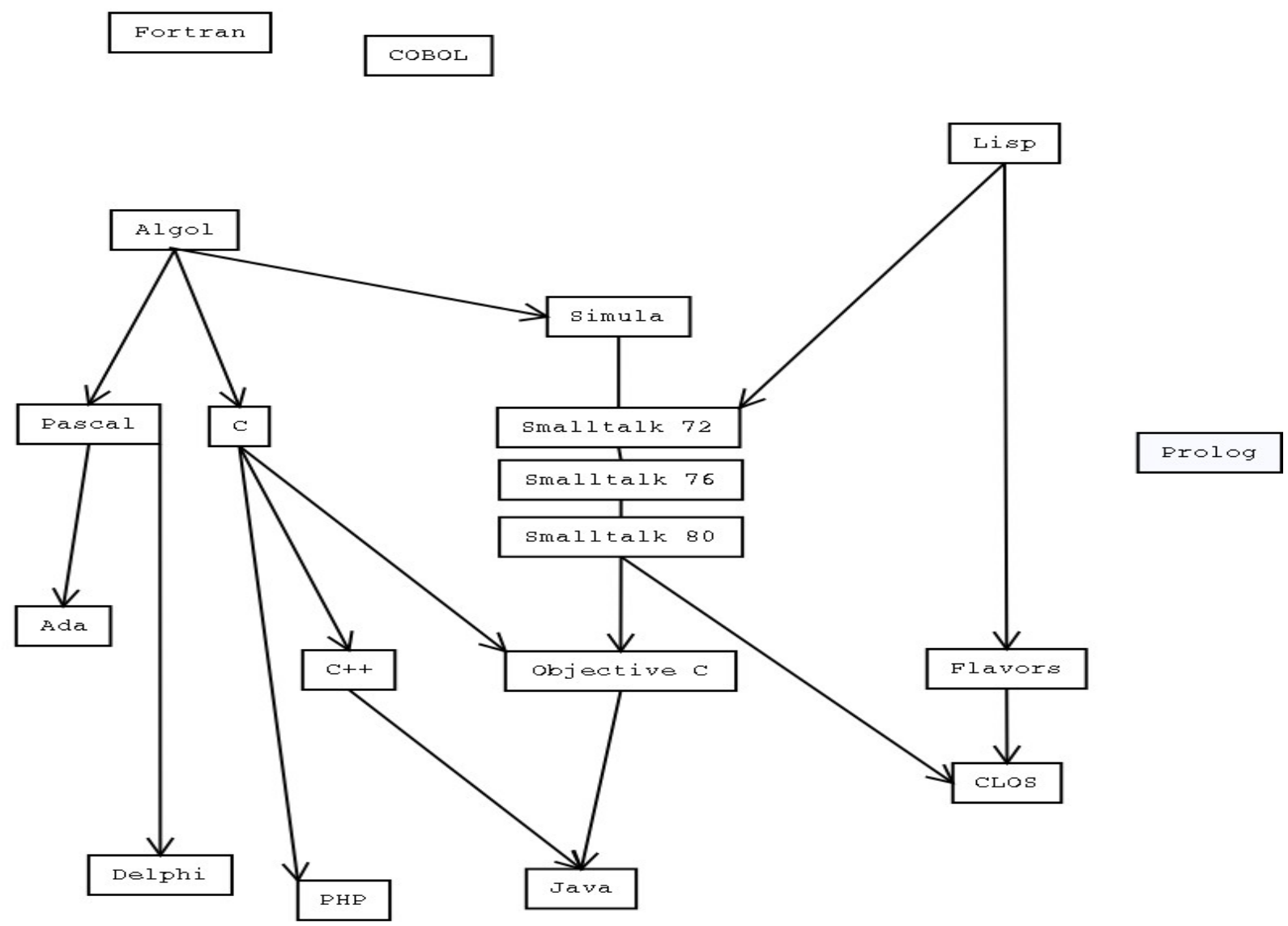
Smalltalk (1972) : généralisation de la notion d'objet.

C++ (1983) : Extension du C s'inspirant de Simula.

Java (1994) : d'abord pour le web puis généraliste.

Autres : C#, eiffel, ada, clu, Object pascal, delphi, python, etc..

50
60
70
80
90



Motivation et avantages de POO

Motivation : concevoir, maintenir et exploiter facilement de gros logiciels.

Avantages : modularité, abstraction, productivité et réutilisabilité, sûreté, encapsulation.

Les autres approches : structurée impérative (C), fonctionnelle (Lisp) et logique (Prolog).

Modularité et Abstraction

Modularité : les objets forment des modules compacts regroupant des données et un ensemble d'opérations.

Abstraction : Les entités objets de la POO sont proches de celles du monde réel. Les concepts utilisés sont donc proches des abstractions familières que nous exploitons.

productivité et réutilisabilité, sûreté

productivité et réutilisabilité : Plus l'application est complexe et plus l'approche POO est intéressante en terme de productivité. Le niveau de réutilisabilité est supérieur à la programmation impérative.

sûreté : L'encapsulation et le typage des classes offrent une certaine robustesse aux applications.

Les concepts fondamentaux

- Objet, classe et instance
- les membres de classe et d'instance
- envoi de message et méthode
- héritage, encapsulation et polymorphisme
- Constructeur et destructeur
- Classe abstraite / concrète

Objet

Les objets sont omniprésents (humain, livre, etc.) dans notre monde.

En POO :

objet = identité + états + comportements

L'identité doit permettre d'identifier sans ambiguïté l'objet (adresse/référence ou nom).

Modélisation informatique : les états sont stockés dans des propriétés (variables) et les comportements sont implémentés à l'aide de méthodes (procédures / fonctions).

Exemple d'un objet

Soit la modélisation d'un objet être humain.

Son identité peut être son nom ou bien un numéro.

Ses états seront sa taille, son poids, la couleur de ses yeux, etc..

Ses comportements seront respirer, marcher, parler, etc..

Classe

Le monde réel regroupe des objets du même type.

Il est pratique de concevoir une maquette (un moule) d'un objet et de produire les objets à partir de cette maquette. En POO, une maquette se nomme une **classe**.

Une classe est donc un modèle de la structure statique (variables d'instance) et du comportement dynamique (les méthodes) des objets associés à cette classe.

Instance

Les objets associés à une classe se nomment des **instances**.

Une instance est un objet, occurrence d'une classe, qui possède la structure définie par la classe et sur lequel les opérations définies dans la classe peuvent être appliquées.

Membre

Membre = propriété + méthode

- Membre de classe
- Propriété de classe

Une propriété de classe est associée à sa classe et non à une instance de cette classe.

- Méthode de classe

Une méthode de classe est associée à une classe et non à un objet.

Une méthode de classe ne peut exploiter que des membres de classe.

Membre (suite)

- Membre d'instance
- Propriété d'instance

Une propriété d'instance est associée à une instance de la classe et non à la classe. Chaque objet possède donc sa propre copie de la propriété.

- Méthode d'instance

Une méthode d'instance est associée à une instance d'une classe et non à la classe. Chaque objet possède donc sa propre copie de la méthode. Une méthode d'instance peut utiliser n'importe quel type de membre (classe ou instance).

Notion de message

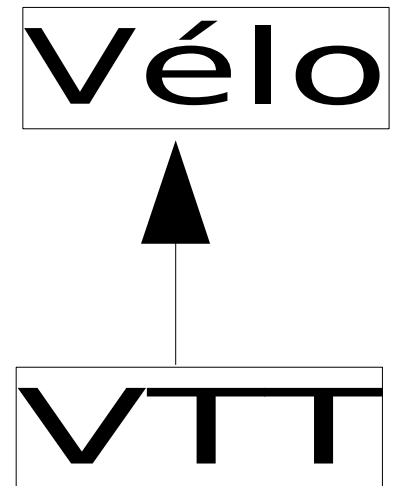
Un objet seul ne permet pas de concevoir une application garantissant les objectifs de la POO.
Un programme est constitué d'objets. Ces derniers communiquent à l'aide de messages.
Un message est composé : du nom de l'objet recevant le message, du nom de la méthode à exécuter et des paramètres nécessaires à la méthode.



Héritage

Les objets sont définis à partir de classes. En POO, les classes sont définies à partir d'autres classes. Par exemple : un VTT est une sous-classe de Vélo.

Une **sous-classe** n'est pas limitée aux caractéristiques de sa **super-classe**. Elle peut étendre cette dernière avec de nouvelles propriétés et méthodes.



Encapsulation

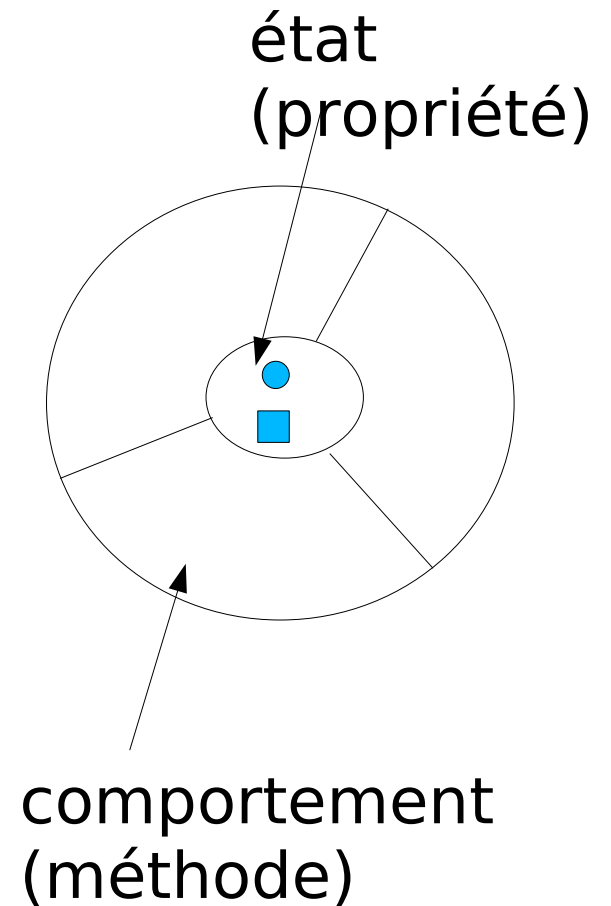
L'**encapsulation** est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés.

Ce mécanisme permet donc de garantir l'intégrité des données contenues dans l'objet.

Encapsulation (suite)

L'encapsulation permet de définir le niveau de visibilité des éléments de la classe. Ils définissent les droits d'accès :

- **publique**
- **privée**
- **protégée.**



Polymorphisme

Le **polymorphisme** désigne un concept de la théorie des types, selon lequel un nom d'objet peut désigner des instances de classes différentes issues d'une même arborescence.

Exemple : ObjetGraphique

Rectangle Cercle

```
graph BT; Rectangle --> ObjetGraphique; Cercle --> ObjetGraphique;
```

Les classes Rectangle et Cercle héritent de la classe ObjetGraphique. On peut définir une instance Rect1 d'un Rectangle comme un ObjetGraphique.

Constructeur

Un constructeur est une opération de classe gérant la construction d'objets. Le nom de cette méthode est identique à celui de sa classe.

Destructeur

Un destructeur est une opération de classe qui détruit des objets. Cette méthode libère donc de l'espace mémoire.

Classe concrète/ abstraite

Une classe concrète est une classe qu'il est possible d'instancier.

Ce n'est pas le cas d'une classe abstraite. Une classe abstraite relève purement de l'organisation et son objectif est uniquement lié à la spécification (description exhaustive d'un élément d'une modélisation).

Complexité des logiciels

- Les tendances actuelles :
 - Programmation avec génération automatique du code
 - Micro-architectures (patterns).
 - Importance de l'architecture
 - Informatique distribuée
 - Multimédia.

Modélisation

- Proposer une structure pour la résolution de problèmes (généricité)
- Exploration de multiples solutions.
- Proposer une abstraction pour gérer la complexité (compréhension du problème et approche de la solution)
- Syntaxe et sémantique
- Réduire les coûts (financiers – temps).



UML



- ◆ UML : Unified Modeling Language
- ◆ Un langage graphique permettant de spécifier, visualiser, construire et documenter les artefacts de systèmes logiciels.
- ◆ C'est un langage de modélisation, pas une méthode (Cf. RUP : Rational Unified Process).
- ◆ Standardisation d'une notation mais ne dit rien sur comment appliquer cette notation.
- ◆ Un métamodèle pour la modélisation orientée objet.

Origines

Booch	Catégories et sous-systèmes
Embley	Classes singletons et objets composites
Fusion	Description des opérations, numérotation des messages
Gamma, et al.	<i>Frameworks, patterns</i> , et notes
Harel	Automates (<i>Statecharts</i>)
Jacobson	Cas d'utilisation (<i>use cases</i>)
Meyer	Pré- et post-conditions
Odell	Classification dynamique, éclairage sur les événements
OMT	Associations
Shlaer-Mellor	Cycle de vie des objets
Wirfs-Brock	Responsabilités (CRC)

Origines

1.1 -> 1.2. -> 1.3 -> 1.4 -> 1.5 -> 2.0



Soumission de UML 1.0 à OMG pour adoption (janvier 1997).

Industrialisation



Standardisation



Unification



Fragmentation

UML 1.0

(juin 96 - oct. 96) **UML 0.9 & 0.91**

UML

partners expertise

OOPSLA'95 Unified Method 0.8

Booch 93

OMT-2

Autres méthodes

Booch 91

OMT-1

OOSE

(≈ 50)

Grady Booch

Jim Rumbaugh

public feedback

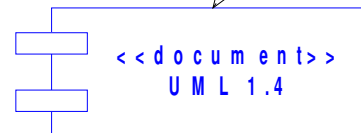
Historique UML

2001
(planned major revision)



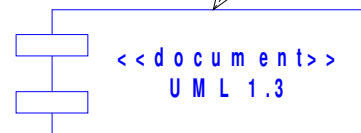
<<refine>>

Q3 2000
(planned minor revision)



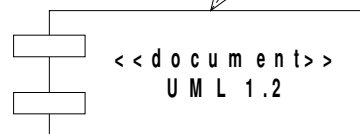
<<refine>>

Q3 1999



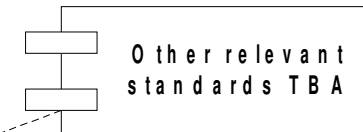
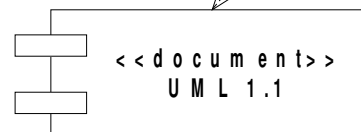
<<refine>>

Q2 1998

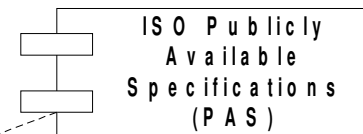


<<refine>>

Q3 1997
(OMG Adopted Technology)



<<informalLiaison>>



<<formalLiaison>>

Editorial revision with no significant technical changes.

Unification of major modeling languages, including Booch, OMT and Objectory

UML : Objectifs

- Définir un langage de modélisation graphique facile à apprendre et sémantiquement riche.
- Possibilités de communication entre acteurs.
- Incorporer des “best practices” du domaine industriel.
- Gérer des questions clés du développement de logiciels : Distribution, Concurrence, Montée en charge.

Processus de développement simplifié

- Vue d'ensemble d'un processus (itératif et incrémental) de développement :
 - Initialisation :: établir les règles de gestion du projet et sa portée.
 - Elaboration :: recueillir les besoins, définir une architecture de base et création d'un plan de construction.
 - Construction :: comprend de nombreuses itérations avec les étapes habituelles : analyse, conception, implémentation et tests.
 - Transition :: bêta-tests, optimisation du code et formation des utilisateurs.

Processus : élaboration

- Cette phase présente plusieurs risques liés aux :
 - spécifications :: bien définir les besoins de l'application
 - technologies :: les technologies adoptées sont-elles les bonnes ?
 - compétences :: conception d'une équipe compétente.

Risques liés aux spécifications

- Les **cas d'utilisation** d'UML constituent le point de départ car ils pilotent tout le processus de développement.
- Un cas d'utilisation représente l'utilisation typique d'un utilisateur avec le système lorsqu'il s'agit d'atteindre un objectif donné.
- Mais il ne présente pas une image globale de l'application. Pour cela, on exploite un modèle du domaine.
- Pour représenter ce modèle du domaine nous exploitons le **diagramme de classe**.

Risques liés aux spécifications (2)

- Si le domaine a également une forte composante sur le flux des données, nous pouvons la représenter au moyen de **diagramme d'activités**. Ils encouragent la découverte de processus parallèles.
- On peut également exploiter les **diagrammes d'interaction** pour explorer les différents rôles joués par les participants au développement du projet et les interactions.

Risques liés aux spécifications (3)

- Pour les classes dont le cycle de vie est intéressant, nous pouvons consolider les diagrammes de classe et d'activité par un **diagramme d'états**.
- Principal risque : accès à l'expertise.

Risques liés aux technologies

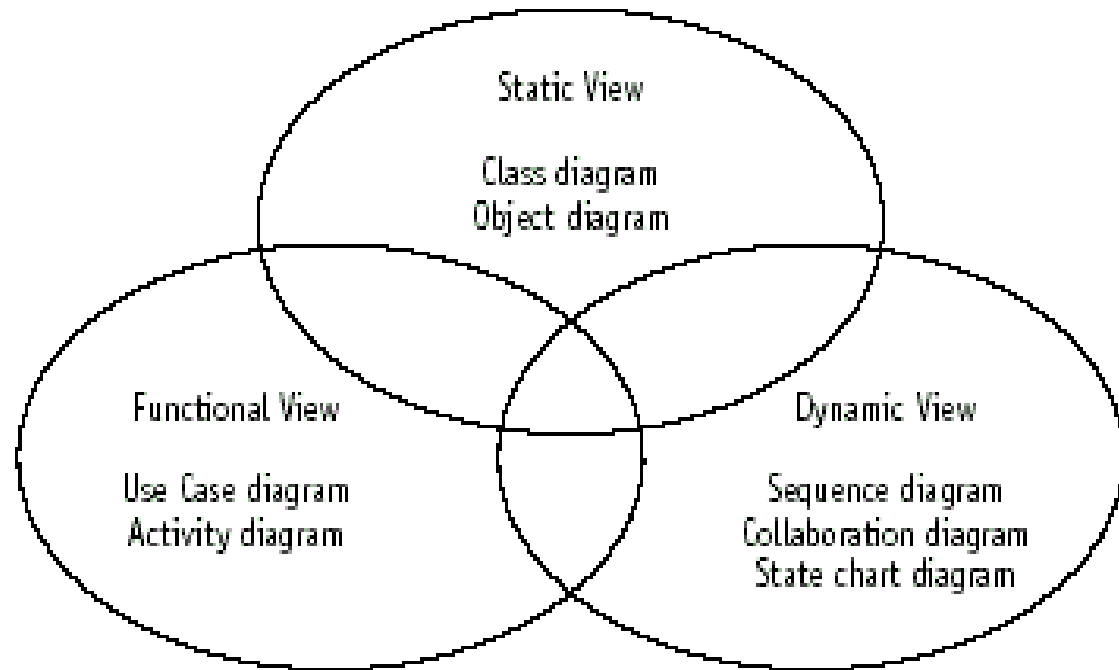
- Un problème important : collaboration entre technologies (ex : lang de prog. et bd).
- On peut examiner les cas d'utilisation pour déterminer un élément qui risque de bloquer votre conception.
- Utilité des techniques UML :
 - Diag. De classe et d'interaction sont utiles pour mettre en évidence la façon dont les composants communiquent.
 - **Diag. De déploiement** peuvent fournir une vue d'ensemble sur la distribution des éléments.
 - **Diag. De package** peuvent montrer une image de haut niveau des composants.

Construction et planification

- Les cas d'utilisation servent de base à la planification du projet, et c'est la raison pour laquelle UML y attache beaucoup d'importance.
-

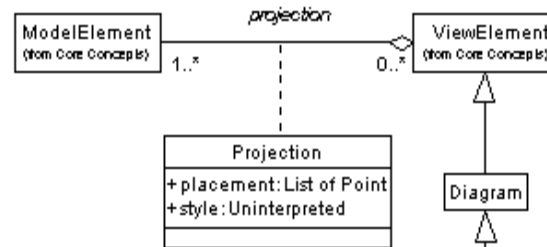
Vues UML

- Une vue est une collection de diagrammes décrivant un aspect du projet.

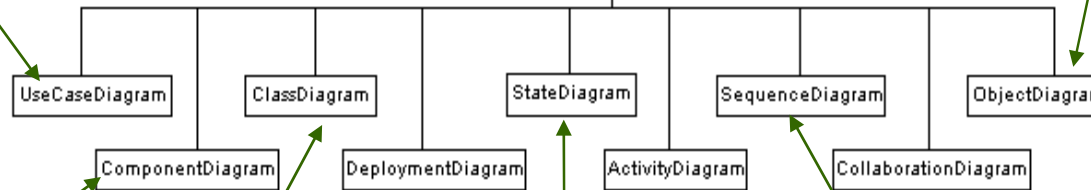


Vue générale

Fonctions du système
du point de vue
de l'utilisateur.



Les objets et les relations de base
entre ces objets.



Composants
physiques
d'une
application.

Représentation
du comportement
en termes d'états.

Représentation des objets,
des liens mutuels et des
interactions potentielles.

Schémas de l'installation
des composants sur les
dispositifs matériels.

Représentation des objets
et de leurs interactions temporelles.

Représentation
du comportement
des opérations
en termes d'actions.

Structure statique des classes et des relations entre ces classes.

Les diagrammes

- Diagrammes structuraux :
 - Classe, objet, composant et déploiement.
- Diagrammes comportementaux
 - Case d'utilisation, activité, séquence, collaboration, états.

Vue dynamique

- Cette vue permet de modéliser les interactions entre objets.
- Elle regroupe les diagrammes de :
 - Séquence
 - Collaboration
 - “Statechart” qui propose une vision des actions d'un objet en réaction aux stimuli externes.

Vue statique

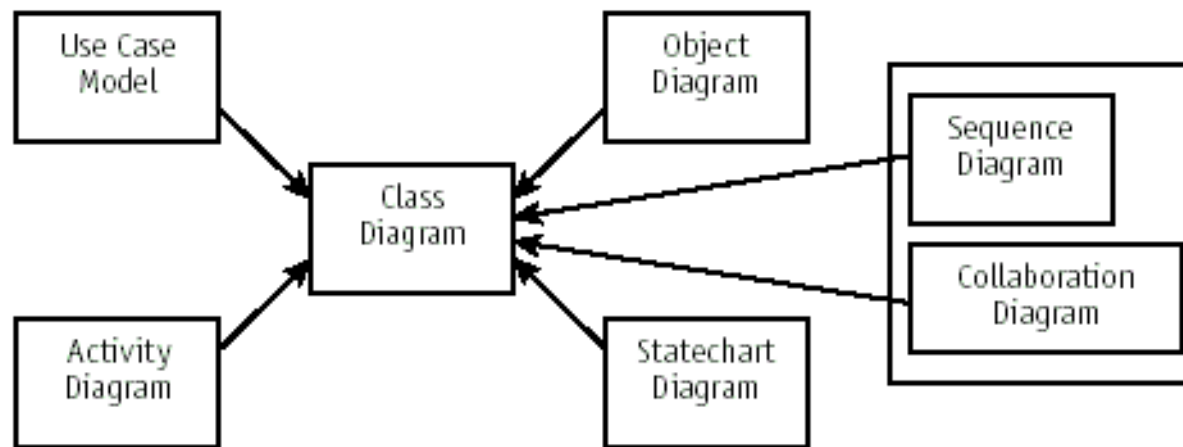
- Le **diagramme de classes** est le principal diagram de la vue statique. Il est à la base de la modélisation des types des objets, la génération du code source et la cible visée par le « reverse engineering ».
- Le **diagramme d'objets** illustre les faits sous la forme d'objets dans l'optique de présenter des exemples et de tester le comportement de l'application. Ce diagramme est utiliser pour tester l'application et/ou comprendre le diagramme de classes à partir d'exemples.

Diagramme de classe

Diagramme de classes

- Il représente les classes, leurs composants et les liens entre classes.
- Ce diagramme regroupe :
 - les attributs
 - Les opérations
 - Les stéréotypes (aident à la compréhension de la classe dans le contexte d'autres classes ayant des rôles similaires dans le système.
 - Les propriétés des classes
 - Les associations (simples, agrégations, compositions, qualifiés et réflexives).
 - Les relations de généralisation/spécialisation (héritage) entre classes.

Importance du diagramme de classes

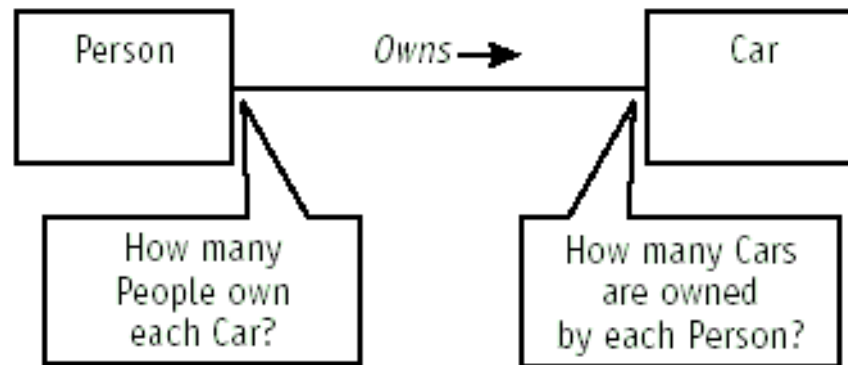


Diagrammes de classes : attributs

- Visibilité
 - Publique (+) offre l'accès à l'ensemble des objets de toutes les classes.
 - Privée (-) limite l'accès à la classe.
 - Protégée (#) offre l'accès aux sous-classes de la classe.
 - Package (~) offre l'accès aux autres objets issus de classe du package.

Diagrammes de classes : associations

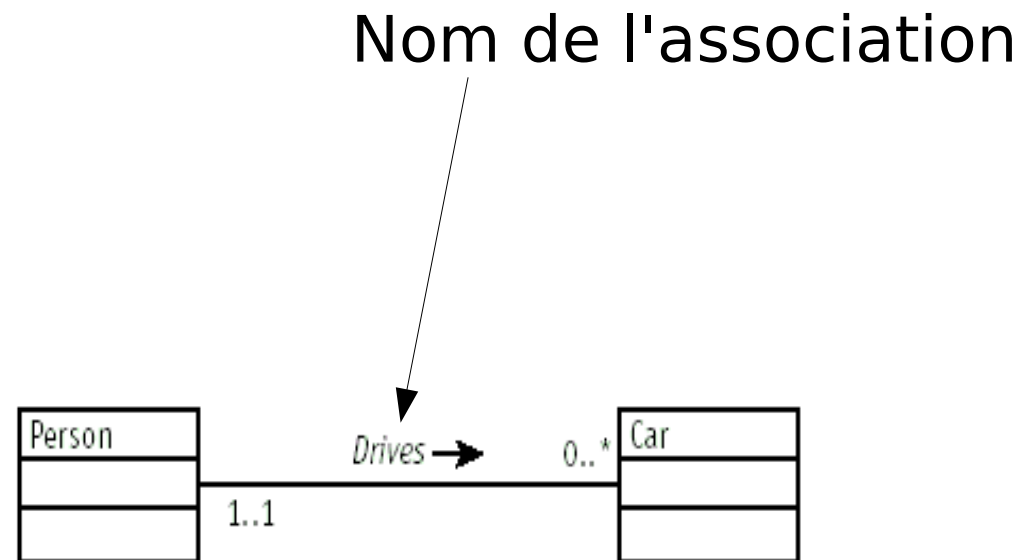
- Une association permet une référence entre classes en mettant en évidence la non redondance des informations (une information élémentaire ne doit figurer qu'à un seul endroit du diagramme de classes).



Diagrammes de classes : associations - cardinalités

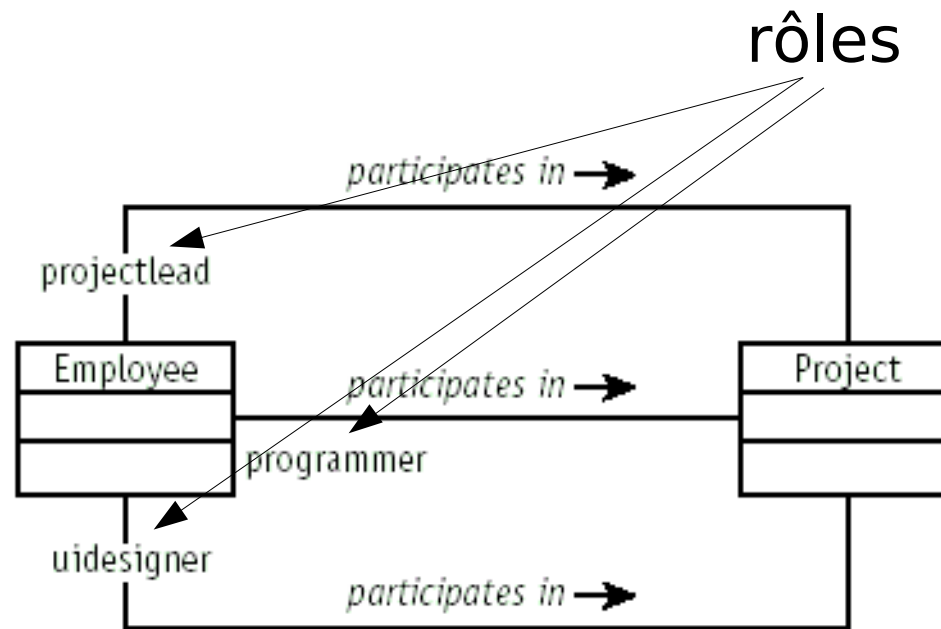
- Cardinalités

- Exactement un : 1
- Exactement n : n
- 0 ou plus : *
- 0 ou 1 : 0..1
- 1 ou plus : 1..*
- Spécifiée : 1..3



- Une voiture est conduite pas une et une seule personne.
- Une personne peut ne pas conduire de voiture ou bien conduire plusieurs voitures.

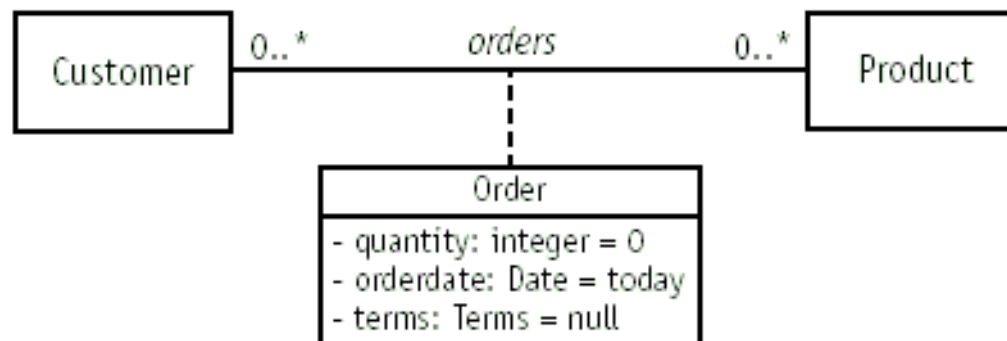
Diagrammes de classes : associations - rôles



- Le nom d'un rôle génère du code alors que le nom d'une association ne génère pas de code.

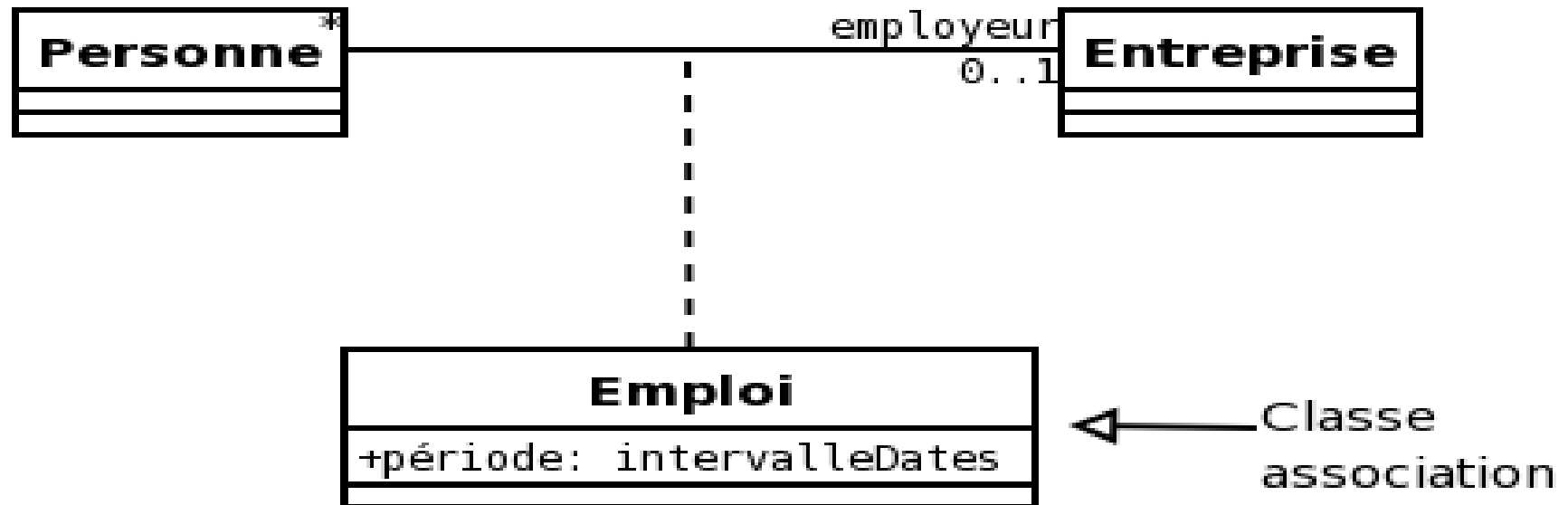
Diagrammes de classes : classes d'associations

- Une classe d'association encapsule des informations sur l'association. Très souvent lors d'une association "many-to-many".

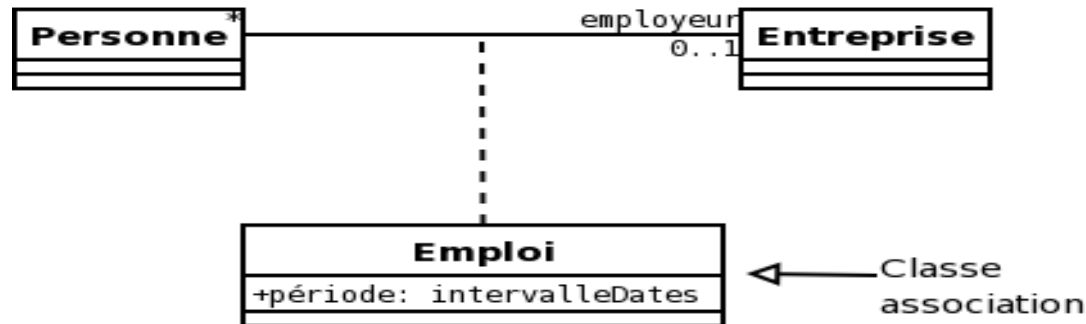


Diagrammes de classes : classes d'associations (2)

- Une classe d'association permet donc d'ajouter des attributs, des opérations et d'autres fonctionnalités aux associations.



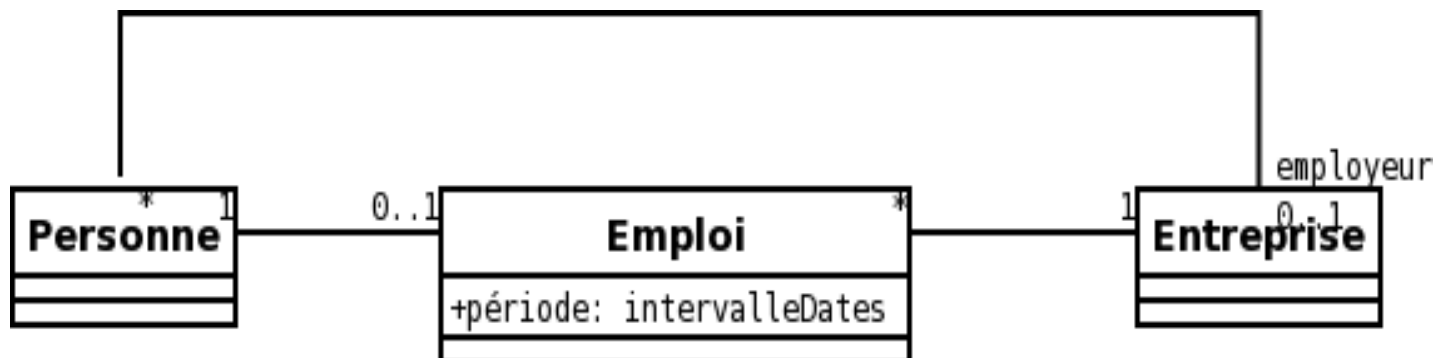
Diagrammes de classes : classes d'associations (3)



- Une personne ne peut travailler que pour une seule entreprise. Nous pouvons mémoriser la période durant laquelle chaque employé travaille pour chaque entreprise.

Diagrammes de classes : classes d'associations (4)

- Le diagramme précédent ne permet pas à une Personne d'avoir plus d'un emploi dans la même entreprise.
- Si l'on veut que cela soit possible, il faut dessiner le diagramme suivant :



Diagrammes de classes : associations réflexives

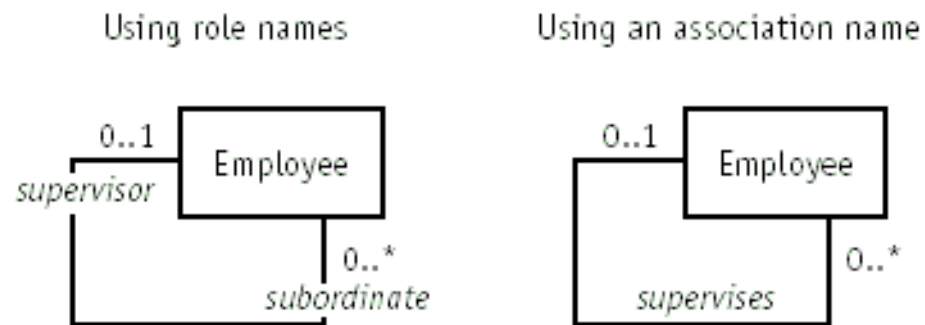
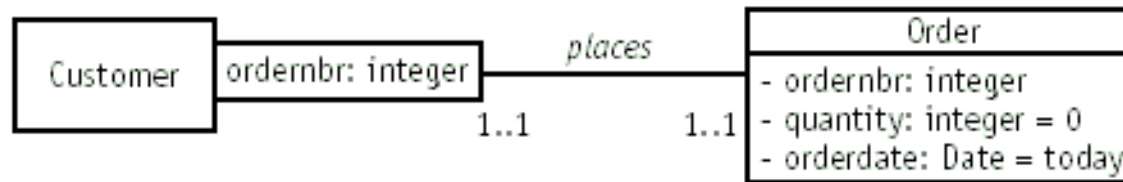


Figure 10-7 Two ways to model a reflexive association

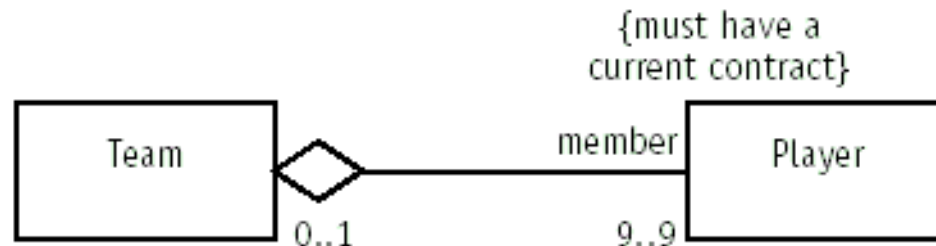
Diagrammes de classes : associations qualifiées



- Le client exploite *ordernbr* pour accéder à une commande.
- On exploite les qualifieurs pour réduire la multiplicité des accès (idem index pour les BD).

Agrégation

- Une forme spécifique d'association.
- Une association présente toutes les propriétés d'une association, plus quelques notions supplémentaires.
- Une agrégation (représentation par un diamant vide) va relier 2 classes : l'agrégat est composé d'un composant. Exemple : une équipe est composée de joueurs

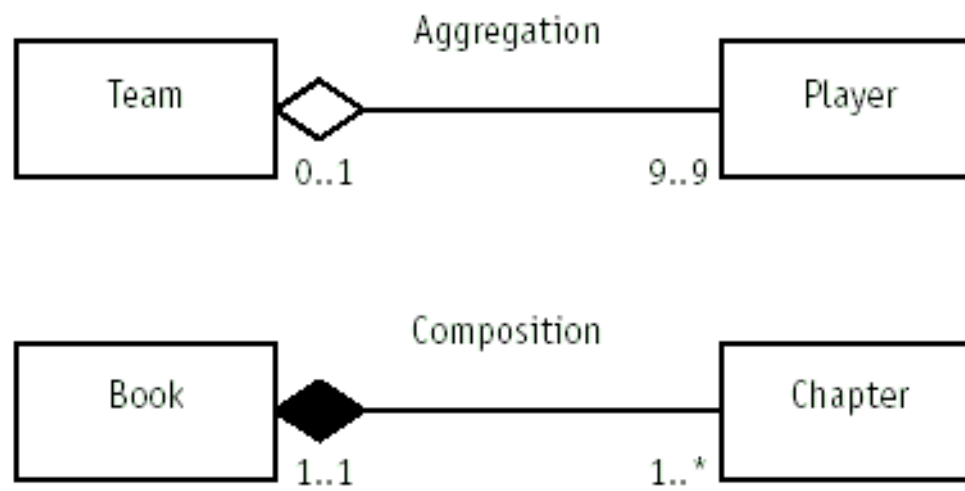


Agrégation

- Une agrégation va définir un point de contrôle unique depuis l'objet agrégat (Equipe).
Exemple : un moteur est le contrôleur de ses composants mais la voiture est le contrôleur du moteur.
- Lorsqu'une instruction doit modifier un ensemble d'objets (les membres de l'agrégation), l'agrégat va dicter comment les composants doivent réagir. Exemple : on appuyant sur la pédale d'accélération, l'ensemble de composants (concernés) de la voiture vont agir pour une accélération.

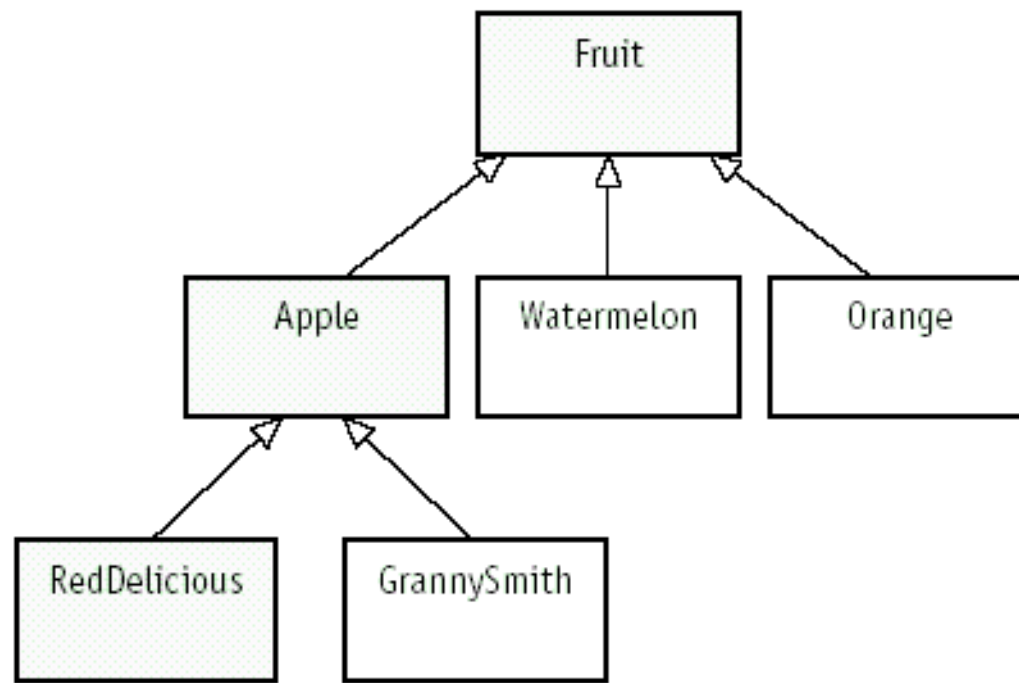
Composition

- Une variété d'agrégation plus forte : l'objet "partie" ne peut appartenir qu'à un seul tout.
- On suppose que les parties sont créées et meurent avec le tout.



Généralisation - spécialisation

- Représentation de hiérarchies de classes à l'aide d'une flèche.



Les stéréotypes

- Un *stéréotype* élargit le vocabulaire d'UML, en permettant de créer de nouvelles sortes de briques de base qui dérivent des sortes existantes mais qui sont adaptées à un problème donné. Par exemple, lors d'un travail dans un langage de programmation, tel que Java ou C++, il est souvent souhaitable de modéliser des exceptions. Dans ces langages, les exceptions ne sont que des classes, même si elles sont traitées de manière très particulière. En règle générale, il est préférable de leur permettre simplement d'être lancées et attrapées et rien de plus. Il est possible de faire de ces exceptions des éléments plus importants des modèles — ce qui signifie qu'elles sont traitées comme des briques de base — en les marquant avec un stéréotype approprié.
- Définition :
stéréotype. Un nouveau type d'élément de modélisation qui étend la sémantique du méta-modèle. Les stéréotypes doivent être basés sur des types ou classes existant dans le méta-modèle.

Diagramme d'objet

Diagramme objets

- Modélise les instances de classes.
- Utilisé pour décrire le système à un instant précis dans le temps.
- On peut considérer que c'est une instance du diagramme de classes.
- Syntaxe :
 - Le nom d'un objet est souligné (nom:Classe, nom, :Classe)

Diagramme objets (2)

- Un lien est une instance d'une association. C'est un tuple de références vers des objets.

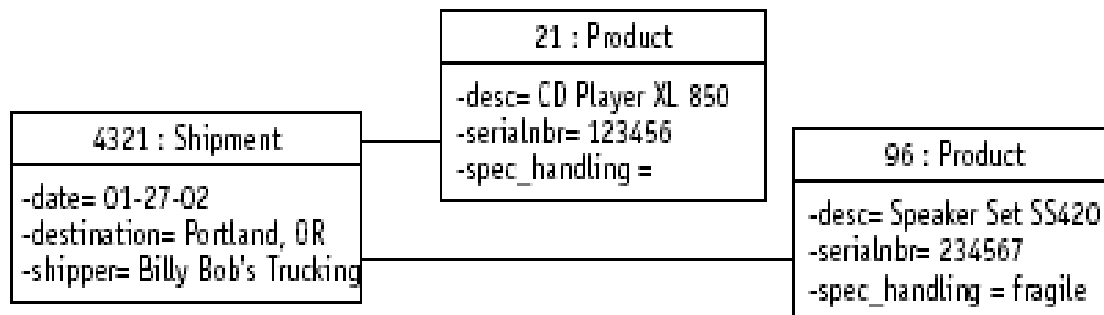
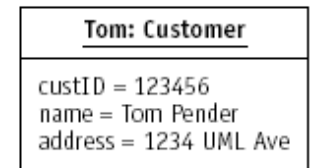


Diagramme de cas d'utilisation

Cas d'utilisation

- Un cas d'utilisation est un ensemble de scénarios reliés par un objectif commun, celui de l'utilisateur.
- Exemple d'un site de e-commerce :
 - *Le client parcourt le catalogue et ajoute les articles sélectionnés à son panier électronique. Quand il veut payer, il fournit les informations sur la livraison et sur sa carte de crédit et valide l'achat. Le système vérifie que la carte de crédit est autorisée, et confirme la vente par un e-mail.*
- Si l'autorisation est refusée, un autre scénario se présente.

Cas d'utilisation (2)

- En général, on va décrire le scénario sous forme d'étapes numérotées ainsi que les variantes de la cette séquence.
- Exemple :
 1. Le client parcourt...
 2. Le client valide son choix ...
 - ..
 - n. Le système valide la transaction.

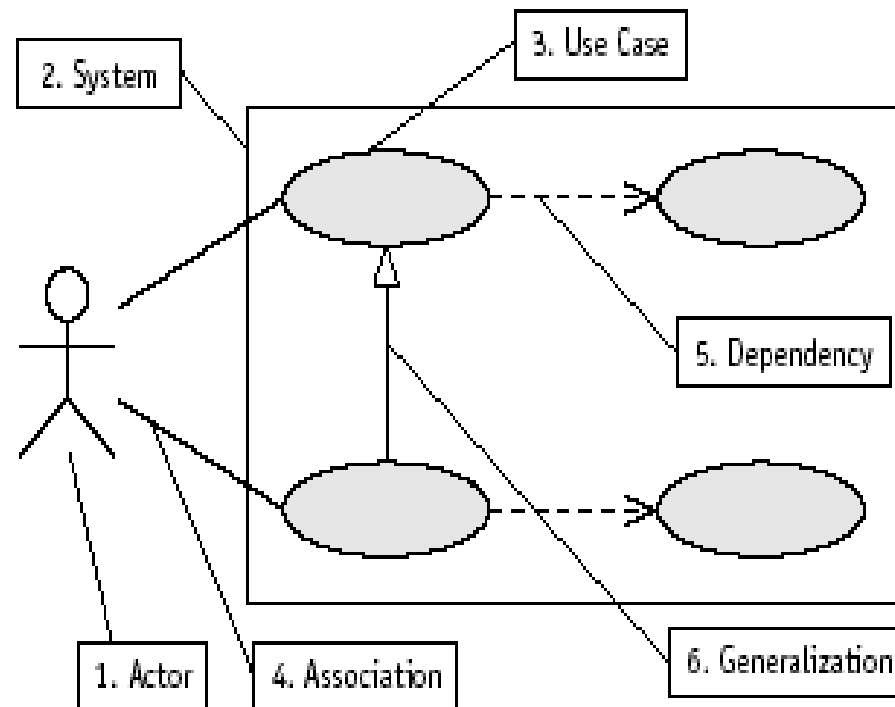
Diagramme de cas d'utilisation (use case)

- Modélise la fonctionnalité proposée par le système (cas d'utilisation), les utilisateurs qui interagissent avec le système (acteurs) et les associations entre les utilisateurs et la fonctionnalité.
- Un cas d'utilisation permet de décrire **ce que** le futur système devra faire, sans spécifier **comment** il le fera.

Diagramme de cas d'utilisation (2)

- Représente un ensemble de séquences d'actions qui sont réalisées par le système et qui produisent un résultat observable intéressant pour un acteur particulier.
- Acteur : un rôle joué par une entité externe interagissant directement avec le système. C'est une sorte de classe particulière. Ce n'est pas nécessairement un humain.

Les éléments d'un diagramme de cas d'utilisation



Les éléments d'un diagramme de cas d'utilisation (2)

- Acteur : Le rôle joué par une personne, système ou matériel dans le fonctionnement d'un système.
- Cas d'utilisation : identification d'un élément prépondérant du système.

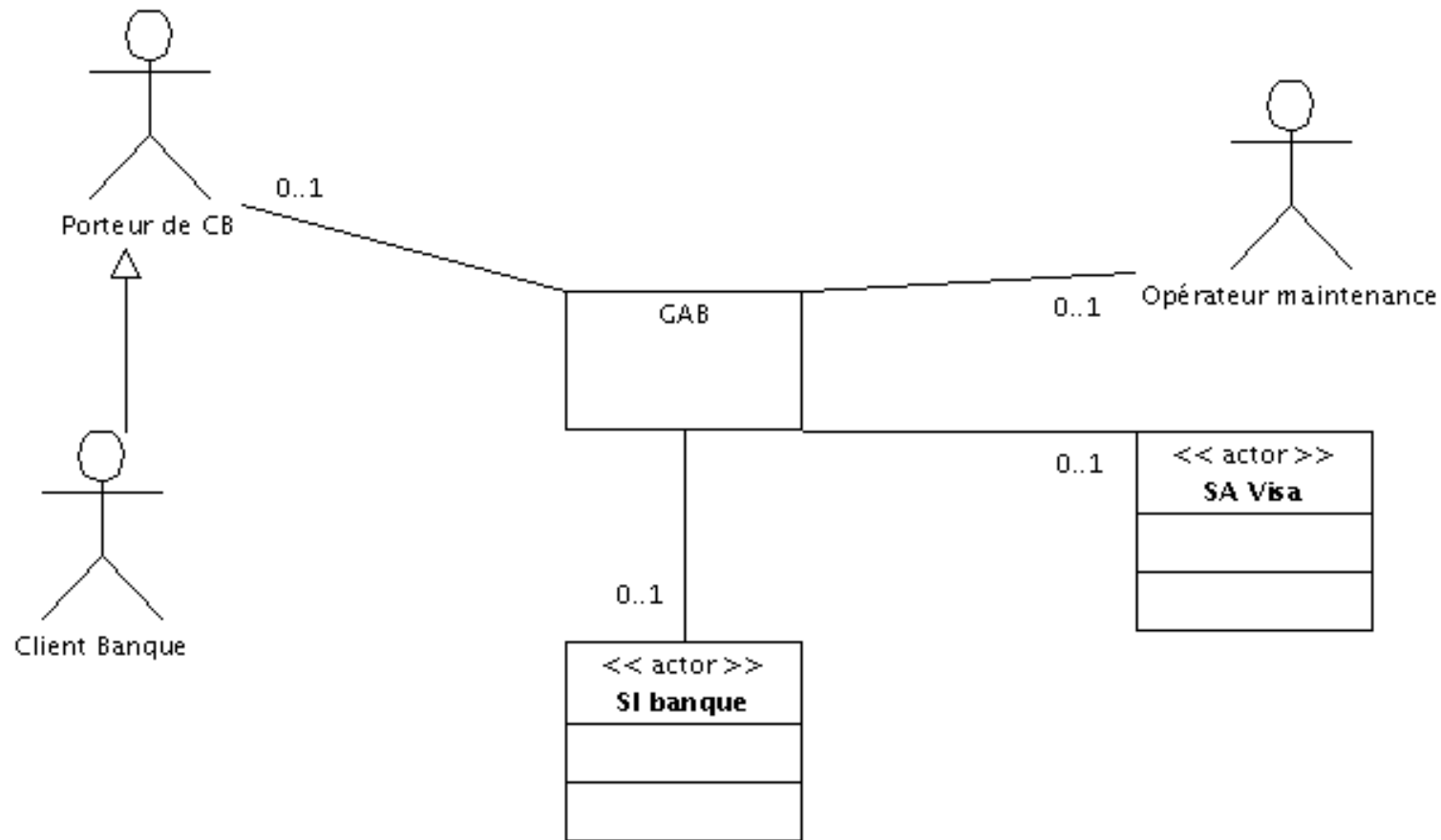
Exemple

- Système simplifié de Guichet automatique de banque (GAB) qui offre les services suivants :
 - Distributeur d'argent à tout porteur d'une carte de crédit (Visa ou carte de la banque) via un lecteur de carte et un distributeur de billets.
 - Consultation de solde de compte, dépôt de numéraire et dépôt de chèques pour les clients de la banque porteurs d'une carte de crédit de la banque.
 - Toutes les transactions sont sécurisées. Le système d'autorisation (SA) Visa, pour les transactions de retrait effectuées avec une carte Visa. Un système d'information (SI) de la banque pour autoriser toutes les transactions effectuées par un client avec sa carte de banque, mais également pour accéder au solde de comptes.
 - Il est parfois nécessaire de recharger le distributeur, etc..

Les acteurs de l'exemple

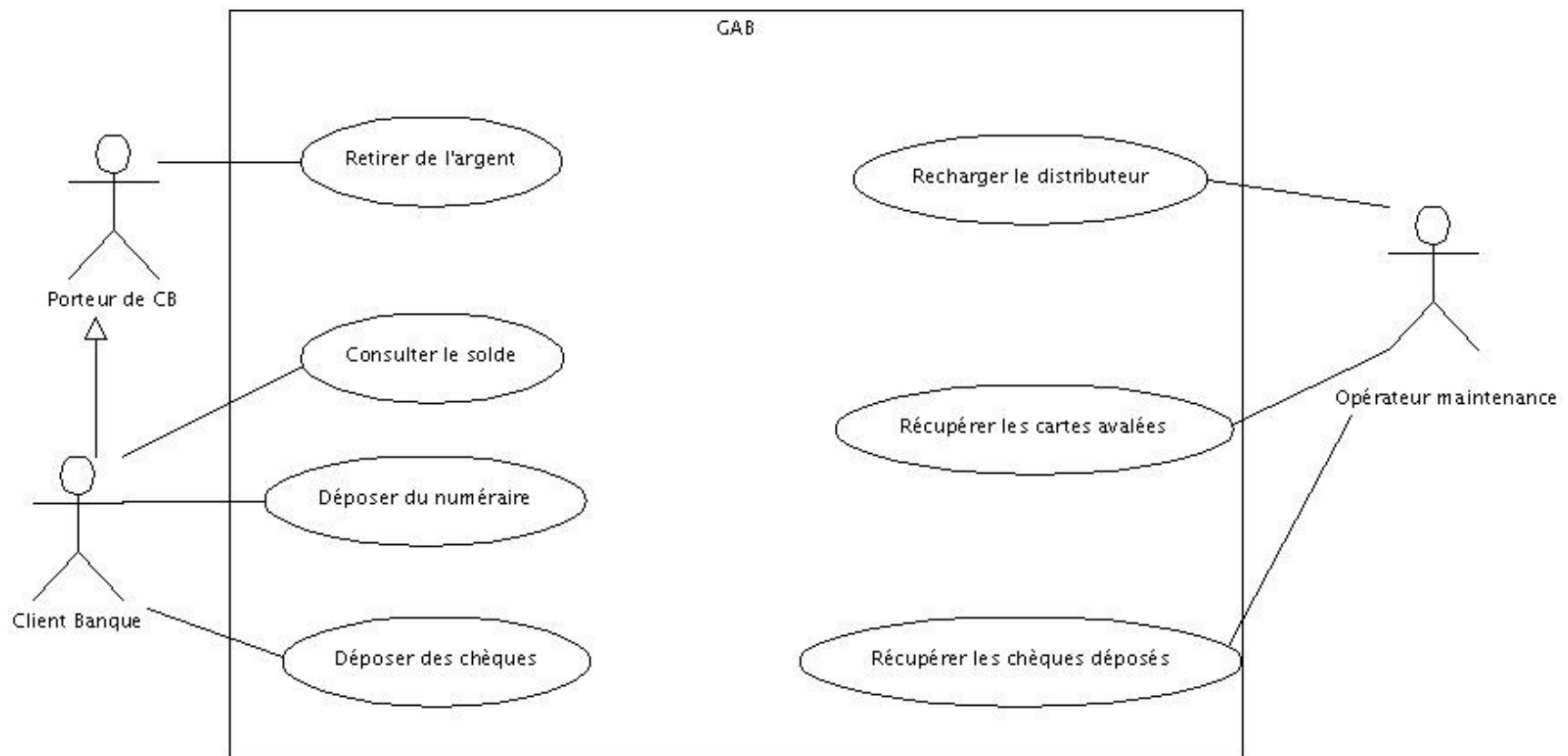
- Acteur 1 : « porteur d'une carte de crédit ».
- Acteur 2 : « Client banque »
- Acteur 3 : opérateur maintenance
- Acteur 4 : SA Visa
- Acteur 5 : SI banque

Diagramme de contexte statique de GAB



Created with Poseidon for UML Community Edition. Not for Commercial Use.

Diagramme de cas d'utilisation de GAB



Created with Poseidon for UML Community Edition. Not for Commercial Use.

Relations entre cas d'utilisation

- Include : existe quand un fragment de comportement est similaire dans plusieurs cas d'utilisation.
- Généralisation : lorsqu'un cas d'utilisation est semblable à un autre, mais a une fonctionnalité supplémentaire
- Extend : Semblable à la généralisation mais le cas qui "étend" peut ajouter un comportement au cas de base.

Cas d'utilisation métier et système

- Un cas d'utilisation système est une interaction avec le logiciel.
- Un cas d'utilisation métier aborde la manière dont un "métier" répond à un client ou à un événement.

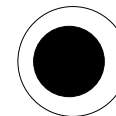
Diagramme d'activité

Diagrammes d'activité

- Les **diagrammes d'activités** sont un type particulier de diagrammes d'état, qui décrit la succession des activités au sein d'un système. Ils présentent la vue dynamique d'un système, sont particulièrement importants dans la modélisation de la fonction d'un système et mettent l'accent sur le flot de contrôle entre objets.
- Contexte d'utilisation : Description du comportement interne
 - D'une classe
 - D'une méthode
 - D'un cas d'utilisation



Etat initial



Etat final



Barre de
synchronisation



Diagrammes d'activité (2)

- Ils décrivent l'organisation des activités, supportant à la fois les comportements conditionnels et les comportements parallèles.
- Une condition repose sur la notion de branchement et de fusion.
 - Un branchement a une seule transition entrante et plusieurs sortantes (une seule peut être empruntée).

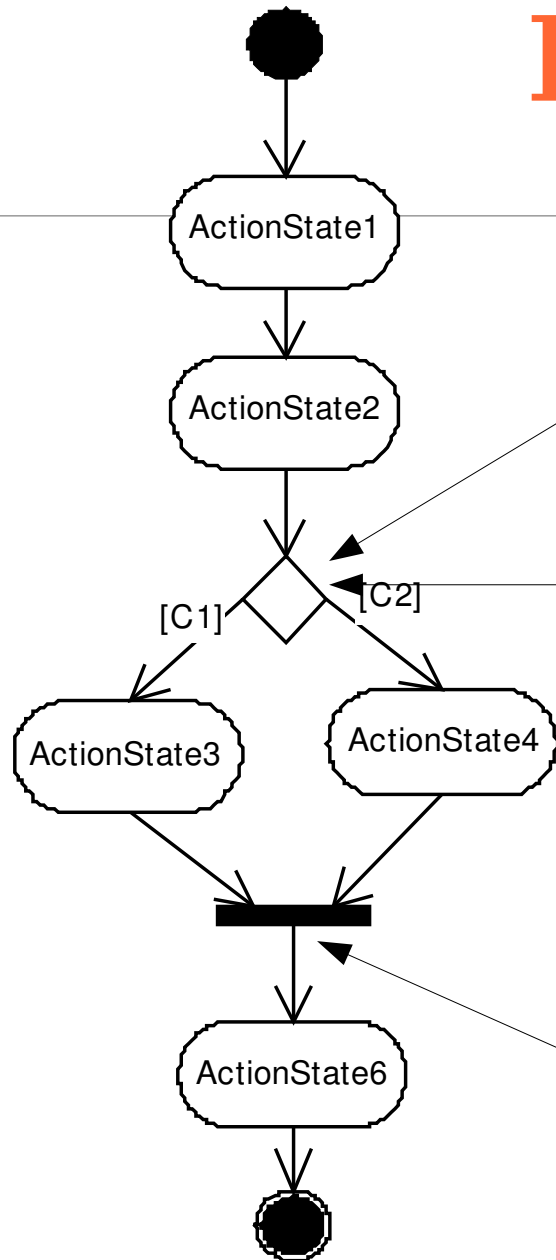
Diagrammes d'activité (3)

- Une fusion a plusieurs transitions entrantes et une seule transition sortante. Elle marque la fin d'un comportement conditionnel. Initialisé par un branchement.

Diagrammes d'activité (4)

- Le comportement parallèle est décrit par les débranchements (fork) et les jonctions (join) :
 - Débranchement : 1 transition entrante et plusieurs sortantes. A partir de cette barre de synchronisation, toutes les transitions sortantes sont empruntées en parallèles.
 - A tout débranchement, correspond une jonction. Avec une jonction, la transition sortante est empruntée que lorsque toutes les activités entrantes sont terminées.

Exemple

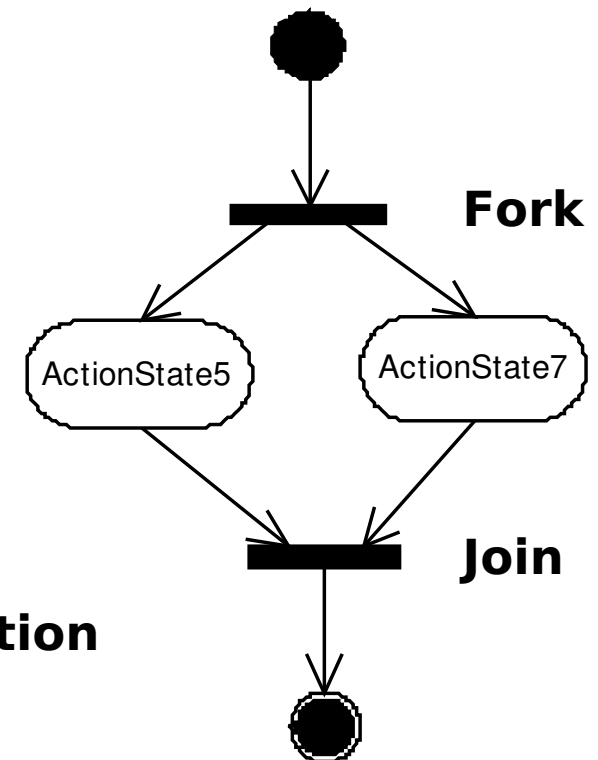


branchement

Condition

Barre de synchronisation

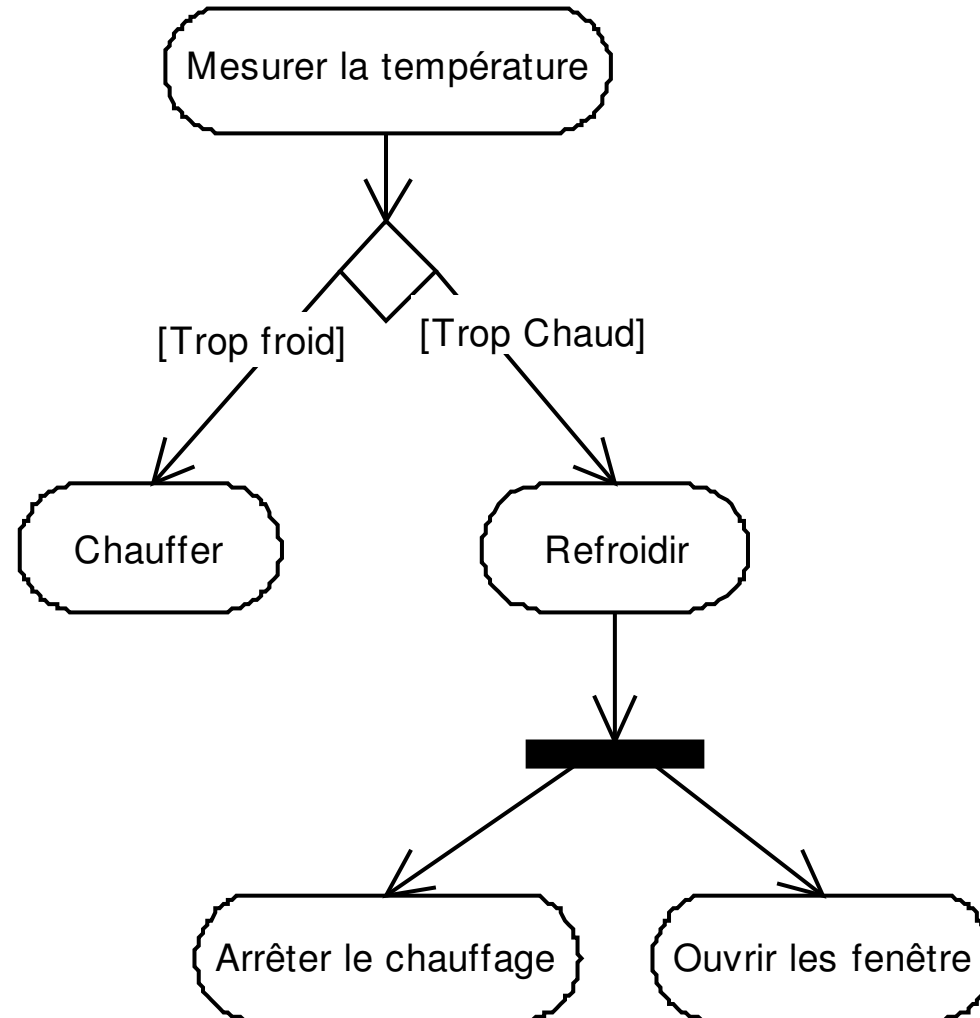
fusion



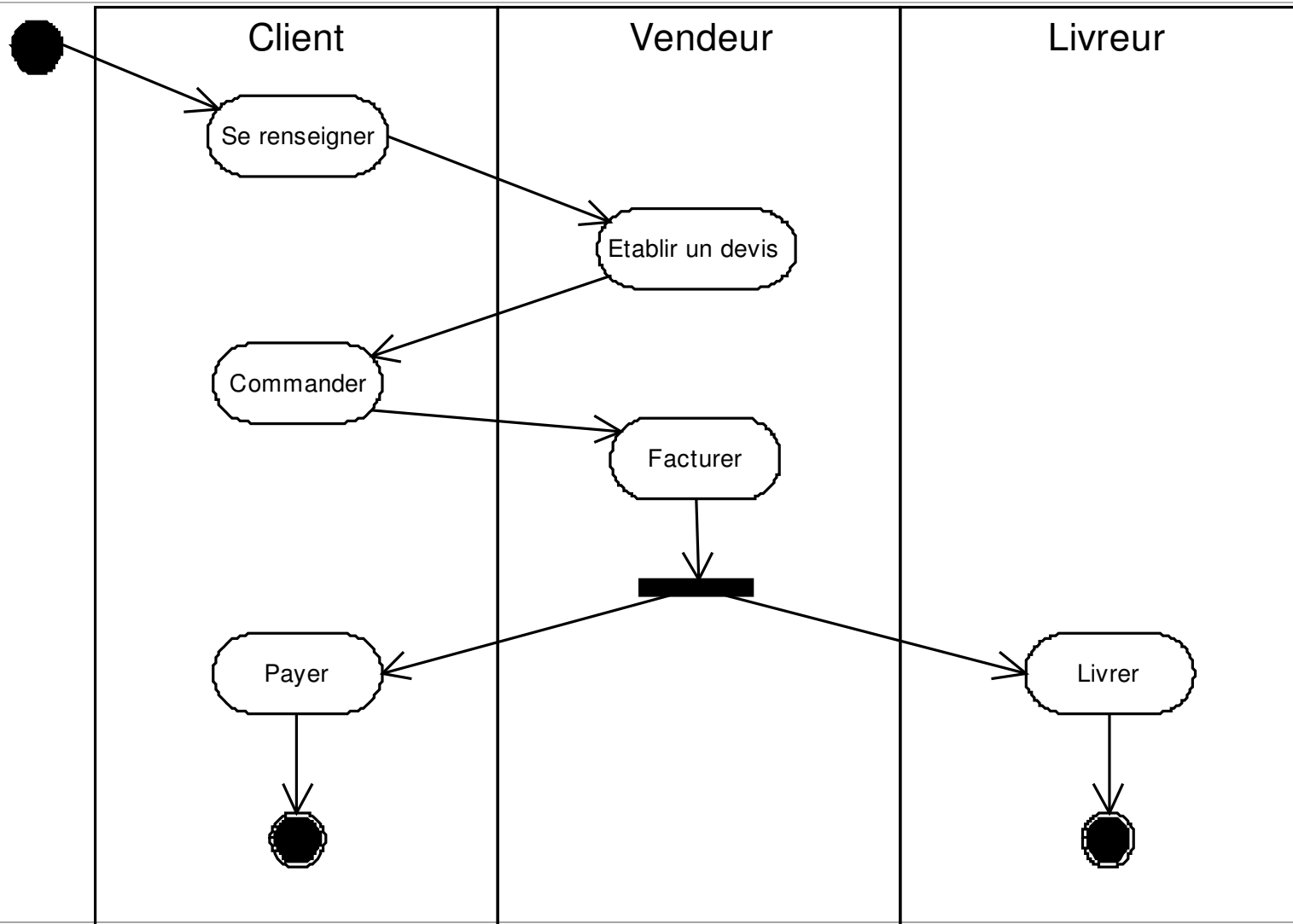
Fork

Join

Exemple 2



Exemple 3



Conclusion sur les diagrammes d'activité

- Ces diagrammes sont particulièrement utiles en liaison avec les workflows et dans les descriptions de comportements massivement parallèles.

Vue dynamique

- Cette vue permet de modéliser les interactions entre objets.
- Elle regroupe les diagrammes de :
 - Séquences
 - Collaboration
 - “Statechart” qui propose une vision des actions d'un objet en réaction aux stimuli externes.

Diagramme de séquences

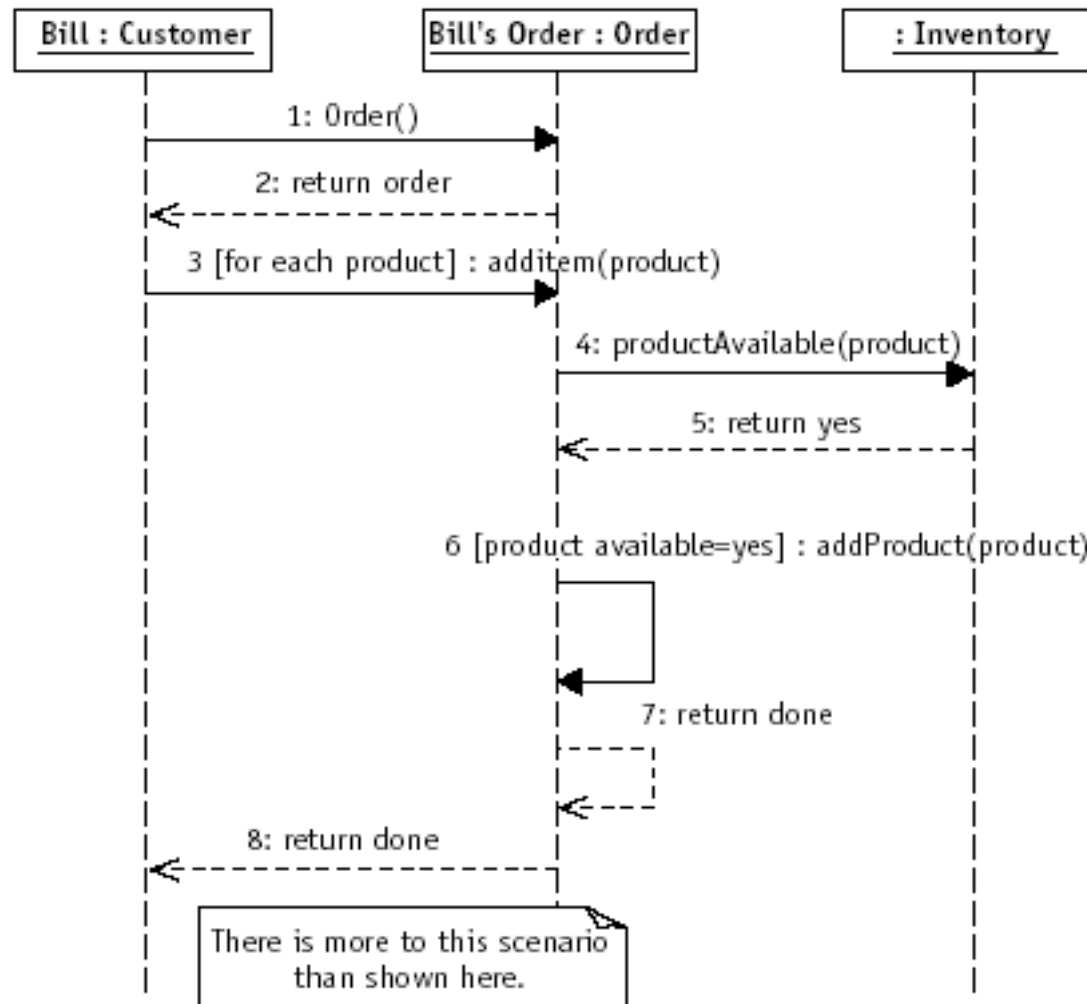
Diagramme de séquences

- Représentation des interactions entre acteurs et objets.
- Vision temporelle d'une interaction
 - Chaque objet est symbolisé par une barre verticale.
 - Le temps s'écoule de haut en bas, de sorte que la numérotation des messages est optionnelle.

Diagramme de séquences (2)

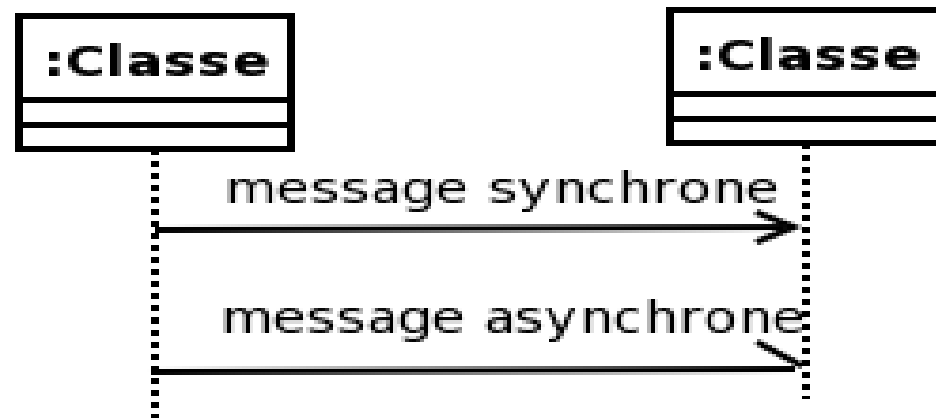
- Met en évidence les objets et les messages qui sont passés entre ces objets dans le cas d'utilisation. Souvent utilisé pour représenter une instance d'un cas d'utilisation.
- Complémentaire du diagramme de collaboration.

Exemple de diagramme de séquences



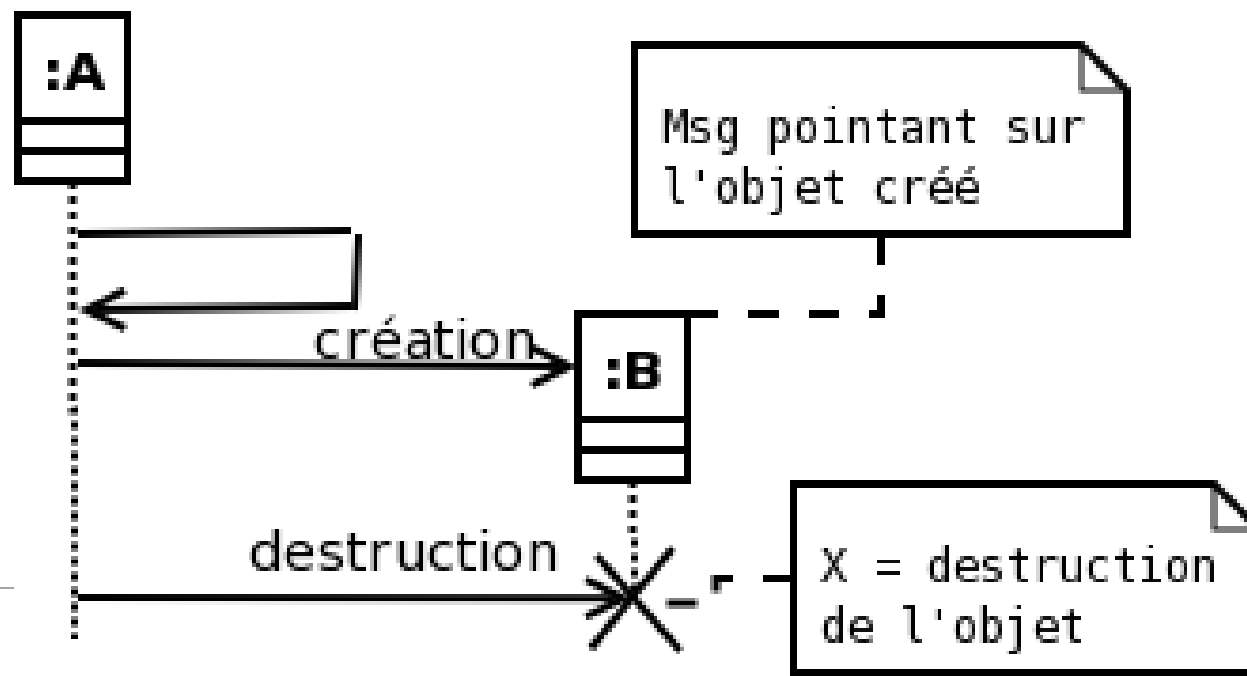
Synchronisation de messages

- On distingue :
 - Les messages synchrones : l'émetteur est bloqué jusqu'au traitement effectif du message.
 - Aux messages asynchrones : l'émetteur n'est pas bloqué et il peut donc poursuivre son exécution.

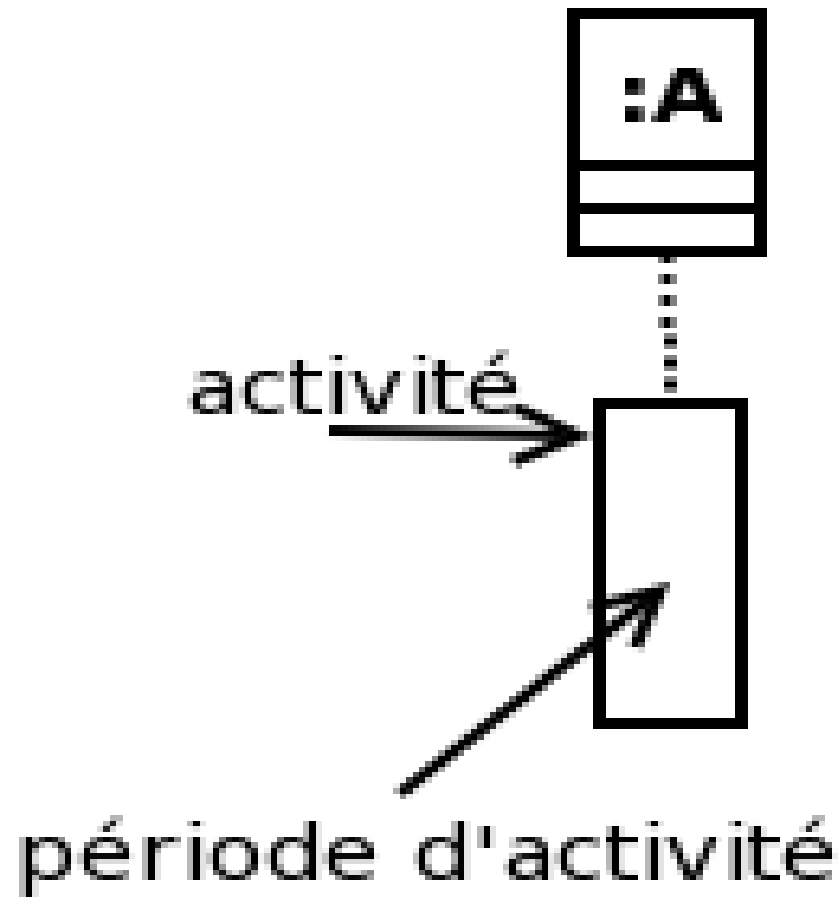


Messages

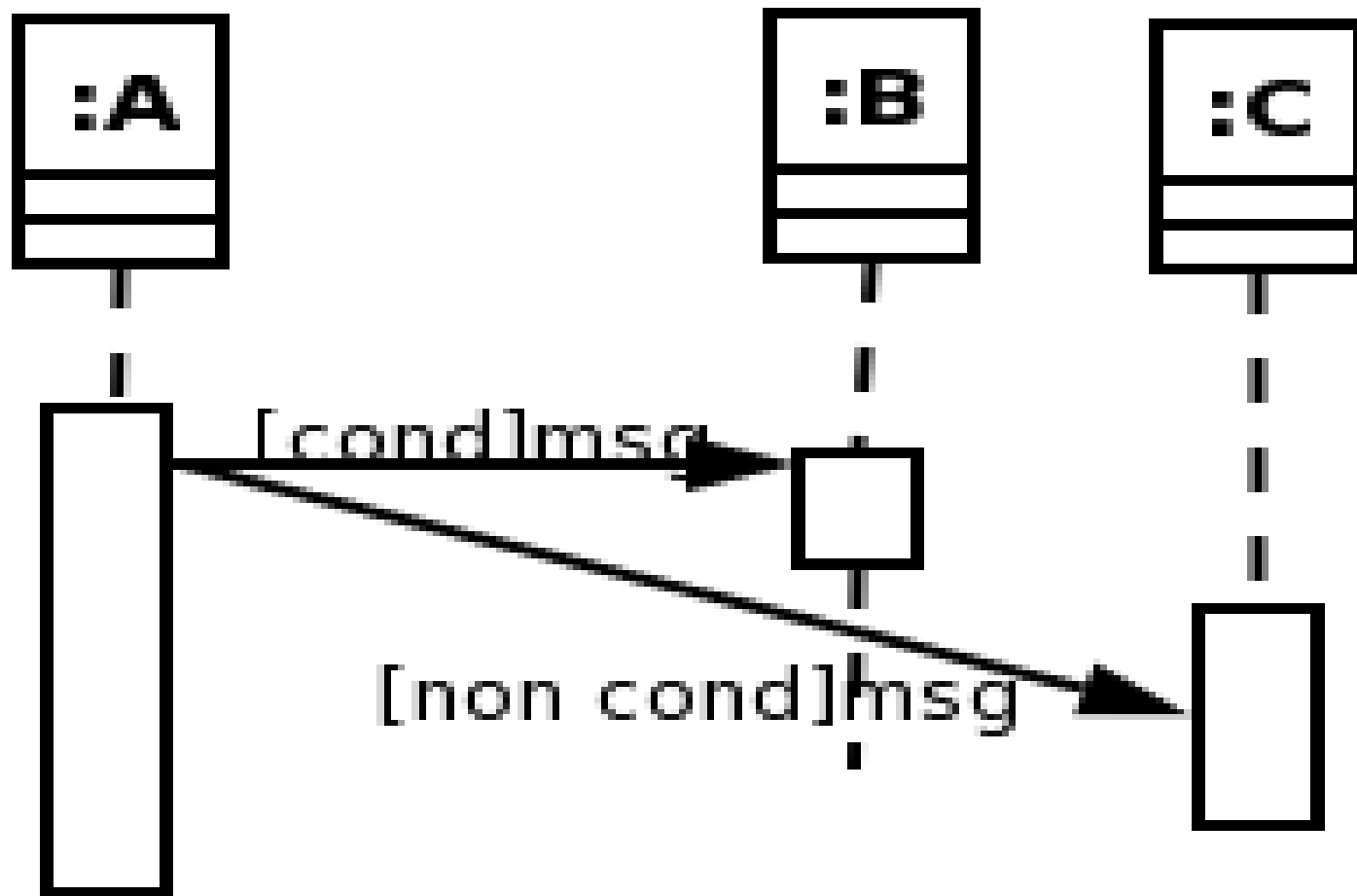
- Un objet peut s'envoyer à message à lui-même.
- On peut créer et détruire un objet à partir d'un message.



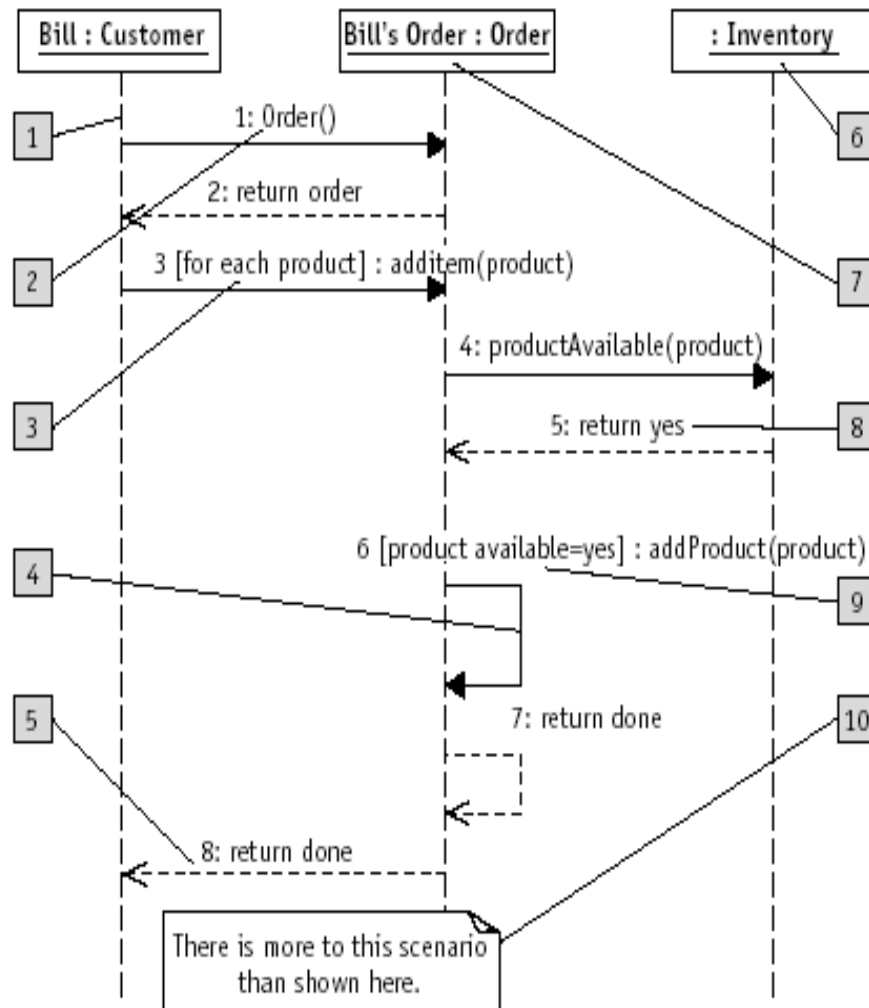
Période d'activité



Envoi conditionnel de message



Exemple de diagramme de séquences (2)



1. Ligne de vie de l'objet
2. Message / Stimulis
3. Iteration
4. Auto-référence
5. Retour
6. Objet anonyme
7. Objet nommé
8. Numéro de séquence
9. Condition
10. Commentaire

Stimuli et messages

- Stimulis :
 - Une communication entre deux instances, dans le but de déclencher une action.
- Message :
 - Une spécification de stimulus définissant les rôles de l'émetteur et récepteur.

Diagramme de collaboration

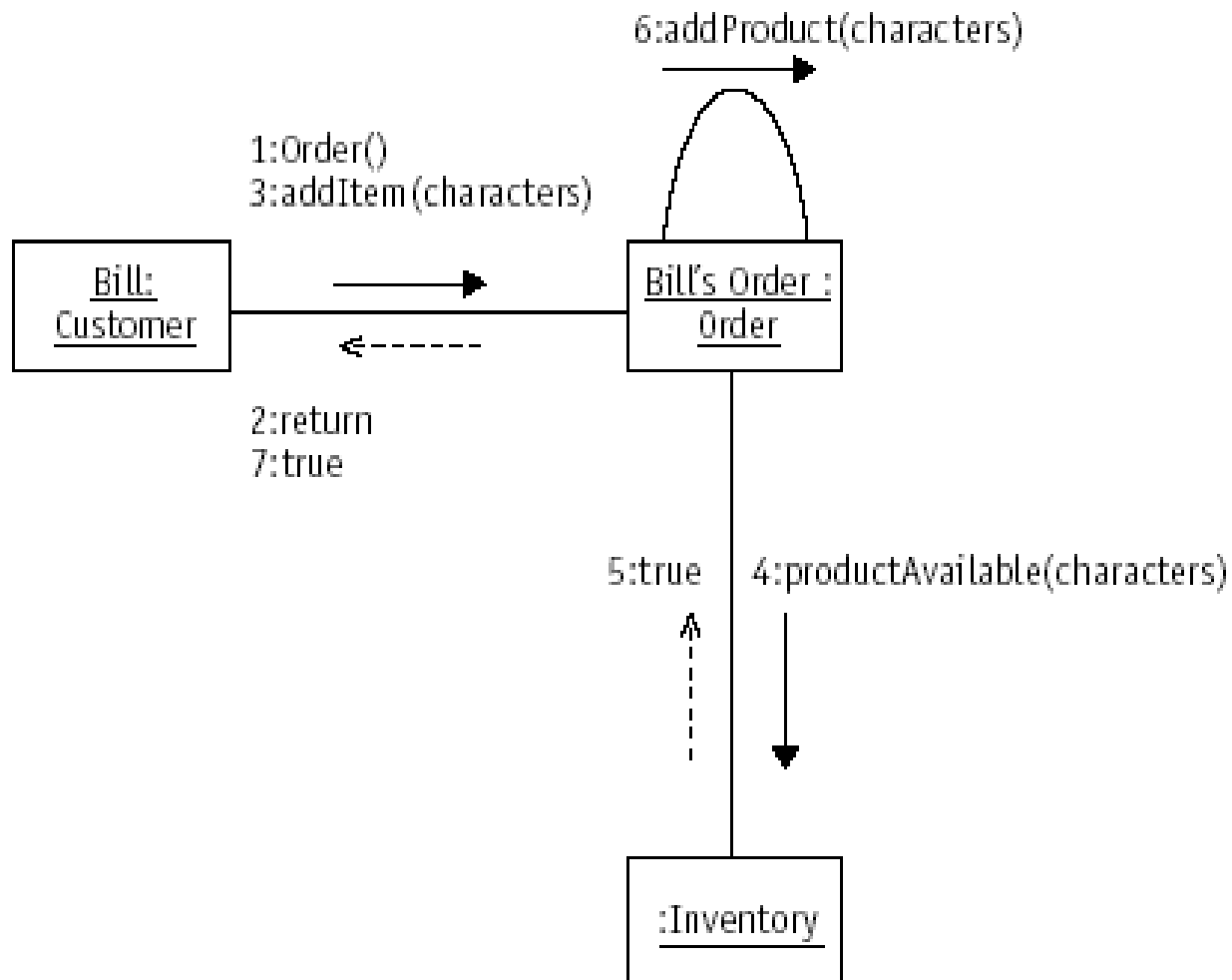
Diagramme de collaboration

- Extension du diagramme objet : vue dynamique
- Représentation d'une collaboration entre rôles
- Représentation spatiale d'une interaction
 - Mise en avant de la structure
 - Représentation des structures complexes (récurives par exemple).
- Pas d'axe temporel (diagramme dual du diagramme de séquences).

Diagramme de collaboration (2)

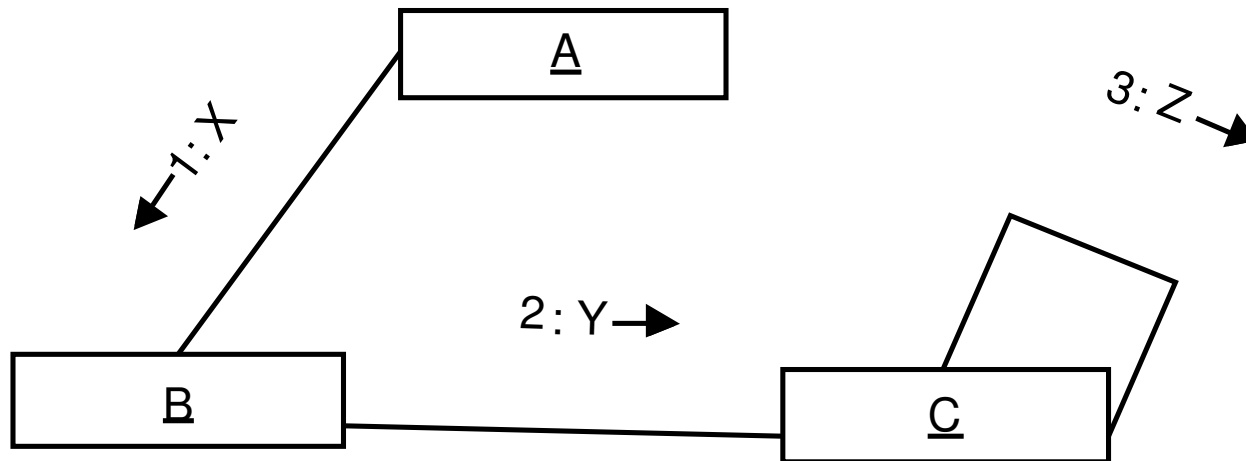
- Des liens relient des objets qui se connaissent
- Les messages échangés par les objets sont représentés le long de ces liens.
- L'ordre d'envoi des messages est matérialisé par un numéro de séquence.

Exemple



Exemple 2

- Un objet A envoie un stimulus X à un objet B, puis l'objet B envoie un stimulus Y à un objet C, et enfin C s'envoie un stimulus Z.



Bilan

- Le diagramme de collaboration modélise les mêmes informations que le diagramme de séquences : les interactions entre objets.
- Mais l'approche est différente :
 - Le diagramme de collaboration voit les interactions au niveau de la structure des objets et des relations inter-objets.
 - Le diagramme de séquences se concentre sur l'aspect temporel

Bilan (2)

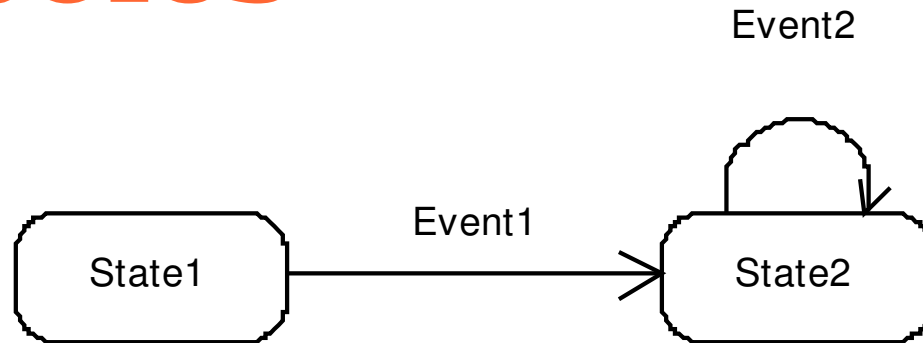
- Le diagramme de collaboration permet donc de valider et/ou découvrir de nouvelles associations entre classes.

Diagramme d'état

Diagramme d'état

- Les *diagrammes d'état* sont des automates à états, composés d'états, de transitions, d'événements et d'activités. Ils présentent la vue dynamique d'un système, sont particulièrement importants dans la modélisation du comportement d'une interface, d'une classe ou d'une collaboration et mettent l'accent sur le comportement d'un objet ordonnancé par les événements, ce qui est particulièrement utile dans la modélisation des systèmes réactifs.
- Etat
 - Condition dans laquelle se trouve un objet
- Transition
 - Chemin entre deux états
- Événement
 - Occurrence qui survient dans le domaine

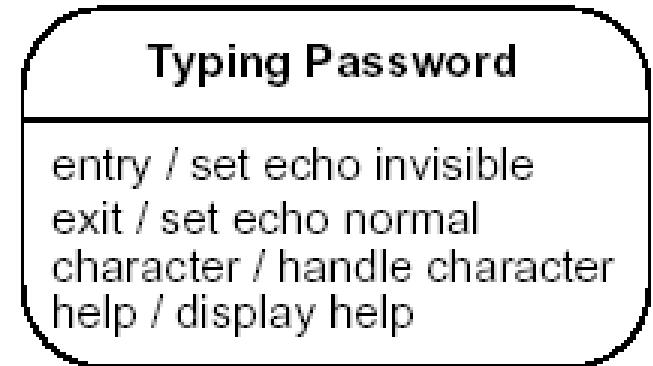
Symboles



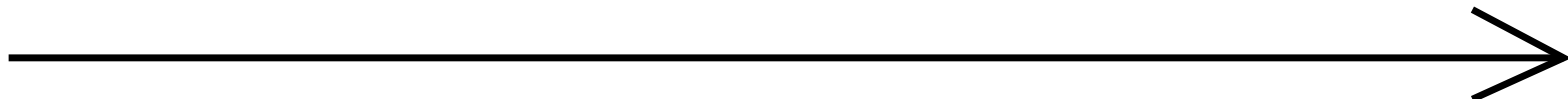
Pseudo Etat Initial



Pseudo Etat Final

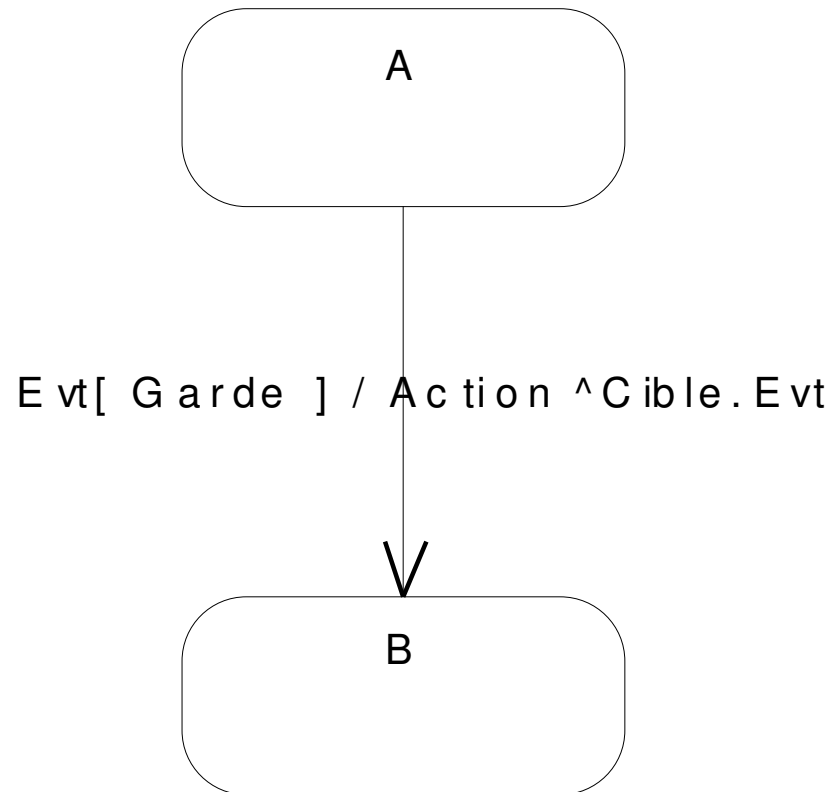
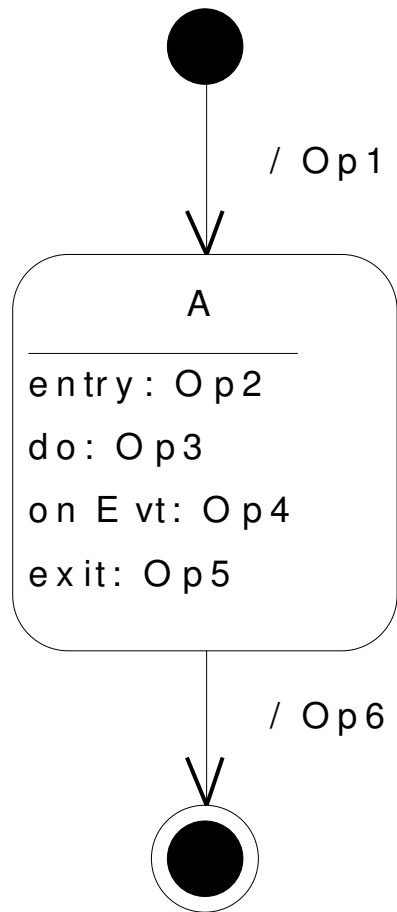


Event [C1] / Action

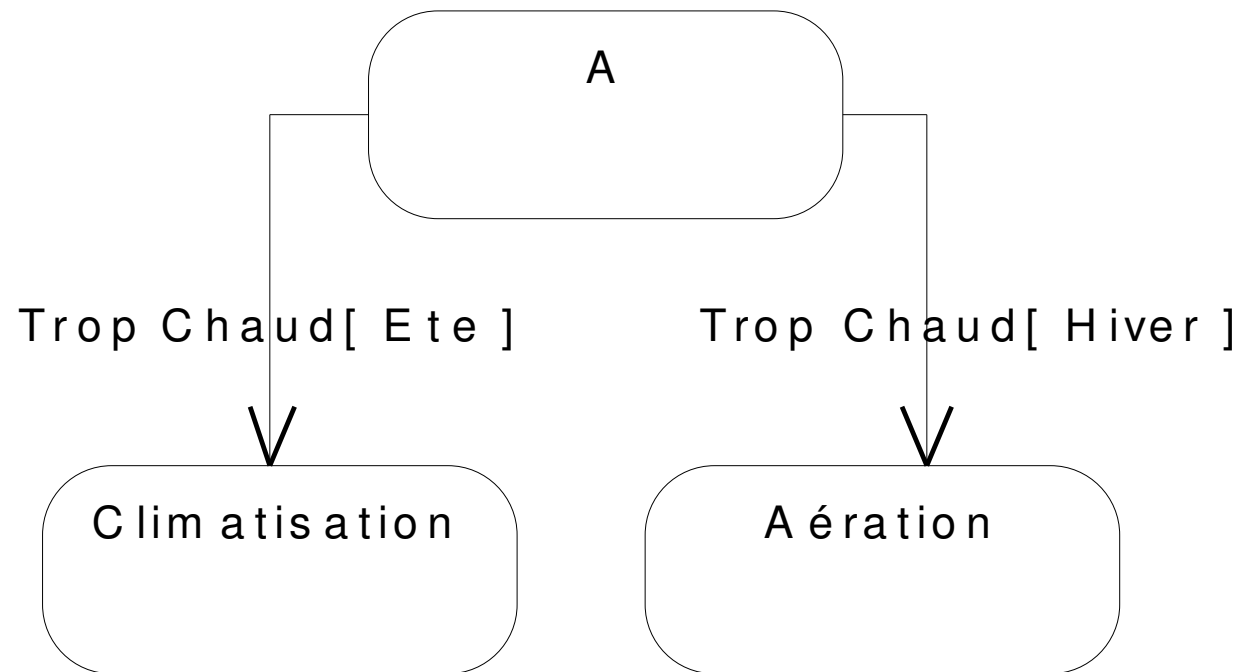


L'action est exécutée instantanément

Les actions et les activités

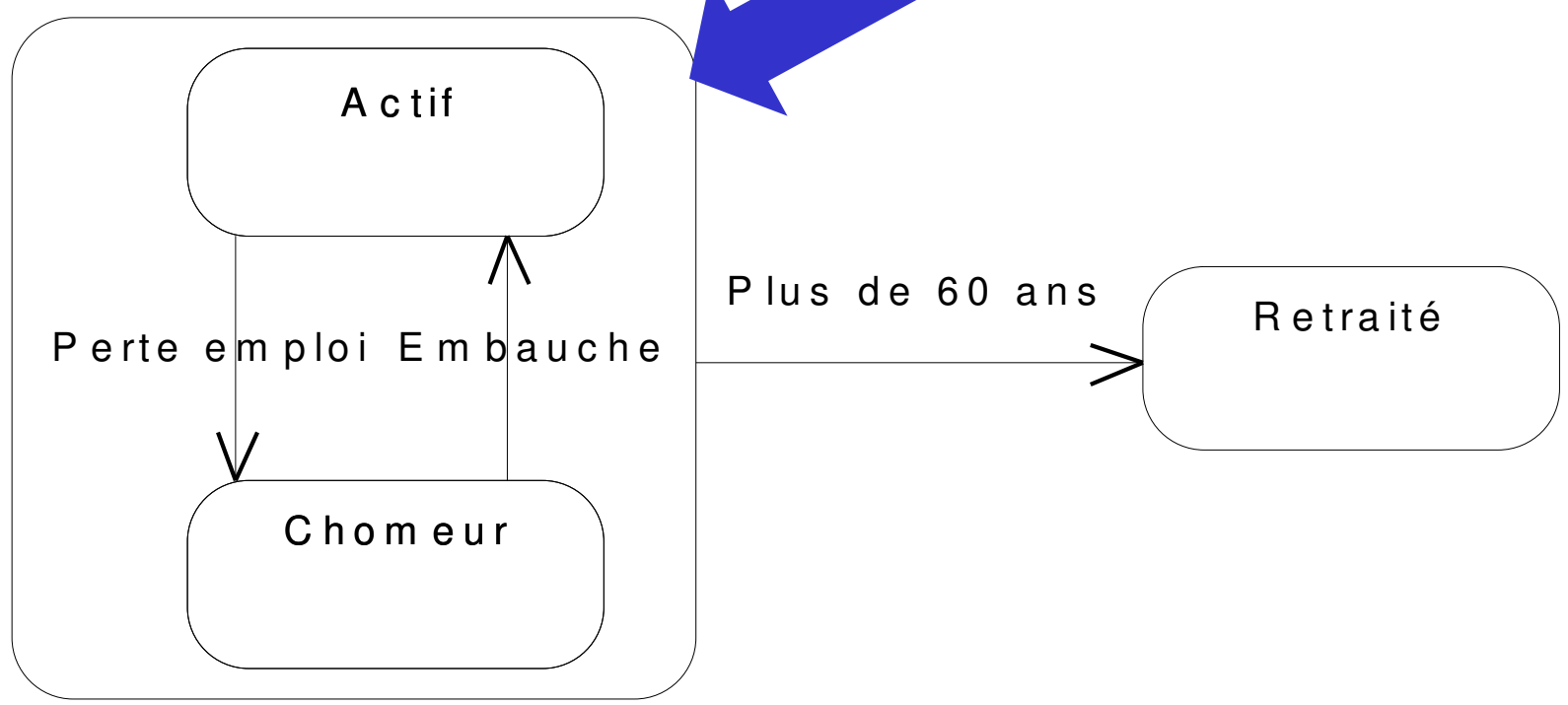
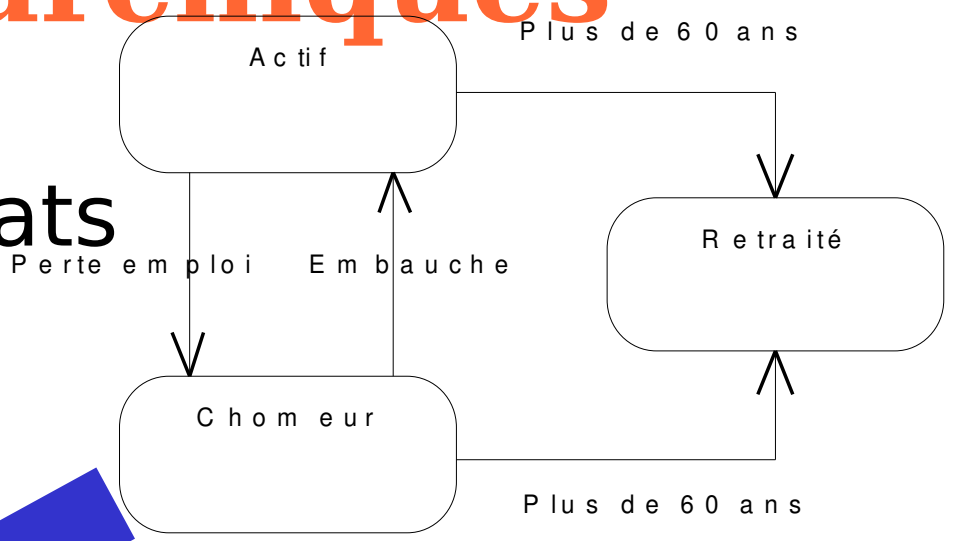


Les gardes

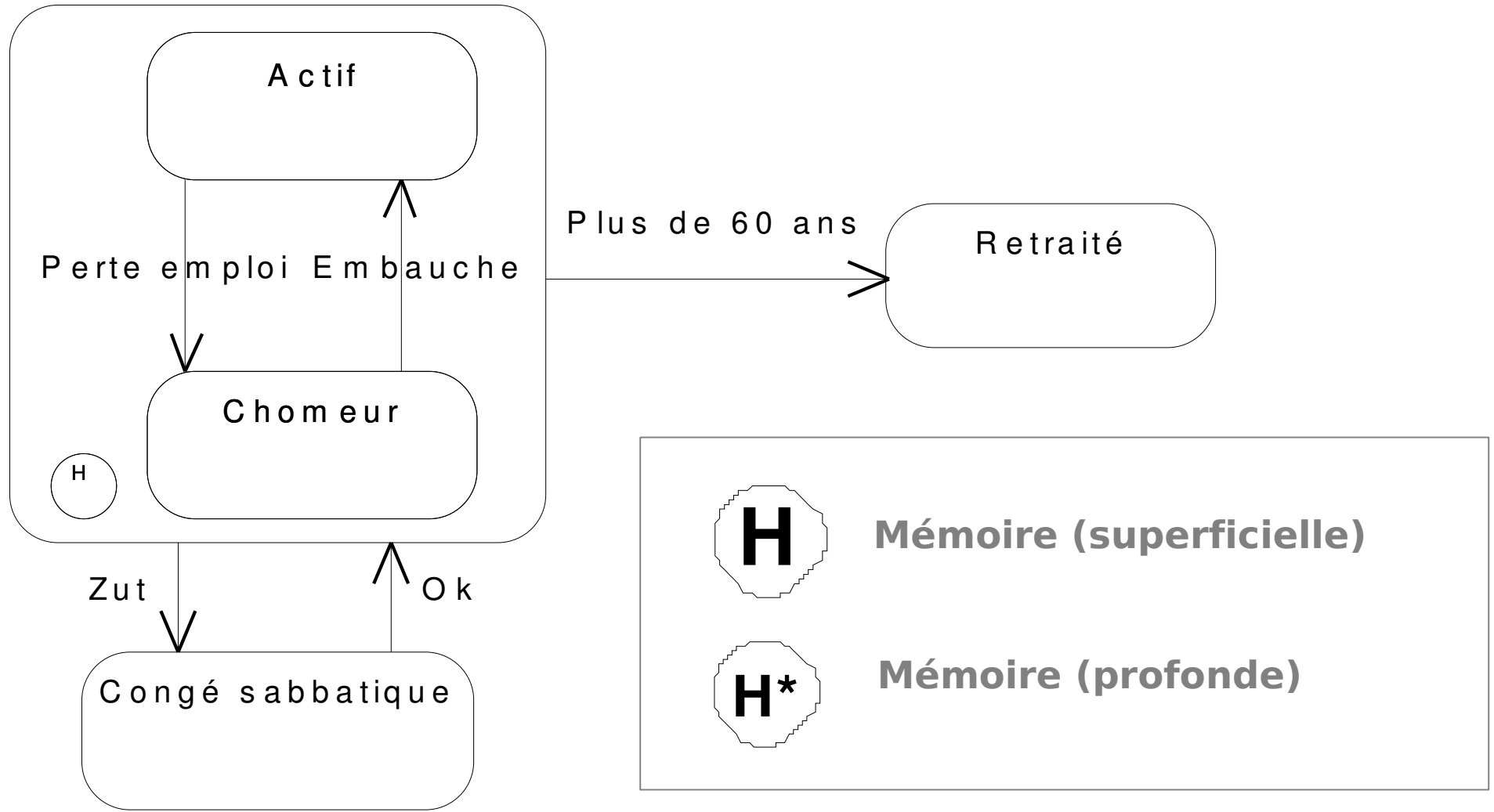


Automates hiérarchiques

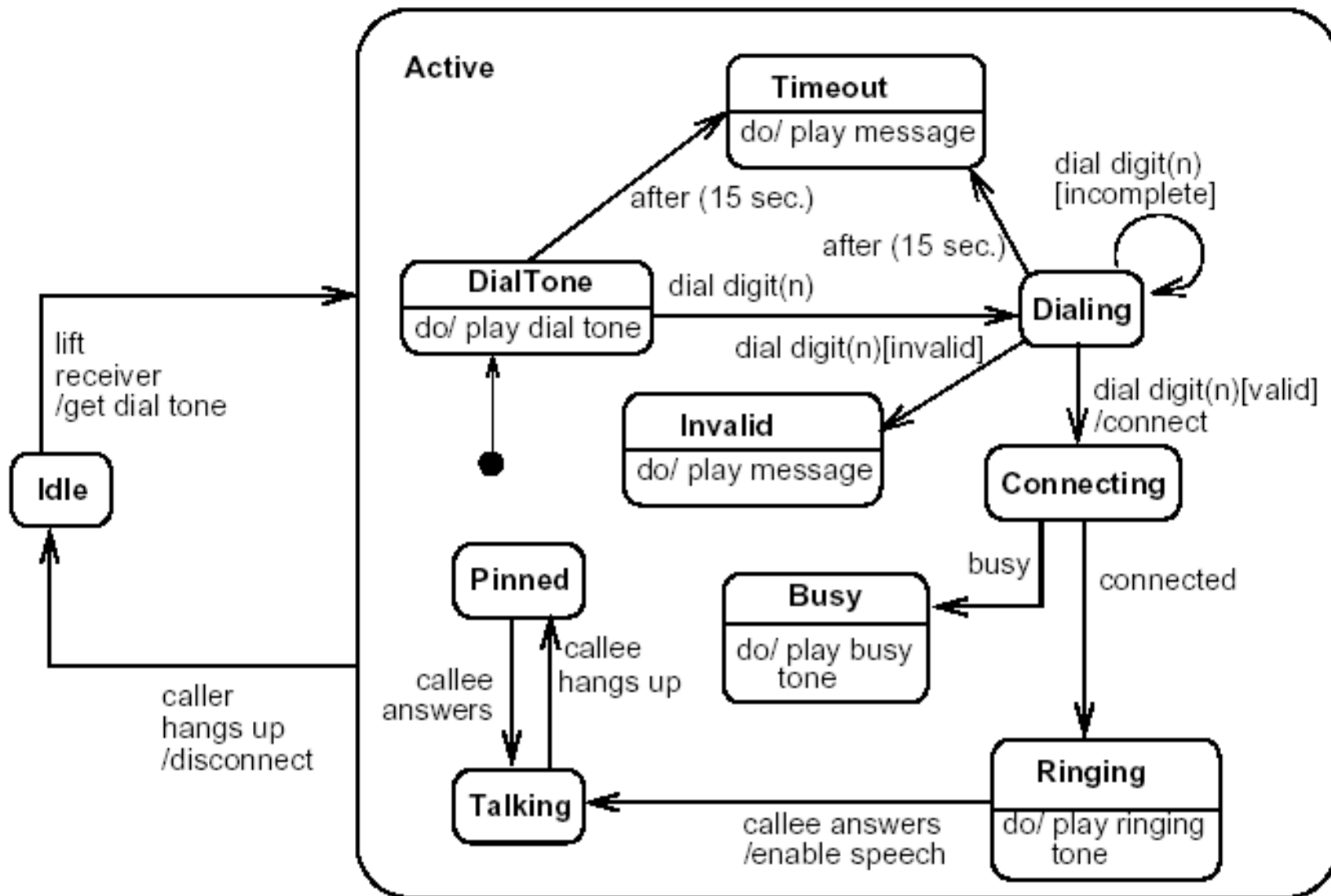
- Généralisation d'états



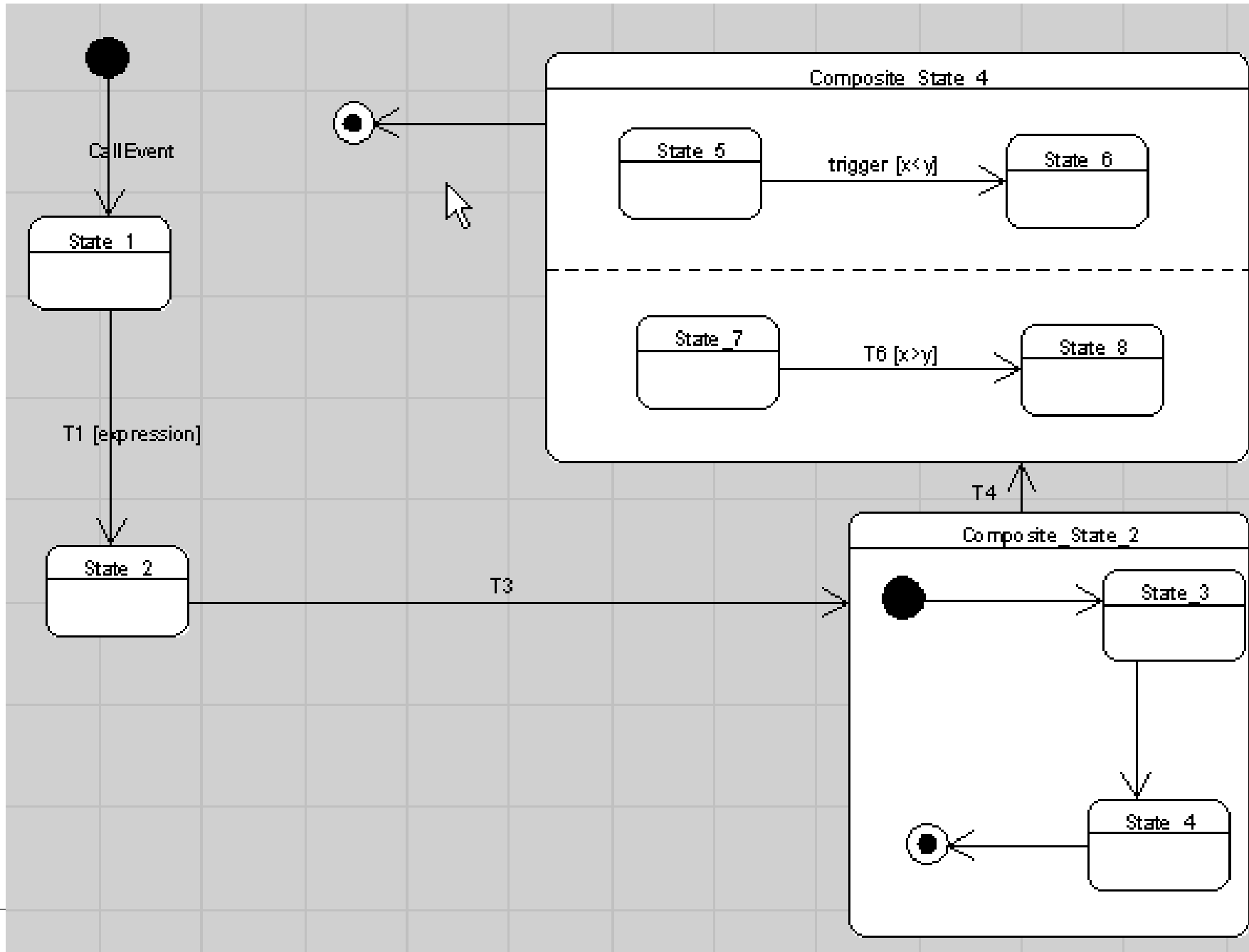
Automate à mémoire



Exemple plus complexe



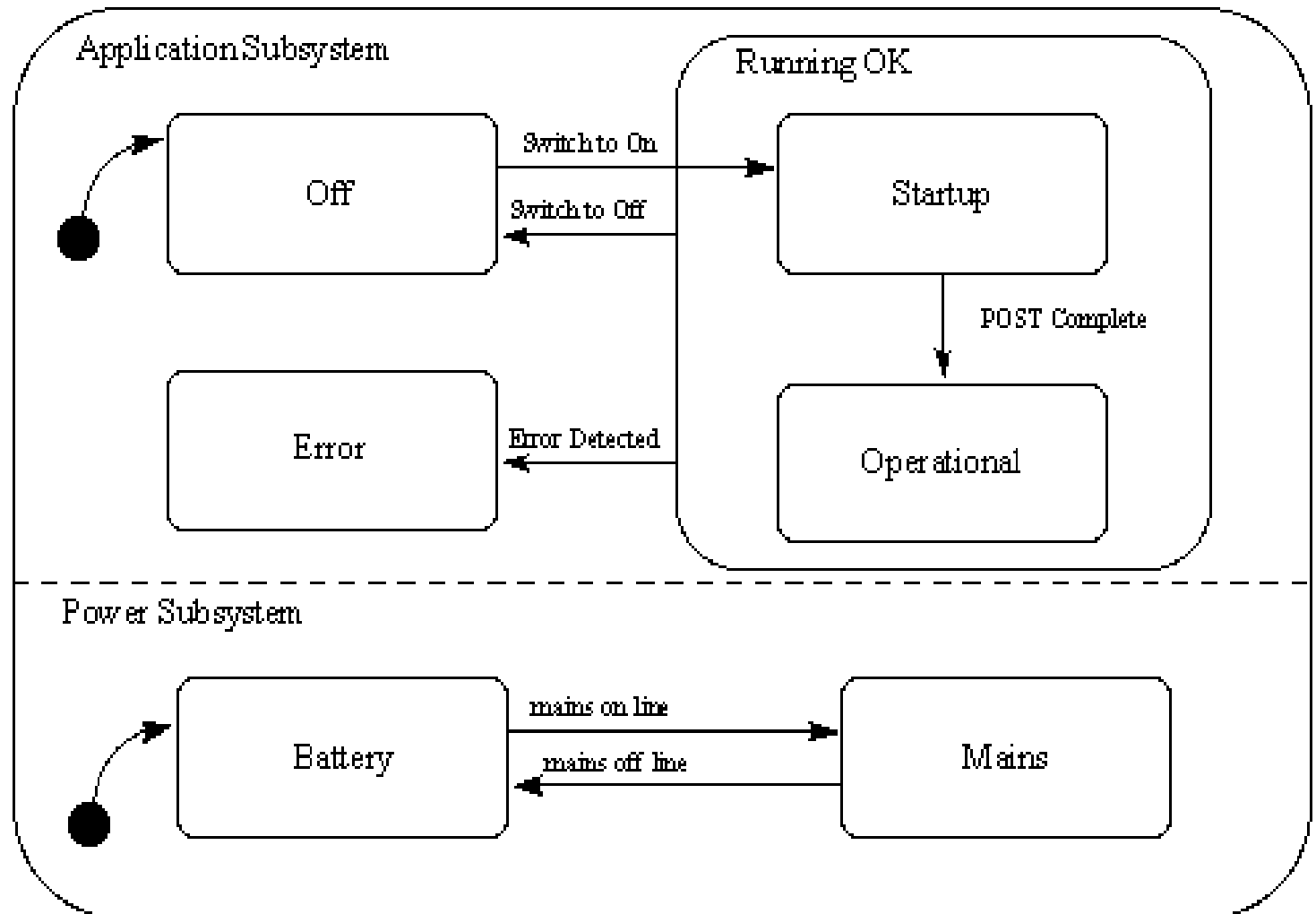
Graphes état-transition



Diagrammes d'états.

Notation de Harel, autorisant :

- gardes sur transitions,
- propagation de transition,
- actions sur transitions,
- actions sur entrée d'état,
- activités apparaissant tant que l'état est actif,
- actions sur sortie d'état,
- imbrication d'états,
- concurrence.



Sous-états séquentiels et concurrents.

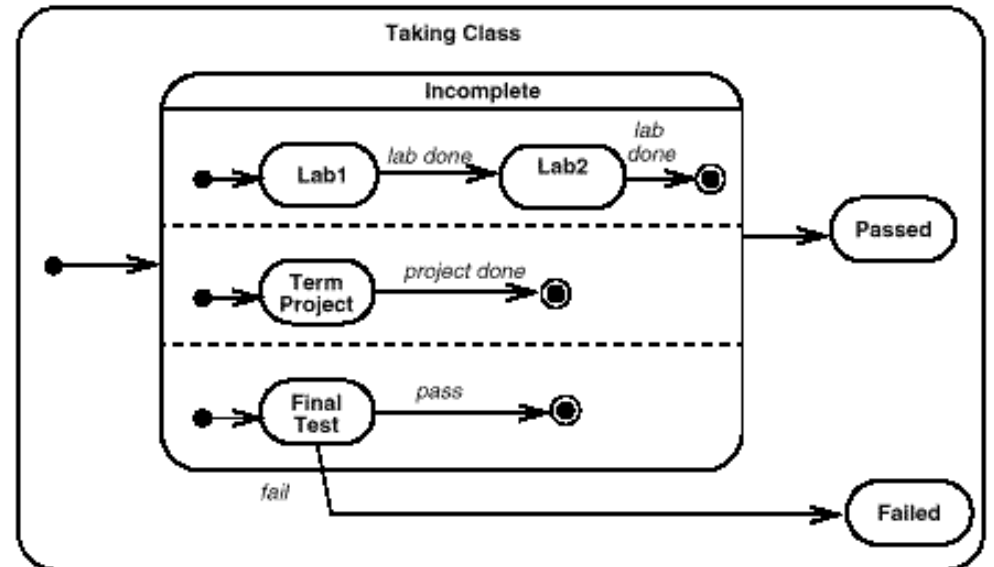
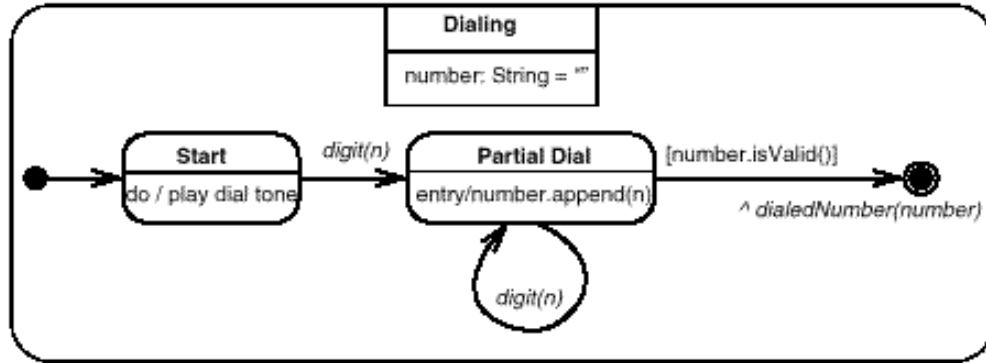
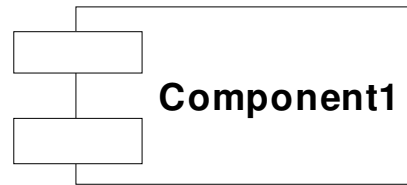


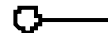
Diagramme de composants

Diagramme de composants

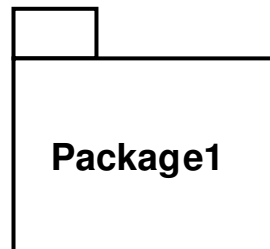
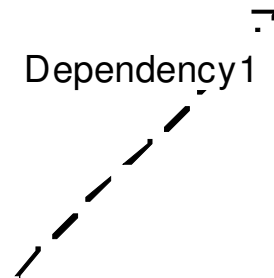
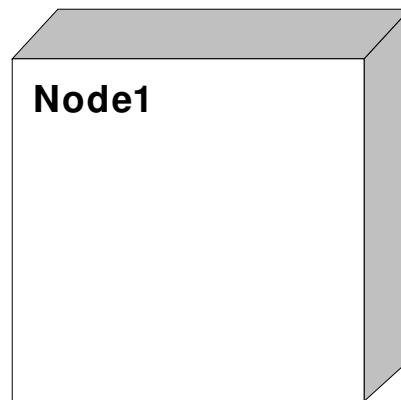
- Les **diagrammes de composants** représentent l'organisation et les dépendances au sein d'un ensemble de composants. Ils présentent la vue d'implémentation statique d'un système et sont liés aux diagrammes de classe dans le sens où un composant correspond généralement à une ou plusieurs classes, interfaces ou collaborations.
- Représentation de la structure du code (fichiers source, binaire, exécutables).
- Partitionnement de l'espace de réalisation (réalisation distribuable)
- Représentation de la structure statique
 - Des composants génériques reliés par des relations de dépendances.

Diagramme de composants



Interface1 

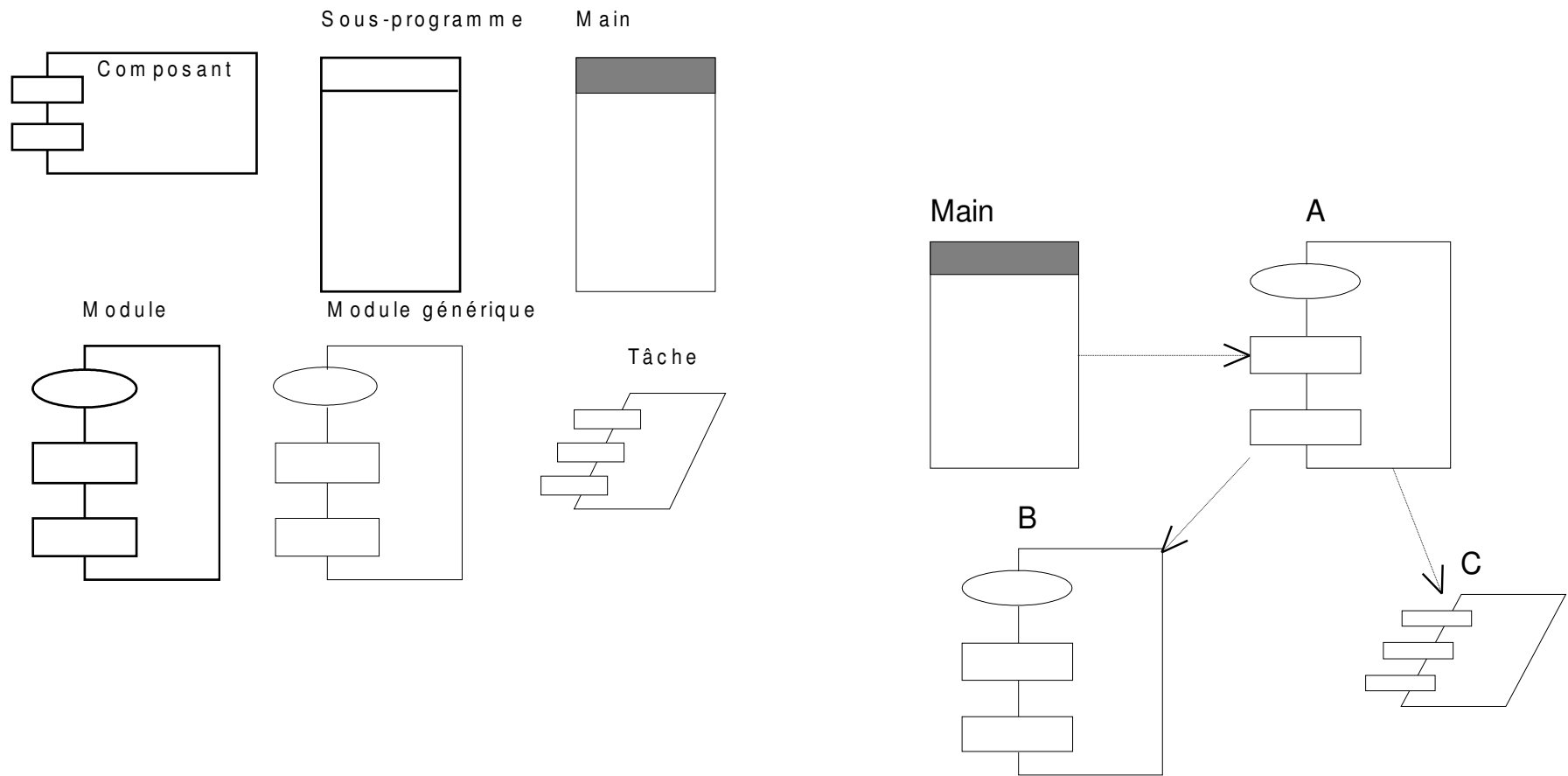
The diagram shows a horizontal line with a small circle at its left end, representing an interface.



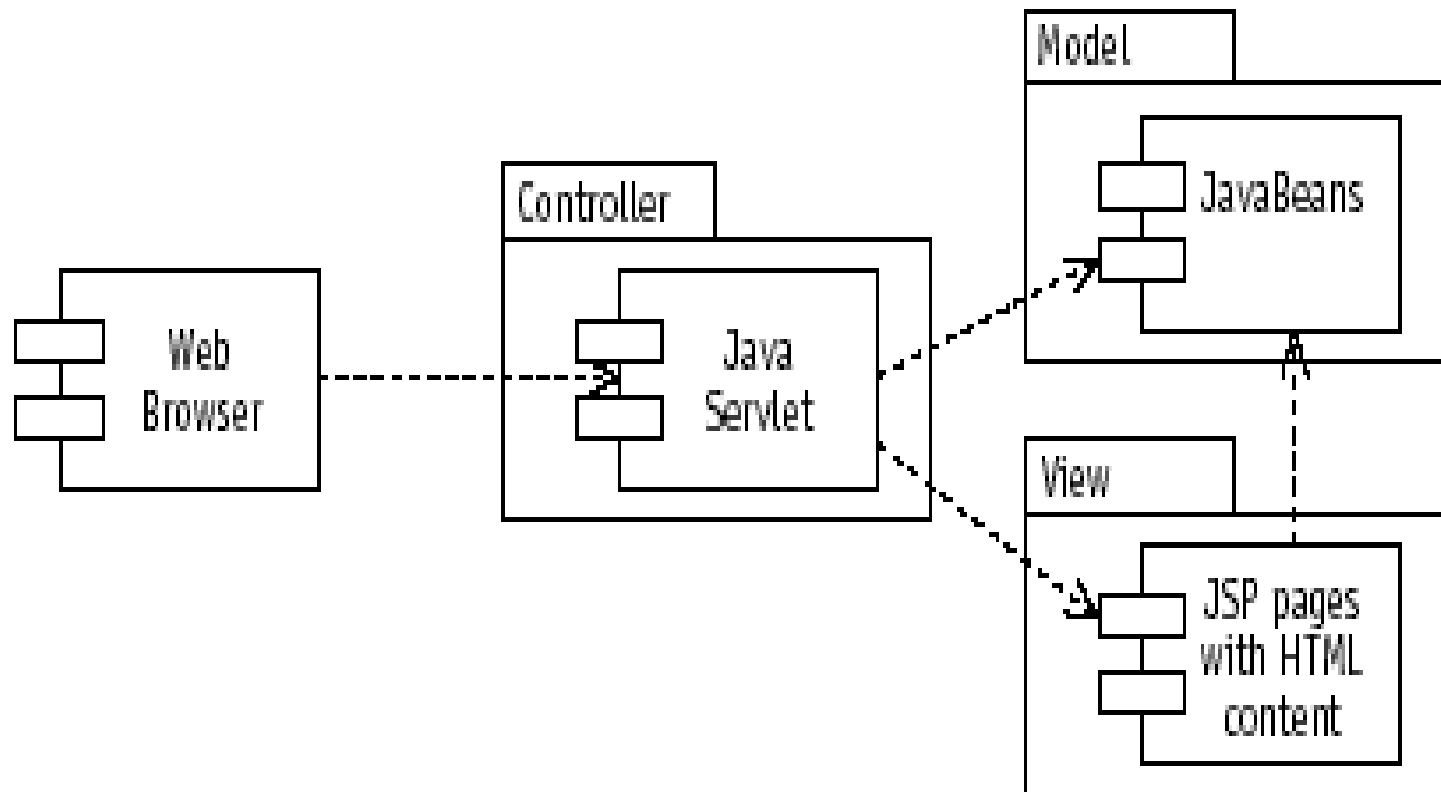
Relation de dépendance des composants

- Une relation d'obsolescence entre 2 composants
 - Flèche avec trait pointillé
 - Concerne les éléments du modèle, pas nécessairement d'instances à l'exécution.

Relation de dépendance des composants



Exemple



Exemple – Serveur JSP

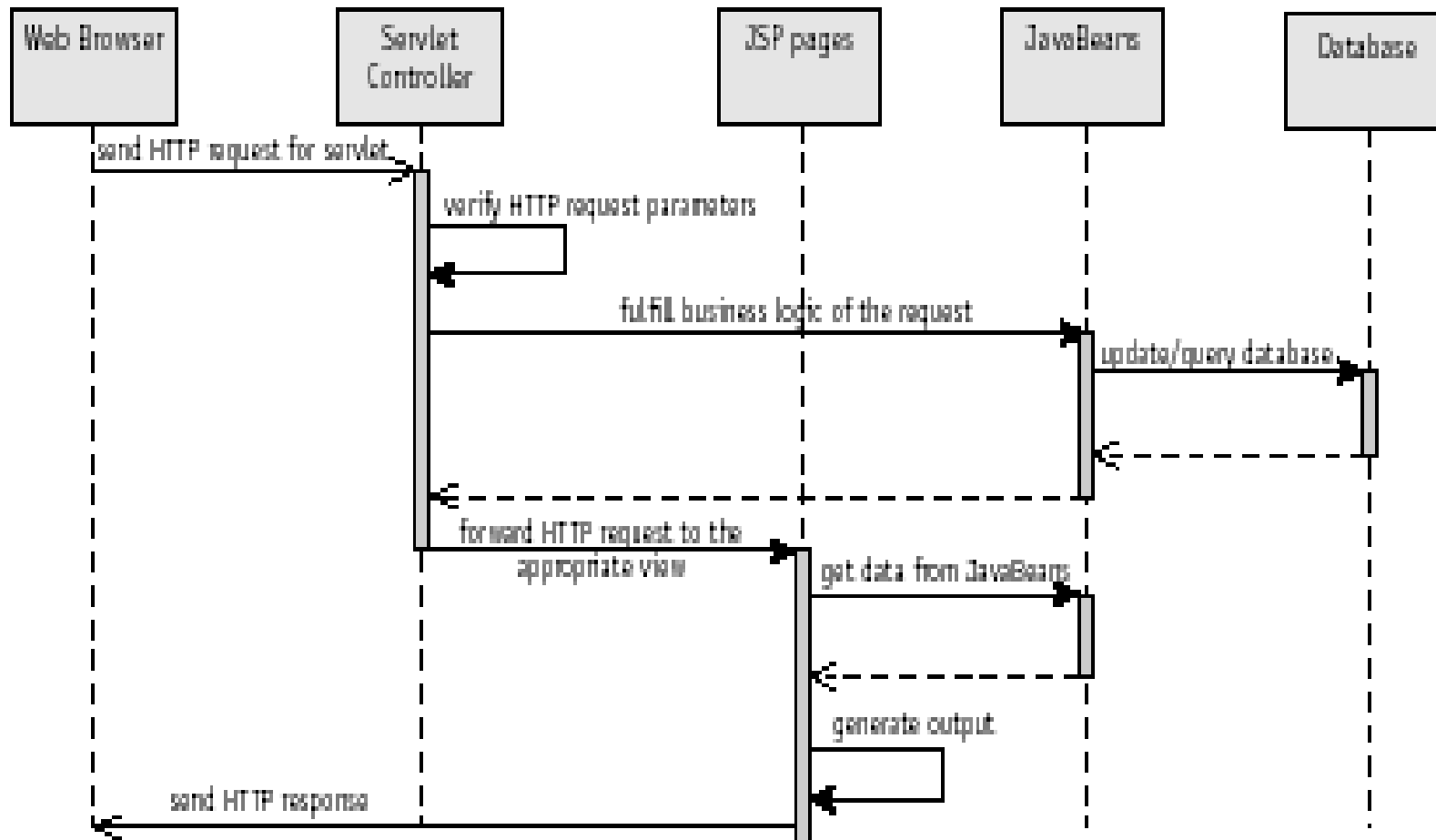
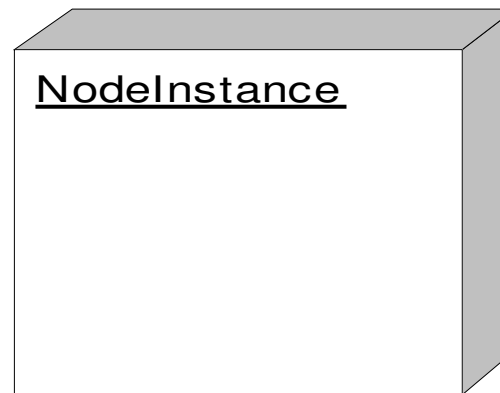
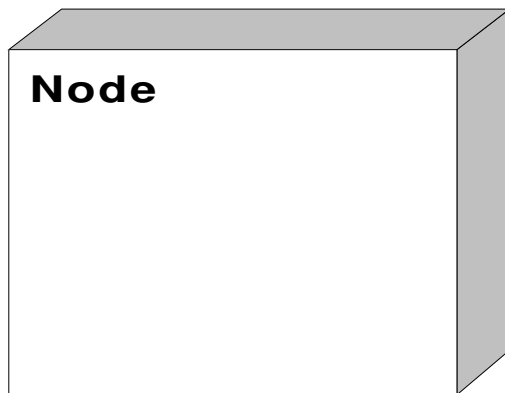
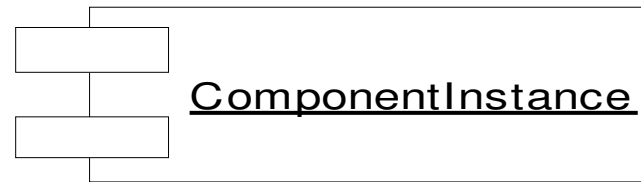
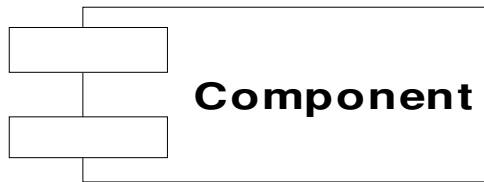


Diagramme de déploiement

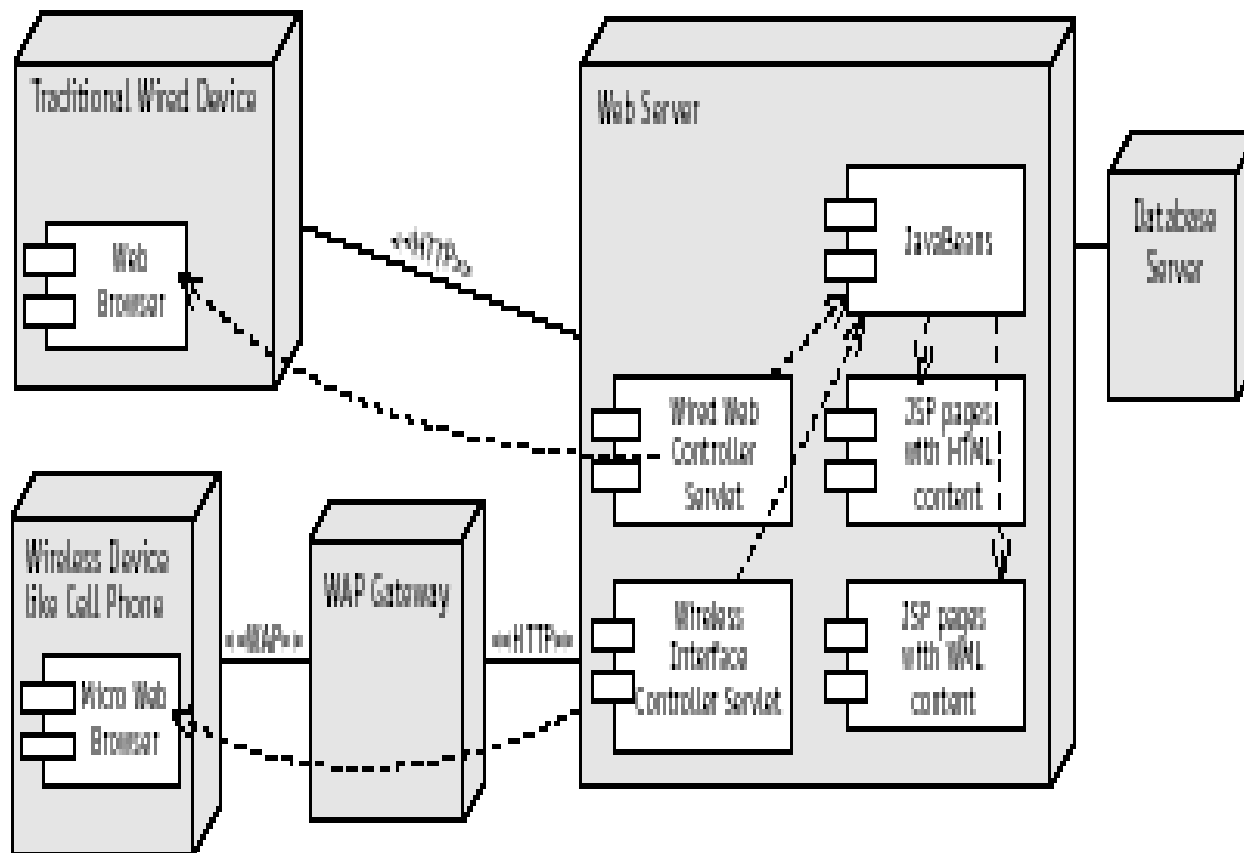
Diagramme de déploiement

- Les *diagrammes de déploiement* représentent la configuration des nœuds de processus en phase d'exécution ainsi que les composants qui y résident. Ils présentent la vue de déploiement statique d'une architecture et sont liés aux diagrammes de composants, dans le sens où un nœud renferme généralement un ou plusieurs composants.
- Représentation de la structure d'un système lors de son exécution
- Relation entre composants logiciels et matériels
- Distributions des composants sur les processeurs
 - Les composants qui n'ont pas d'existence propre à l'exécution se représentent dans les diagrammes de composants

Diagramme de déploiement



Exemple



Résumé

- UML est une notation, pas une méthode
- UML est un langage de modélisation
- UML convient pour toutes les méthodes objets
- UML est dans le domaine public.
- Il existe de nombreux outils (Rational, Poseidon, etc.).