Conception et programmation Orientée objets

Benoît Sonntag

---

sonntag@icps.u-strasbg.fr
sonntag@IsaacOS.com
benoit@sonntag.fr

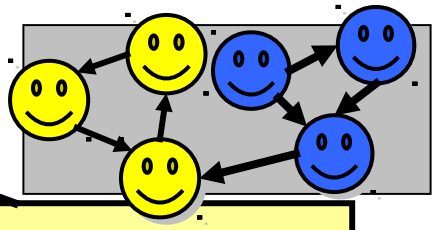
Mon projet…

ISAAC
Object Prototype

www.IsaacOS.com

Technologie à prototype

```
(a > b).if {
    "Ok".print;
};
result
```



Langage à prototype + compilateur

```
01000101
00110011
10100111
01101001
```

Système d'exploitation à prototype

ISAAC
Prototype Language

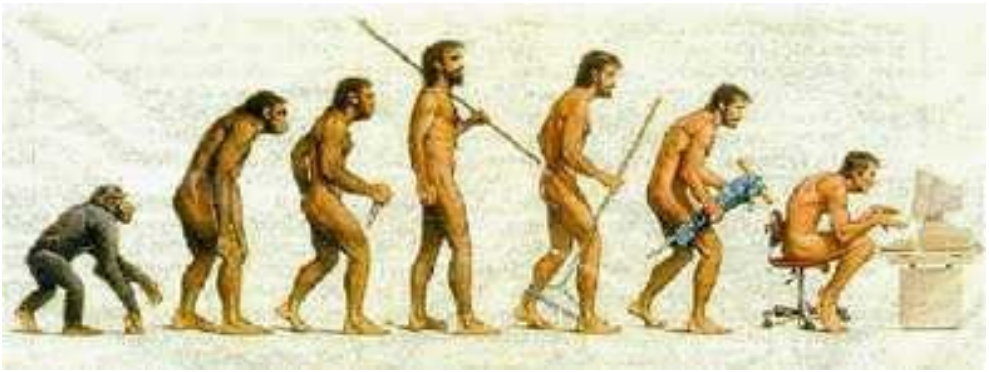
ISAACOS
Operating System

Plan du cours

- Historique et généralités des langages objets.
- Modélisation, analyse et conception objets (UML).
- Programmation Orientée Objet : le langage Java.
- Introduction à l'objet pur : Lisaac

- Cours CM : 12 x 2 = 24 heures.
- TD : 8 x 2 = 16 heures.
- TP : 5 x 2 = 10 heures

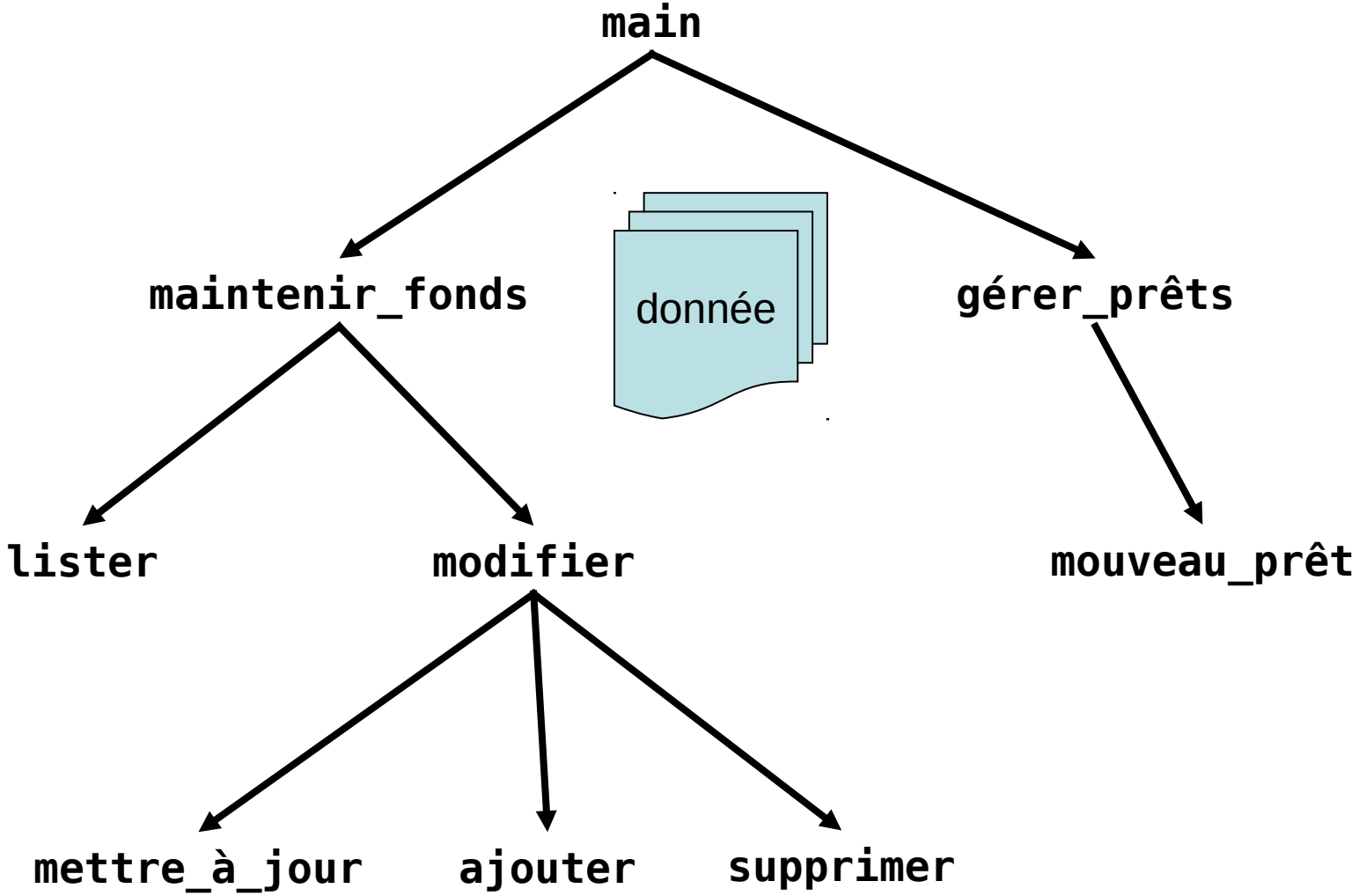
L'évolution des langages de programmation



- L'assembleur : Langage du processeur.
- 1954 : FORTRAN : Le langage des mathématiciens.
- 1960 : **SIMULA** : Le premier langage objet à classe.
- 1963 : BASIC : ***B****eginner's* ***A****ll-purpose* ***S****ymbolic* ***I****nstruction* ***C****ode.*
- 1972 : Le langage C : Programmation système et généraliste.
- 1972 : **SMALLTALK** : Le premier langage interprété à objet pur.
- 1980 : C++ : Le C objet.
- 1994 : EIFFEL : L'apogée des langages objet à classes.
- 1995 : **JAVA** : Le langage objet populaire pour Internet.
- 2002 : LISAAC : Le premier langage à prototype compilé.

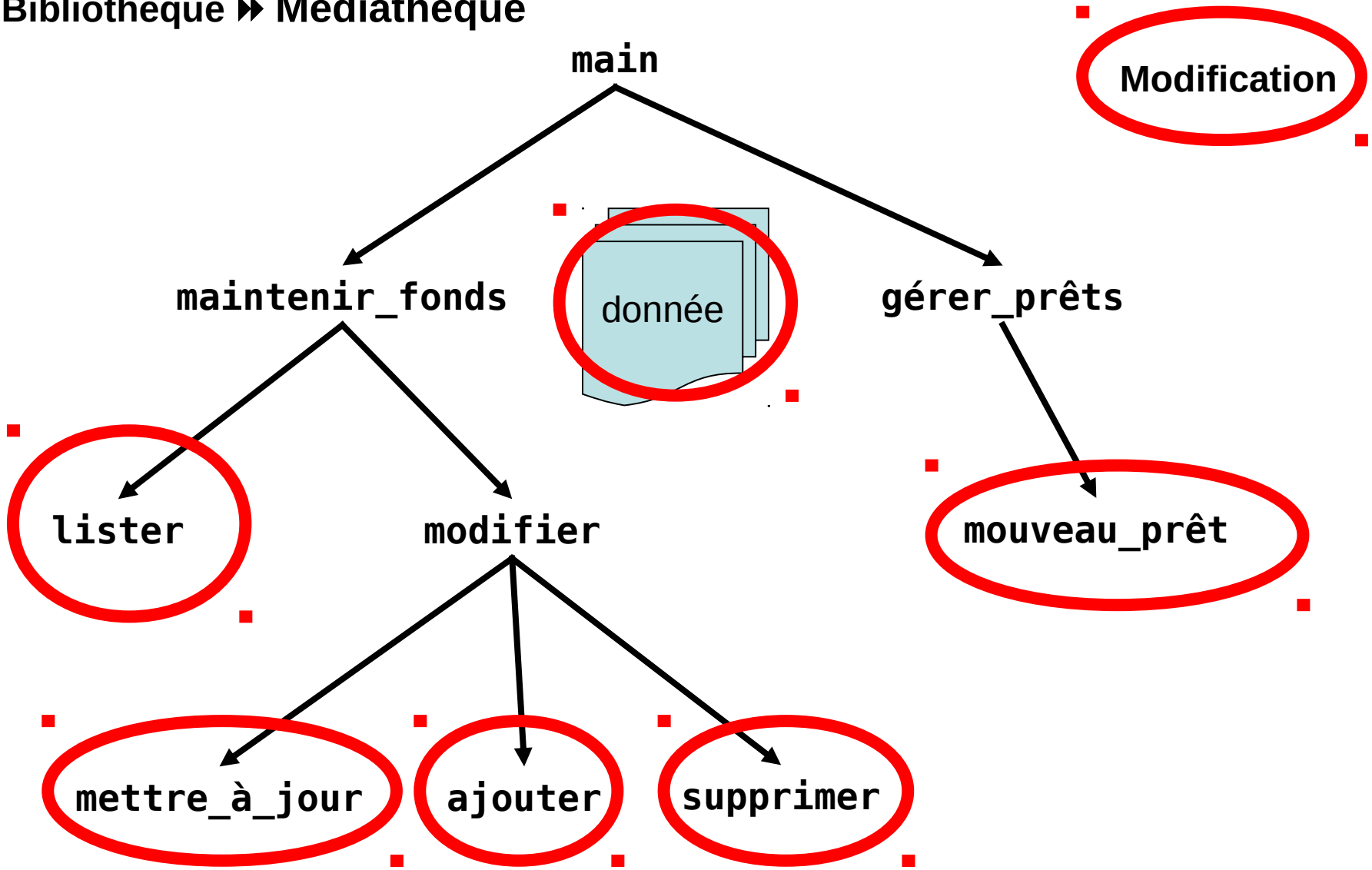
Approche fonctionnelle vs. approche objet

Exemple de découpe fonctionnelle d'un logiciel dédié à la gestion d'une bibliothèque

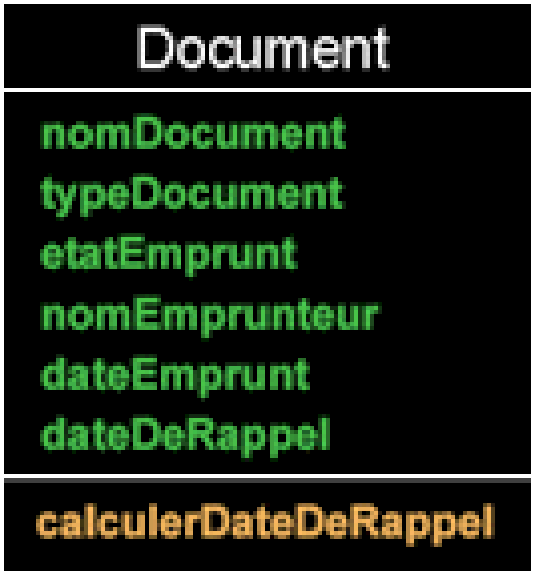


Approche fonctionnelle vs. approche objet

Bibliothèque ⏩ Médiathèque



Approche fonctionnelle vs. approche objet



une entité autonome, qui regroupe un ensemble de propriétés cohérentes et de traitements associés. Une telle entité s'appelle... un **objet** !

Approche fonctionnelle	**: Dirigé par les traitements**
Approche objet	**: Dirigé par le type des données**

Approche fonctionnelle vs. approche objet

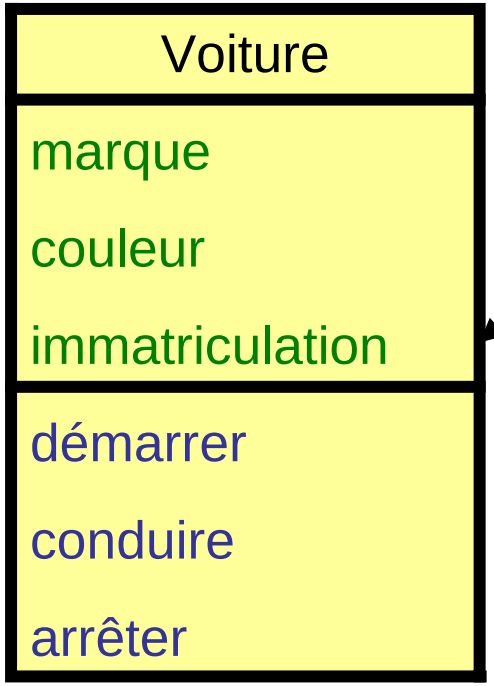
Fonctionnelle

```
void mettre_a_jour(int ref)
{
  /* ... */
  switch (DOC[ref].type)
  {
    case LIVRE:
      maj_livre(DOC[ref]);
      break;
    case CASSETTE_VIDEO:
      maj_k7(DOC[ref]);
      break;
    case CD_ROM:
      maj_cd(DOC[ref]);
      break;
  }
  /* ... */
}
```

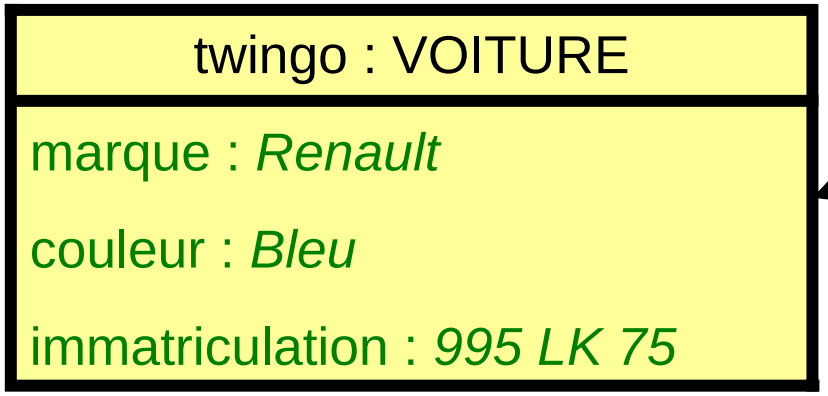
Objet

```
void mettre_a_jour(int ref)
{
  /* ... */
  DOC[ref].maj();
  /* ... */
}
```

Classes et objets



regroupement de données et de traitements dans une **classe** !



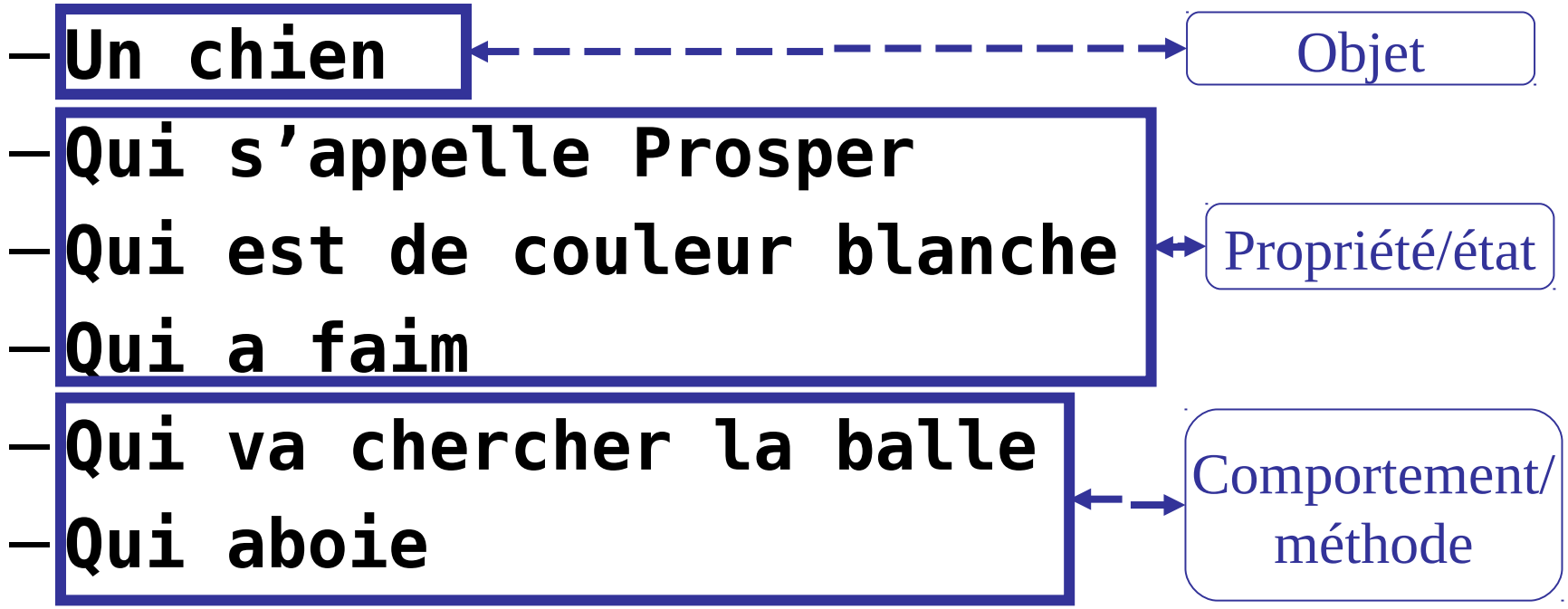
Objet : Instance d'une classe.

Notion d’objet

- **Analogie avec le monde réel :**
  - **Un chien**
  - **Qui s’appelle Prosper**
  - **Qui est de couleur blanche**
  - **Qui a faim**
  - **Qui va chercher la balle**
  - **Qui aboie**

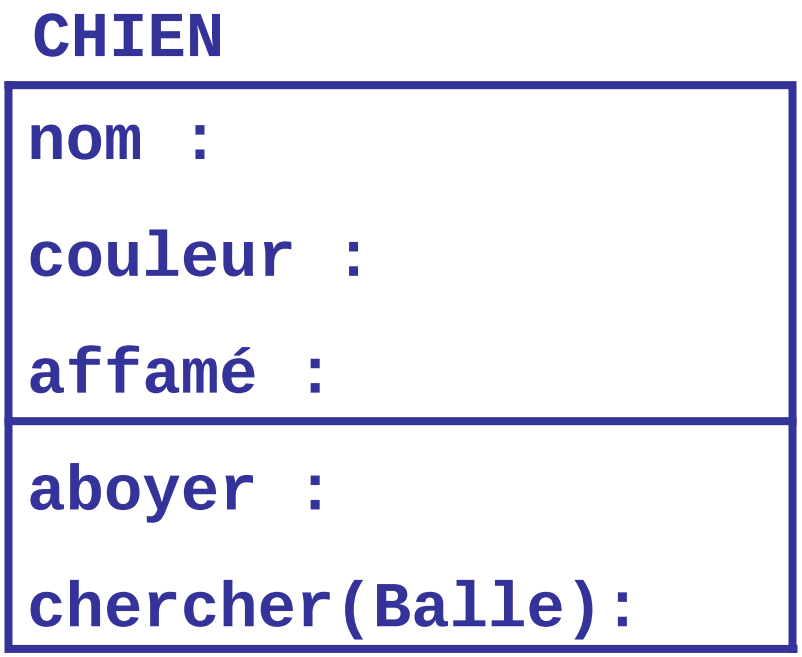
Notion d'objet

- **Analogie avec le monde réel :**



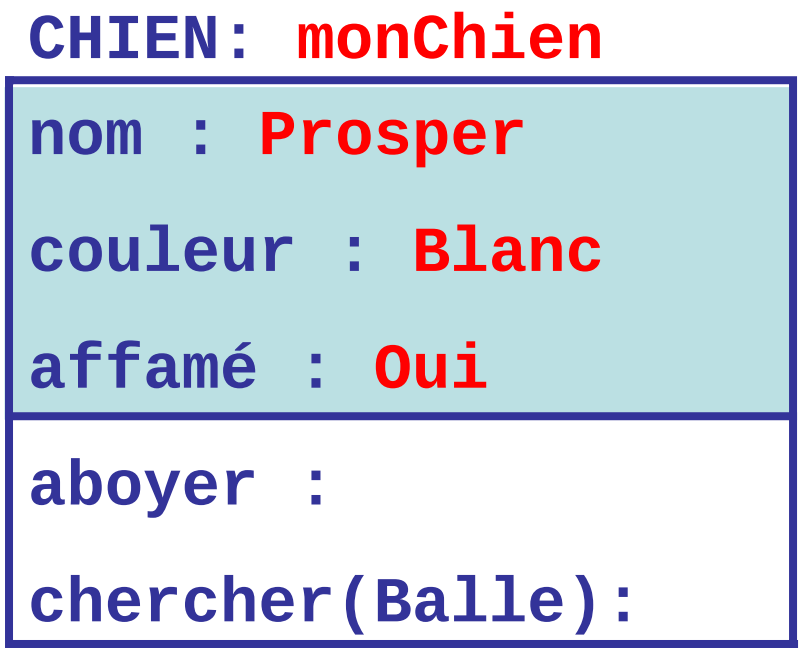
Classe d’objet

Classe : Représentation abstraite d'une catégorie (=type) d'objets.



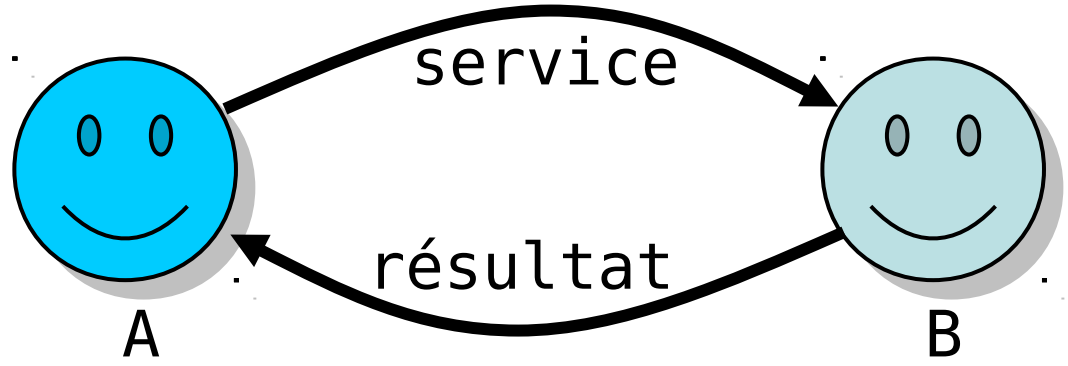
Instance d'une classe : un **objet**

Objet : Représentation en mémoire d'une entité physique (appartenant à une classe)



Communication inter-objets

Dans A: `B . service`

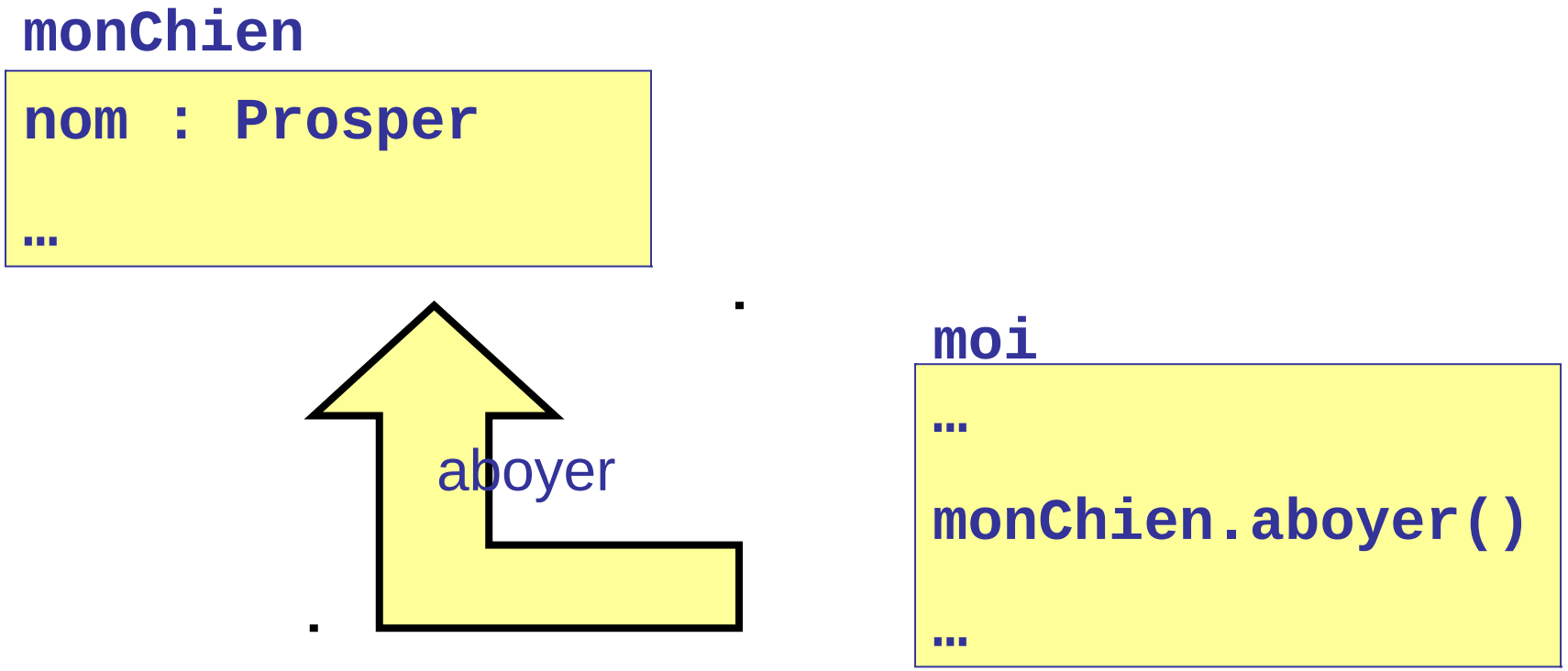


Service: `donnée` ou `une méthode.`

Objet: `données` + `méthodes/fonctions`

Communication inter-objets

<u>Communication</u> : Invocation de message en un point de liaison dynamique



Notion **d'encapsulation**

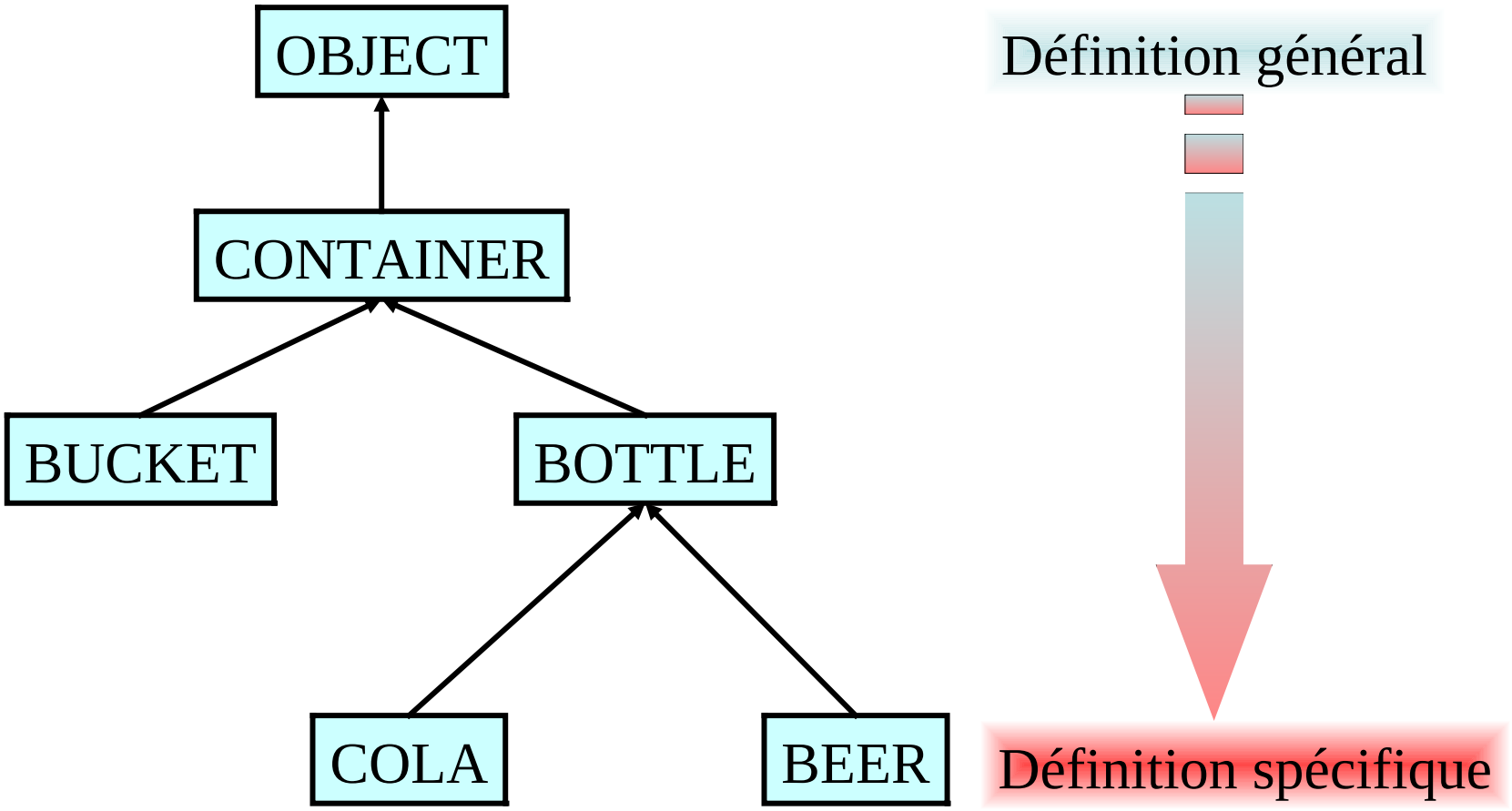
• Masquer les détails d'implémentation d'un objet, en définissant une **interface**.

• **L'interface** : vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.

• L'encapsulation **facilite l'évolution** d'une application car elle stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son interface.

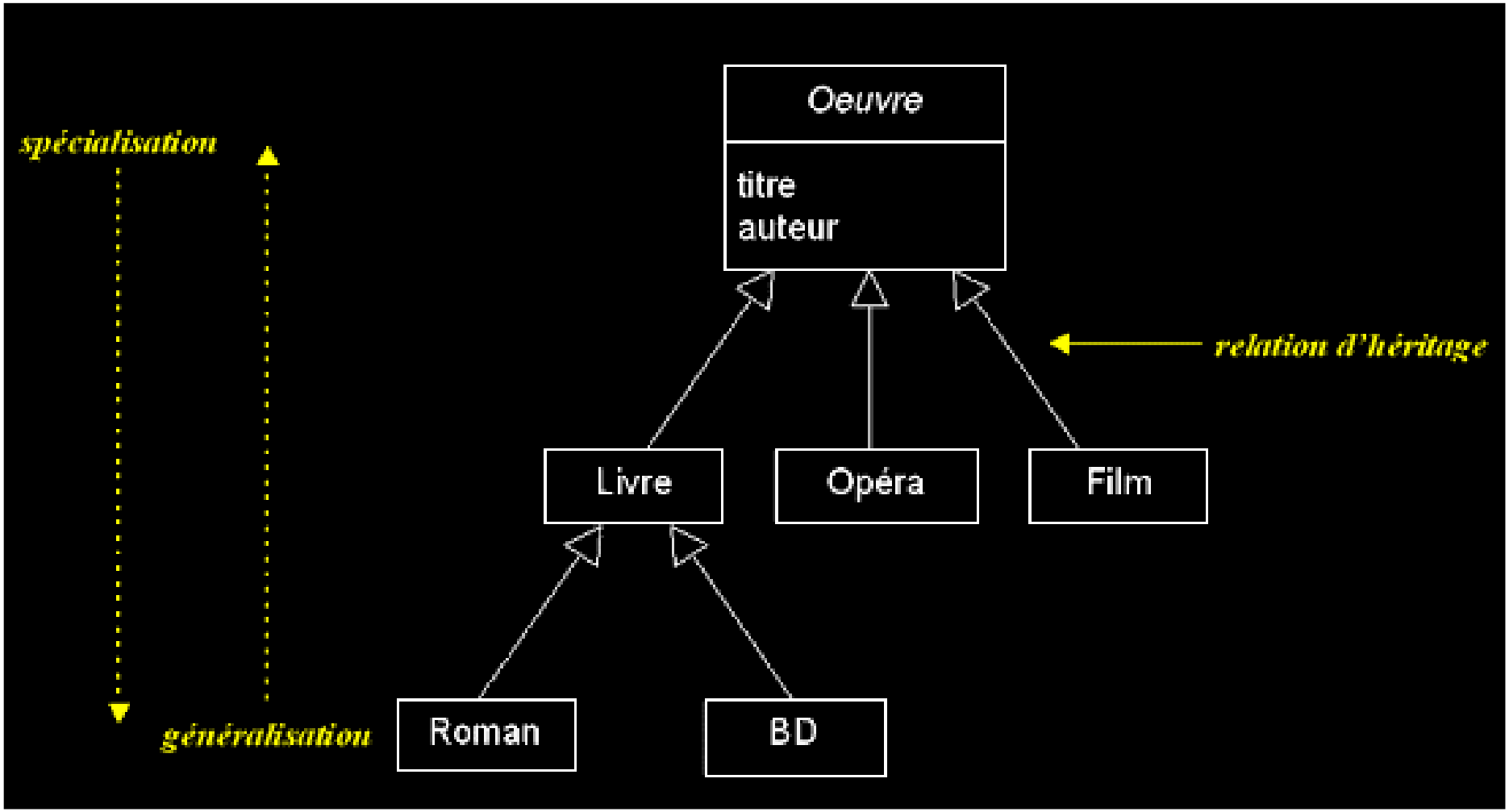
• L'encapsulation **garantit l'intégrité des données**, car elle permet d'interdire l'accès direct aux attributs des objets (utilisation d'accesseurs).

Notion **d'héritage**

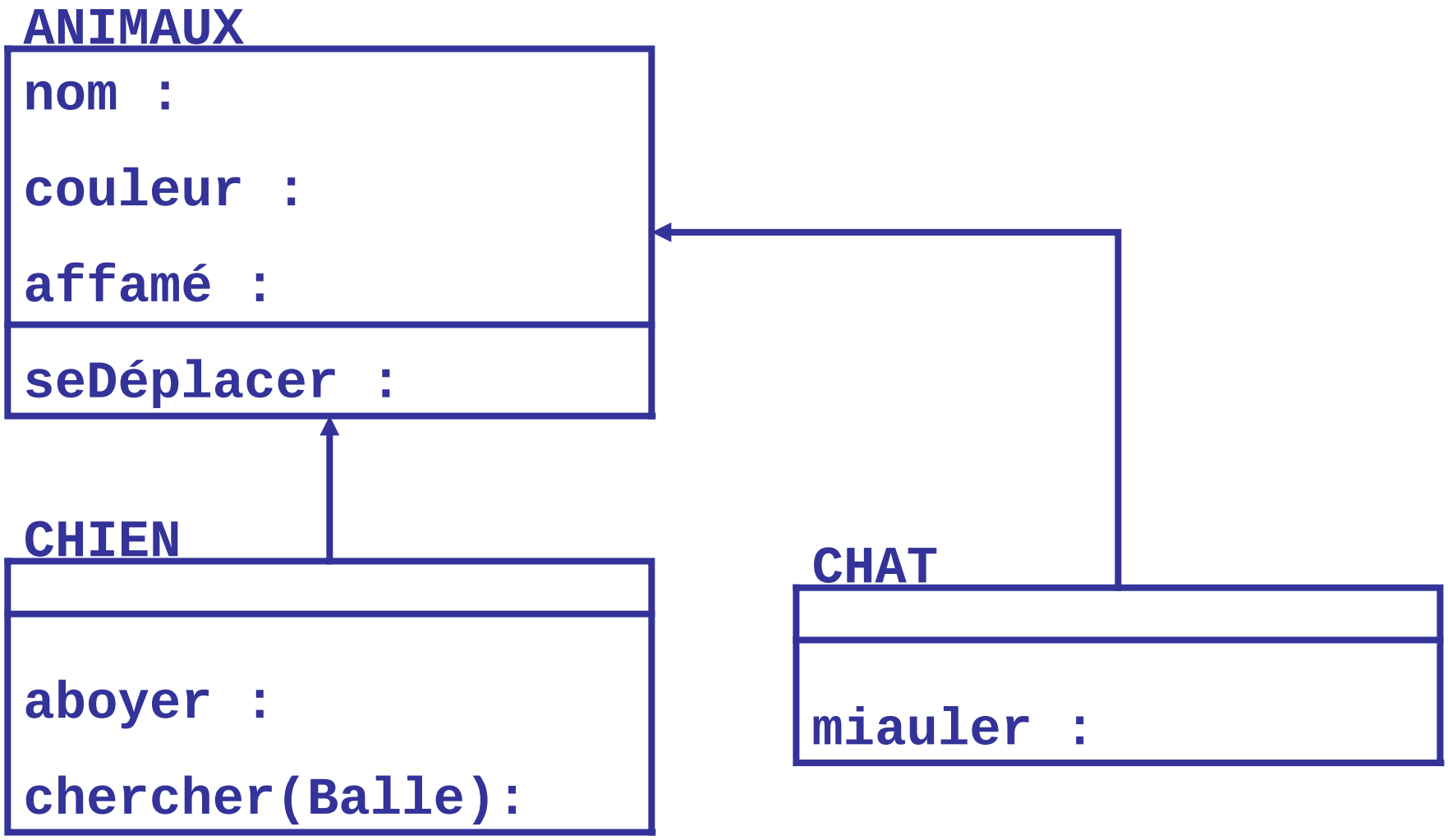


A ⟶ B : A hérite de B

Notion **d'héritage**



Notion **d'héritage**

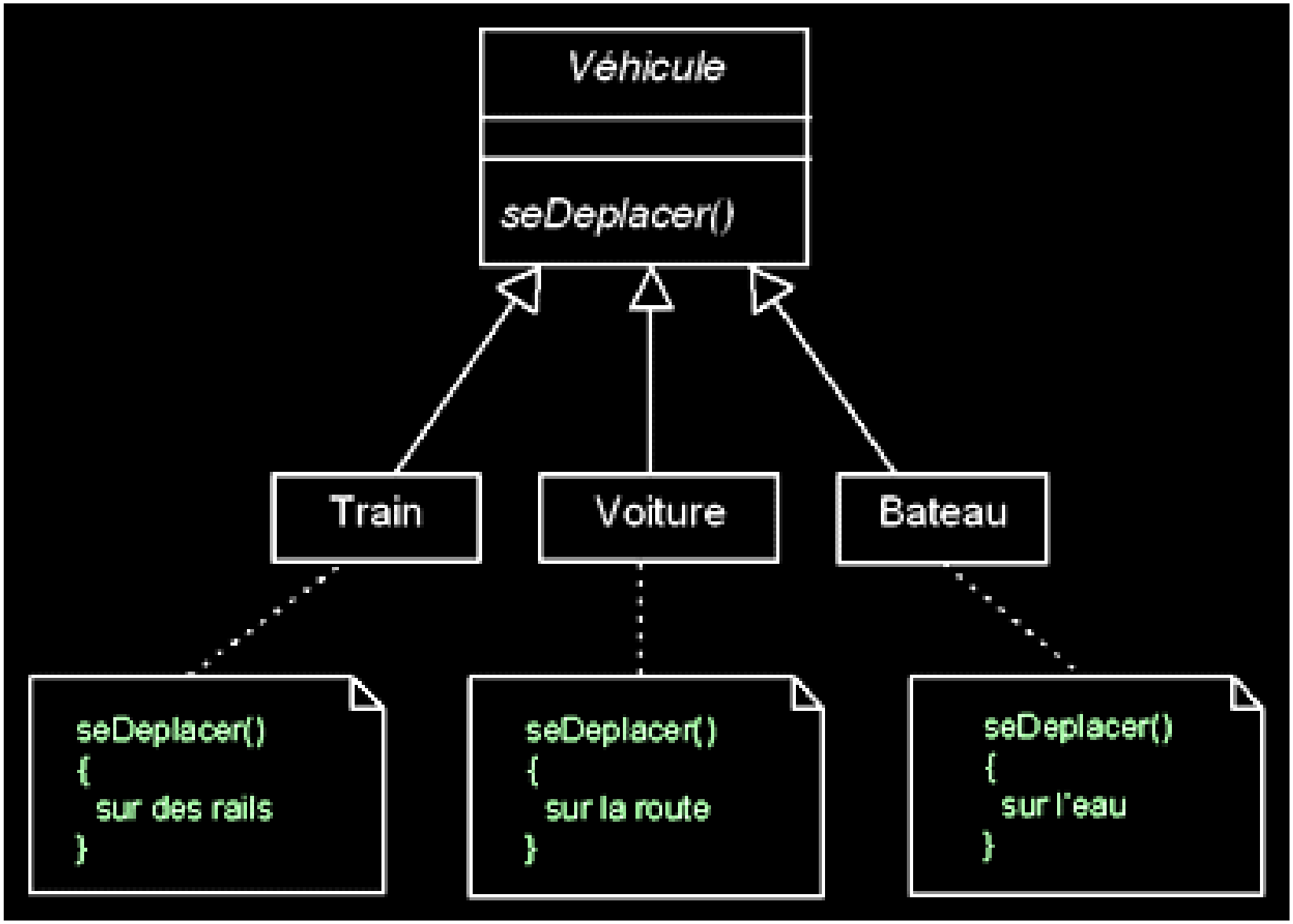


Notion **d'héritage**

- L'héritage est un mécanisme de **transmission des propriétés** d'une classe (ses attributs et méthodes) vers une sous-classe.
- Une classe peut être spécialisée en d'autres classes, afin d'y **ajouter** des caractéristiques spécifiques ou d'en **adapter** certaines (ici, on parle de redéfinition)
- Plusieurs classes peuvent être généralisées en une classe qui les factorise, afin de regrouper les **caractéristiques communes d'un ensemble de classes**.
- La spécialisation et la généralisation permettent de construire des hiérarchies de classes. L'héritage peut être **simple ou multiple**.

L'héritage évite la duplication et encourage la **réutilisation.**

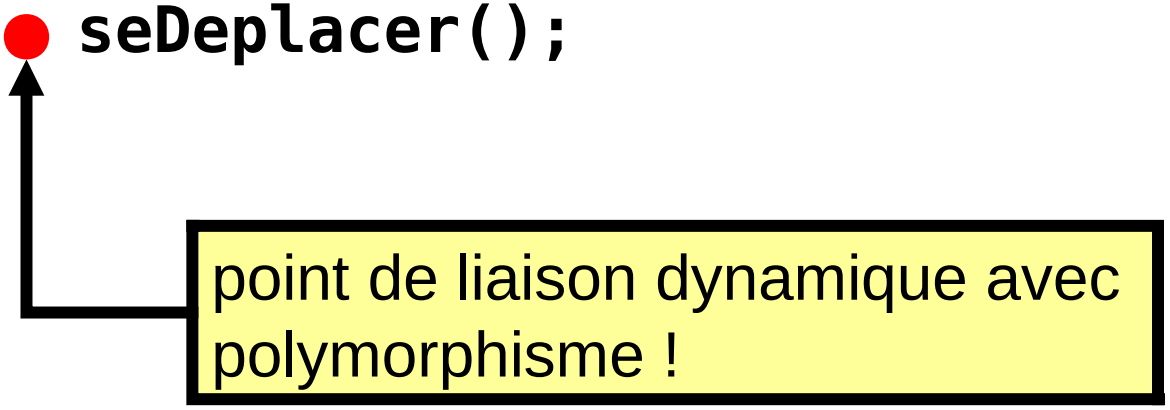
Notion de **polymorphisme**



Notion de **polymorphisme**

```
Vehicule convoi[3] = {
  Train("TGV"),
  Voiture("twingo"),
  Bateau("Titanic")
};

for (int i = 0; i < 3; i++)
{
  convoi[i] seDeplacer();
}
```

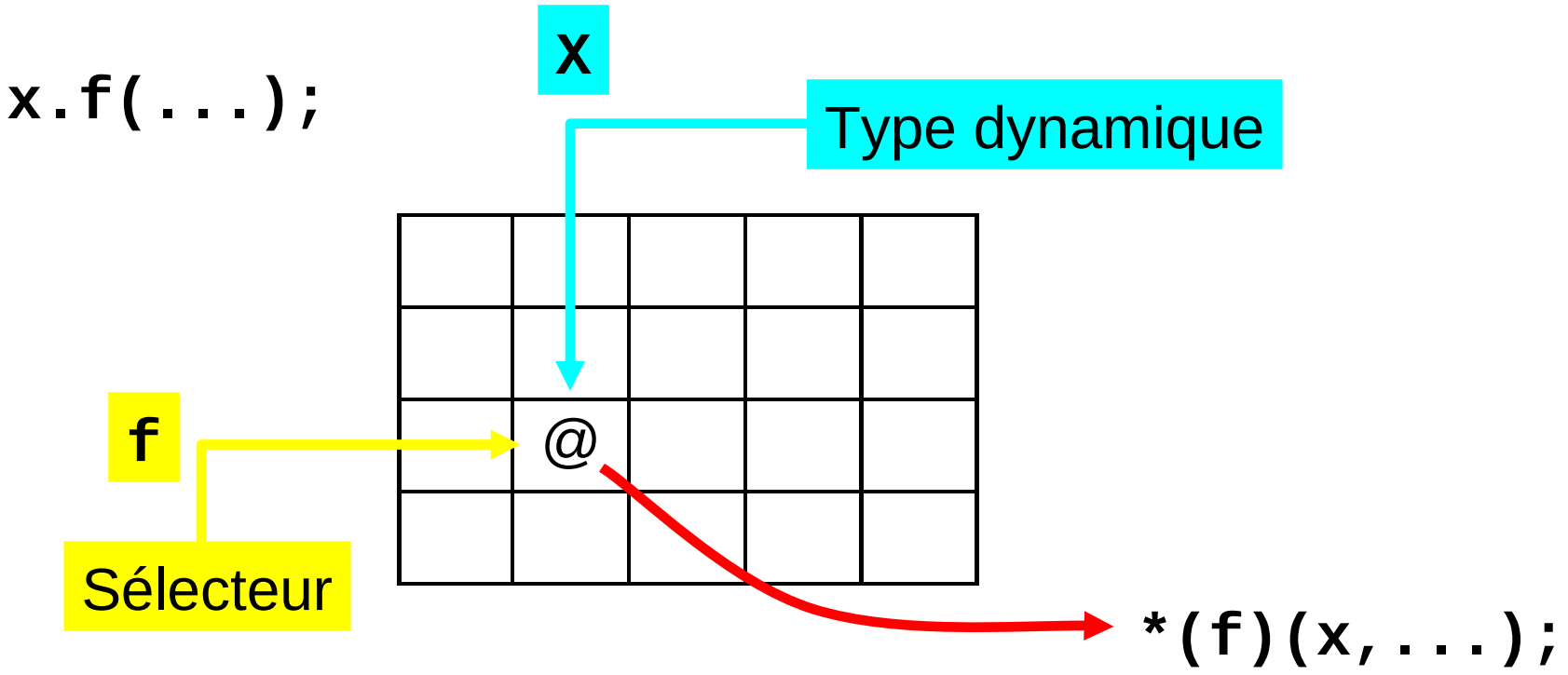


Notion de **polymorphisme**

- Le **polymorphisme** représente la faculté d'une méthode à pouvoir s'appliquer à des objets de **classes différentes**.

Le polymorphisme augmente la généricité du code.

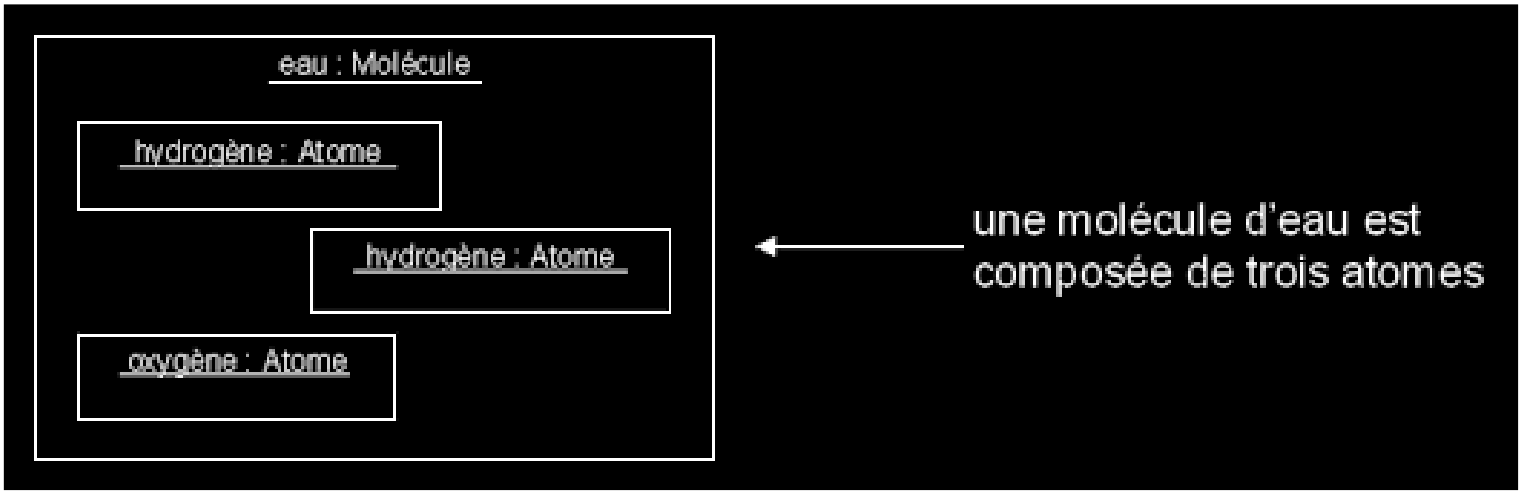
Mécanisme d'envoi de message



Notion **d'Agrégation**

• Il s'agit d'une relation entre deux classes, spécifiant que les objets d'une classe sont des composants de l'autre classe.
Une relation d'agrégation permet donc de définir des objets composés d'autres objets.

• L'agrégation permet d'assembler des objets de base, afin de construire des objets plus complexes.



Les briques fondamentales de l'objet :

Les classes : Représentation abstraite (un modèle) d'une entité logique du système. Composition de propriétés (données) et de traitement sur ces propriétés (fonctions)

Les objets : Représentation concrète (physiquement en mémoire) d'une entité du système. Instance d'une classes.

L'héritage : Définition hiérarchique des objets. Partage des propriétés et des traitements.

Les messages : Communication / consultation / modification entres les objets. Un message s'envoie toujours sur un objet précis.

Le vocabulaire de l'objet :

Encapsulation : L'objet reste le maître absolu de ces propriétés *(Pas comme en Java).* Interface indépendante de l'implantation *(Pas comme en Java !!!!!).*

Agrégation : Définition par composition des objets.

Liaison dynamique : Envoi d'un message sur un objet. Ce lien entre l'objet (type de l'objet) et le message invoqué s'appelle un point de liaison dynamique.

Polymorphisme : Un envoi de message peut être monomorphique (= le traitement reste identique/unique pour tous les types d'objet) ou polymorphique (= traitement diffèrent selon le type l'objet receveur du message).

Les défauts de l'objet :

1. L'approche objet est moins intuitive que l'approche fonctionnelle !
   - Quels moyens utiliser pour faciliter l'analyse objet ?
   - Quels critères identifient une conception objet pertinente ?
   - Comment comparer deux solutions de découpe objet d'un système ?

2. L'application des concepts objets nécessite une grande rigueur !
   - Le vocabulaire est précis (risques d'ambiguïtés, d'incompréhensions).
   - Comment décrire la structure objet d'un système de manière pertinente ?

il nous faut un outil qui apporte une dimension méthodologique à l'approche objet, afin de mieux maîtriser sa richesse et sa complexité.

Introduction à UML

- UML : langage de modélisation objet unifié.
- 1997: Unification de 3 modélisations objets : OMT, BOOCH'93, OOSE

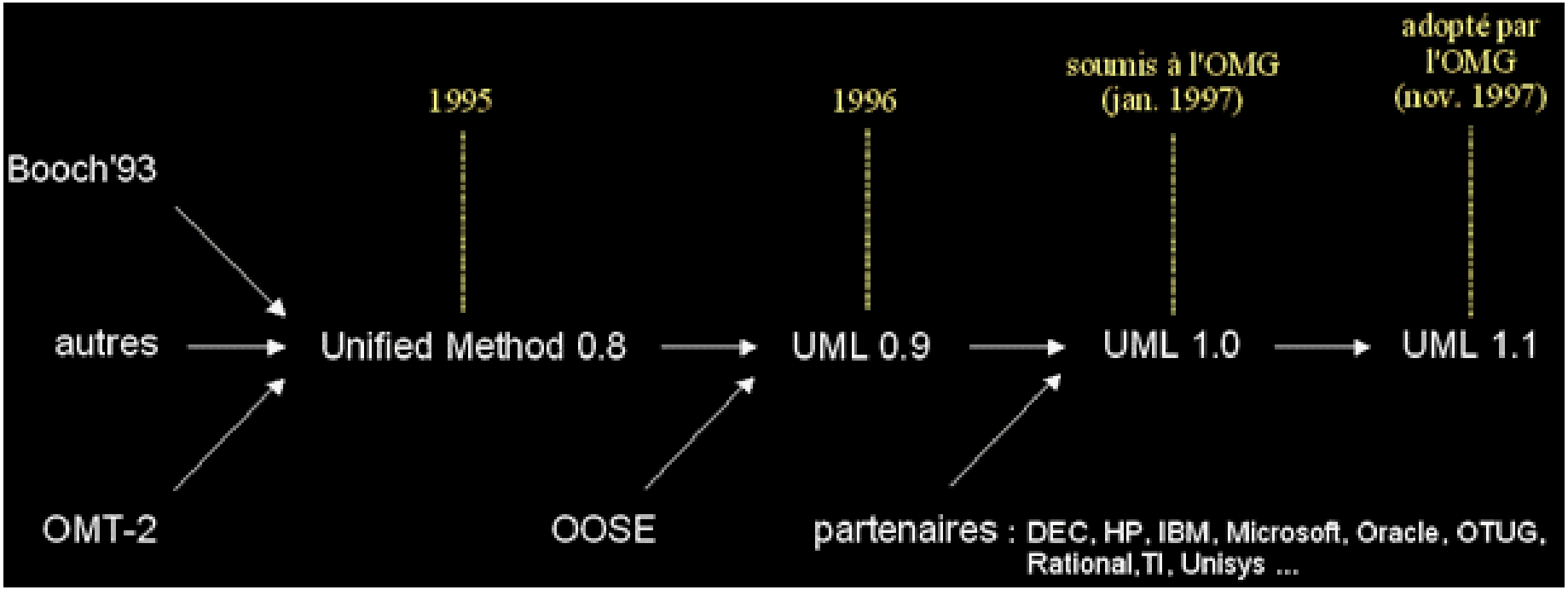
UML, c'est :
- Une norme,
- Un langage de modélisation objet
- Un support de communication
- Un cadre méthodologique

Avant de programmer objet, il faut savoir modéliser objet !

PENSER OBJET !

stop

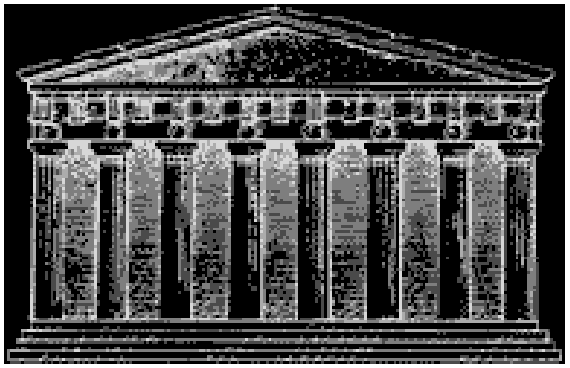
Introduction à UML



Introduction à UML

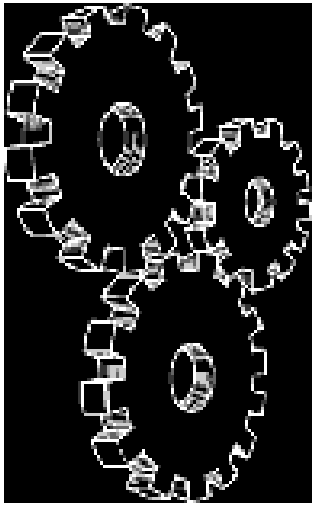
Vues statiques du système :

- ➔ diagrammes de cas d'utilisation
- ➔ diagrammes d'objets
- ➔ diagrammes de classes
- ➔ diagrammes de composants
- ➔ diagrammes de déploiement



Vues dynamiques du système :

- ➔ diagrammes de collaboration
- ➔ diagrammes de séquence
- ➔ diagrammes d'états-transitions
- ➔ diagrammes d'activités

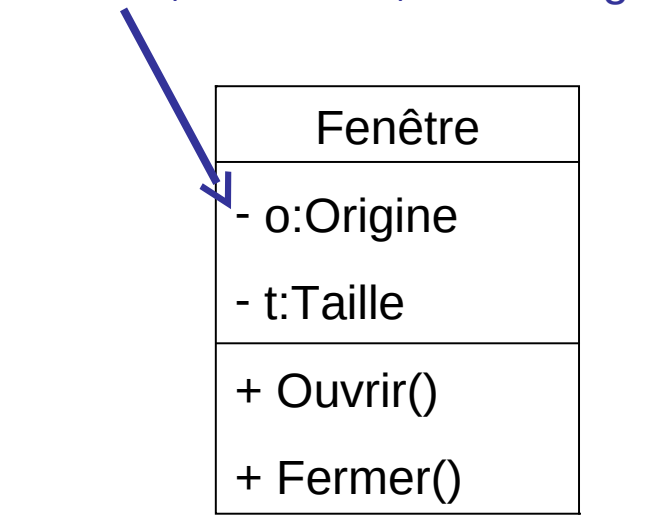


Modéliser les vues statiques d'un système

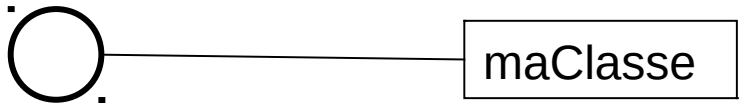
Les éléments structurels

La classe.

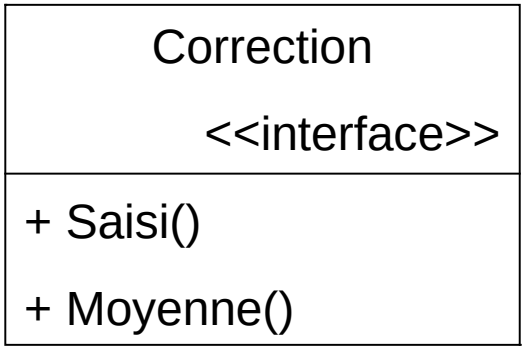
- : Privé, + : Public, # : Protégé



L'interface.

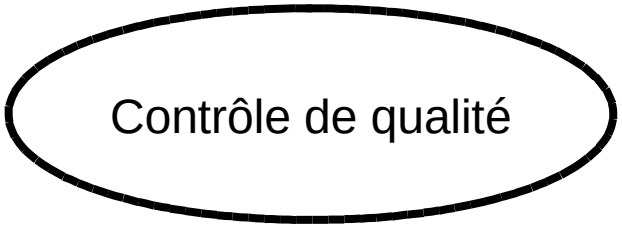


Correction

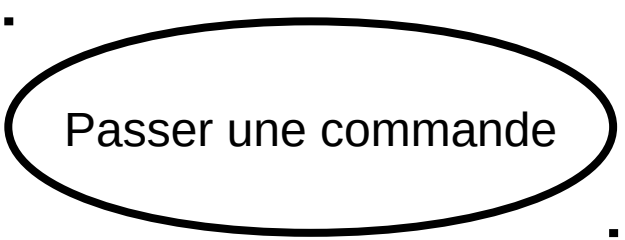


Les éléments structurels

La collaboration.

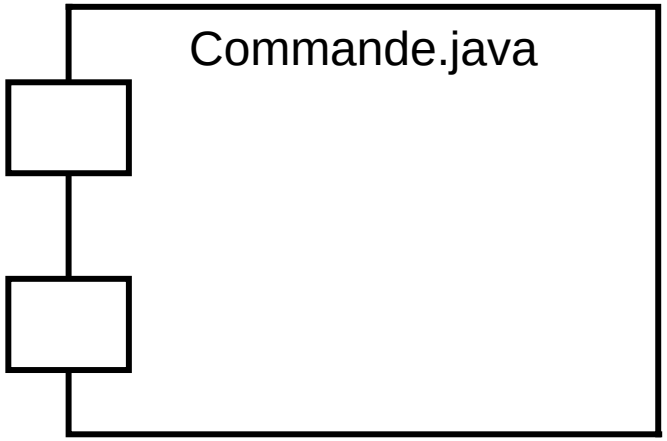


Le cas d'utilisation.

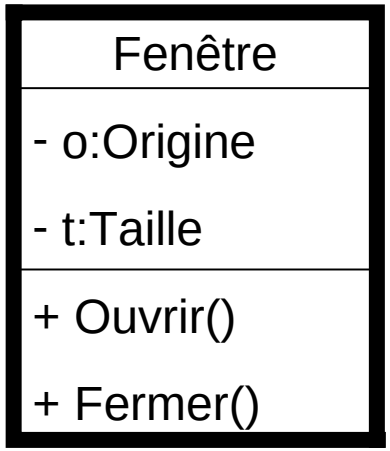


Les éléments structurels

Le composant.

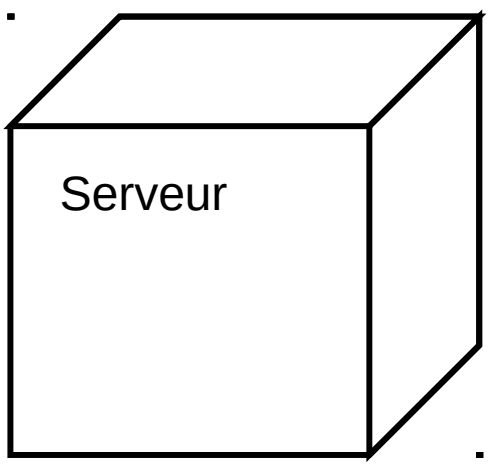


La classe active.



Les éléments structurels

Le nœud.

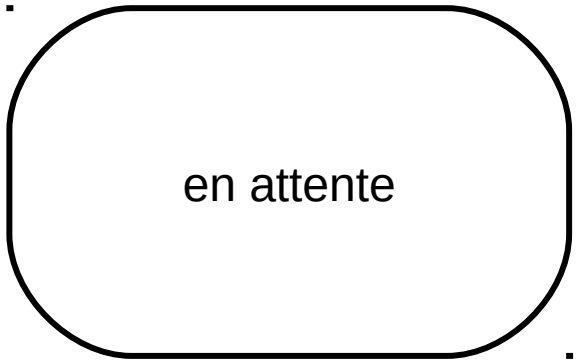


élément physique

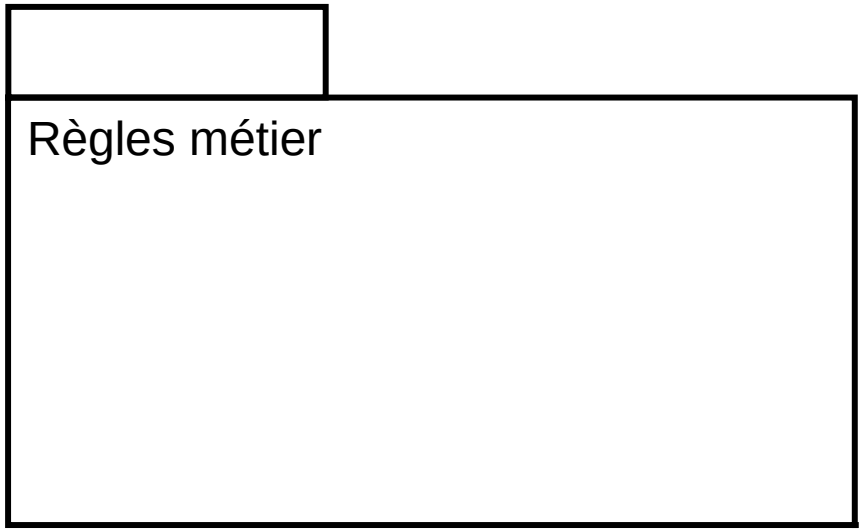
Les éléments comportementaux

Le message.

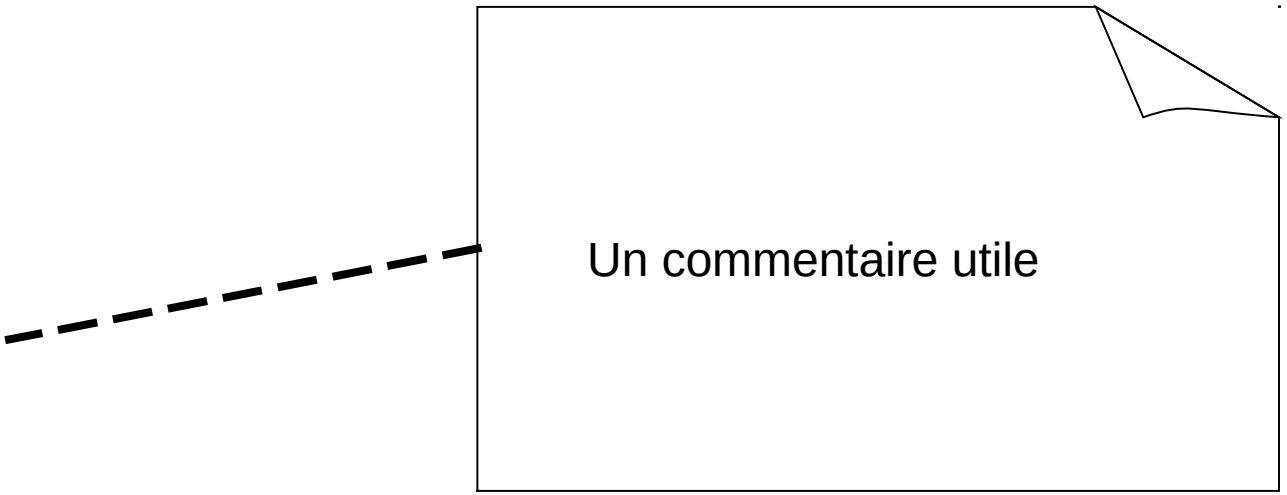
L'état.



Les éléments de regroupement



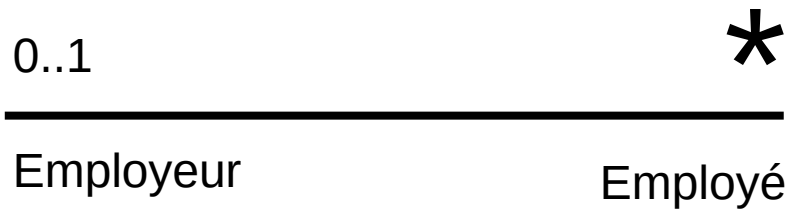
L'élément d'annotation



Les relations de base

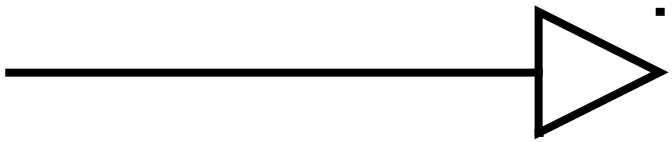
La dépendance

L'association



Les relations de base

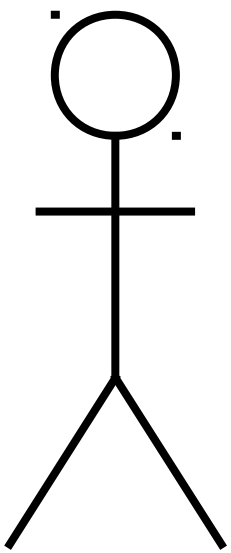
La généralisation / Héritage



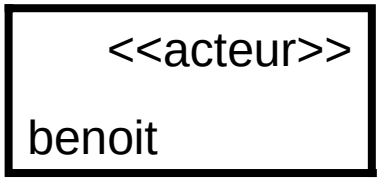
La réalisation / relation sémantique

Les acteurs

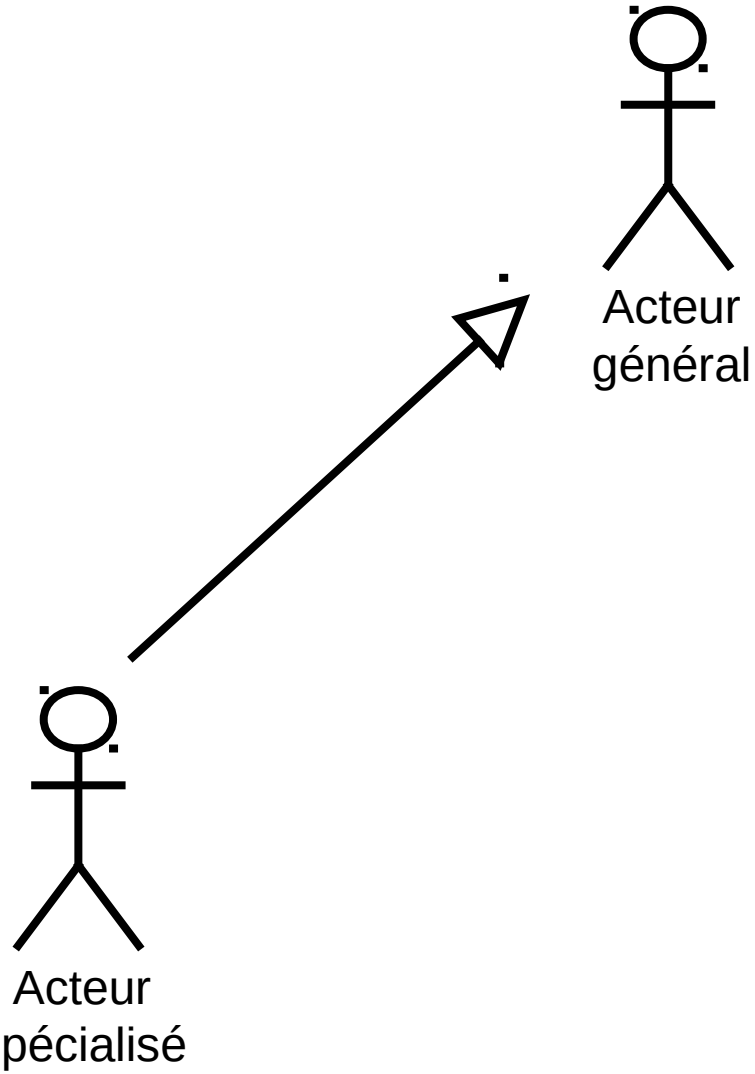
Les acteurs



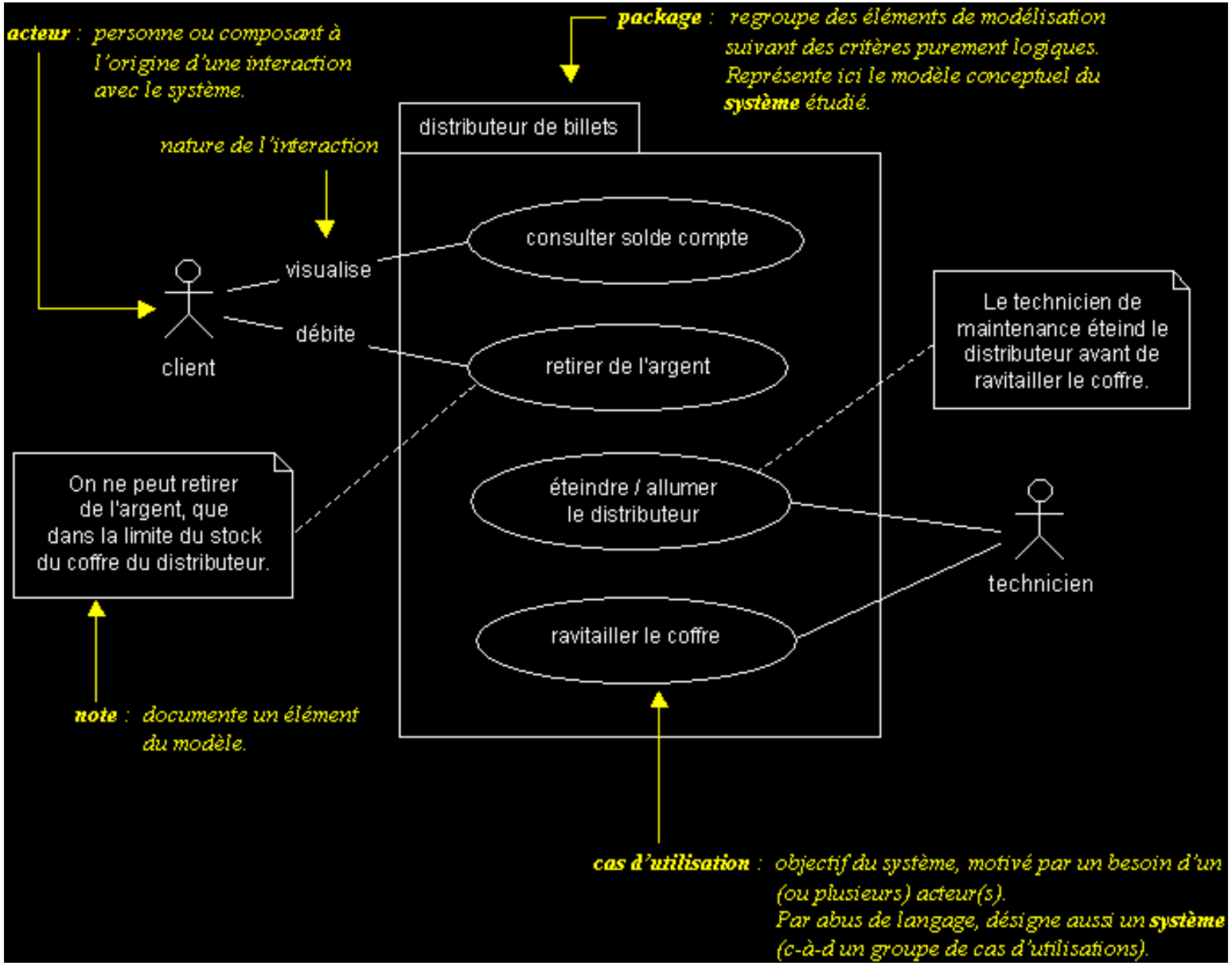
Benoit



Exemple de spécialisation d'un acteur



diagrammes de cas d'utilisation



diagrammes de cas d'utilisation

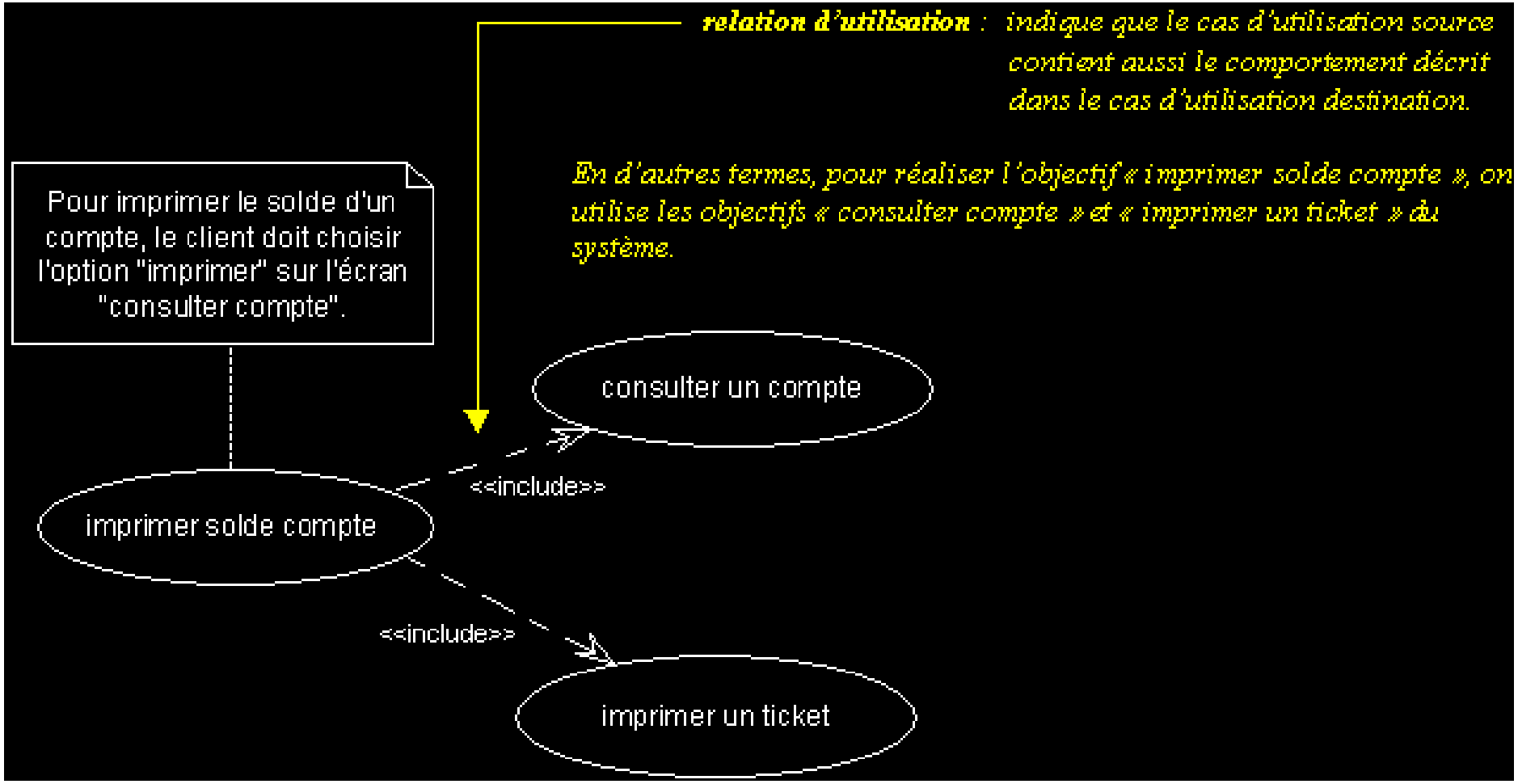
Acteur : entité externe qui agit sur le système (opérateur, autre système...).

- L'acteur peut consulter ou modifier l'état du système.
- En réponse à l'action d'un acteur, le système fournit un service qui correspond à son besoin.
- Les acteurs peuvent être classés (hiérarchisés).

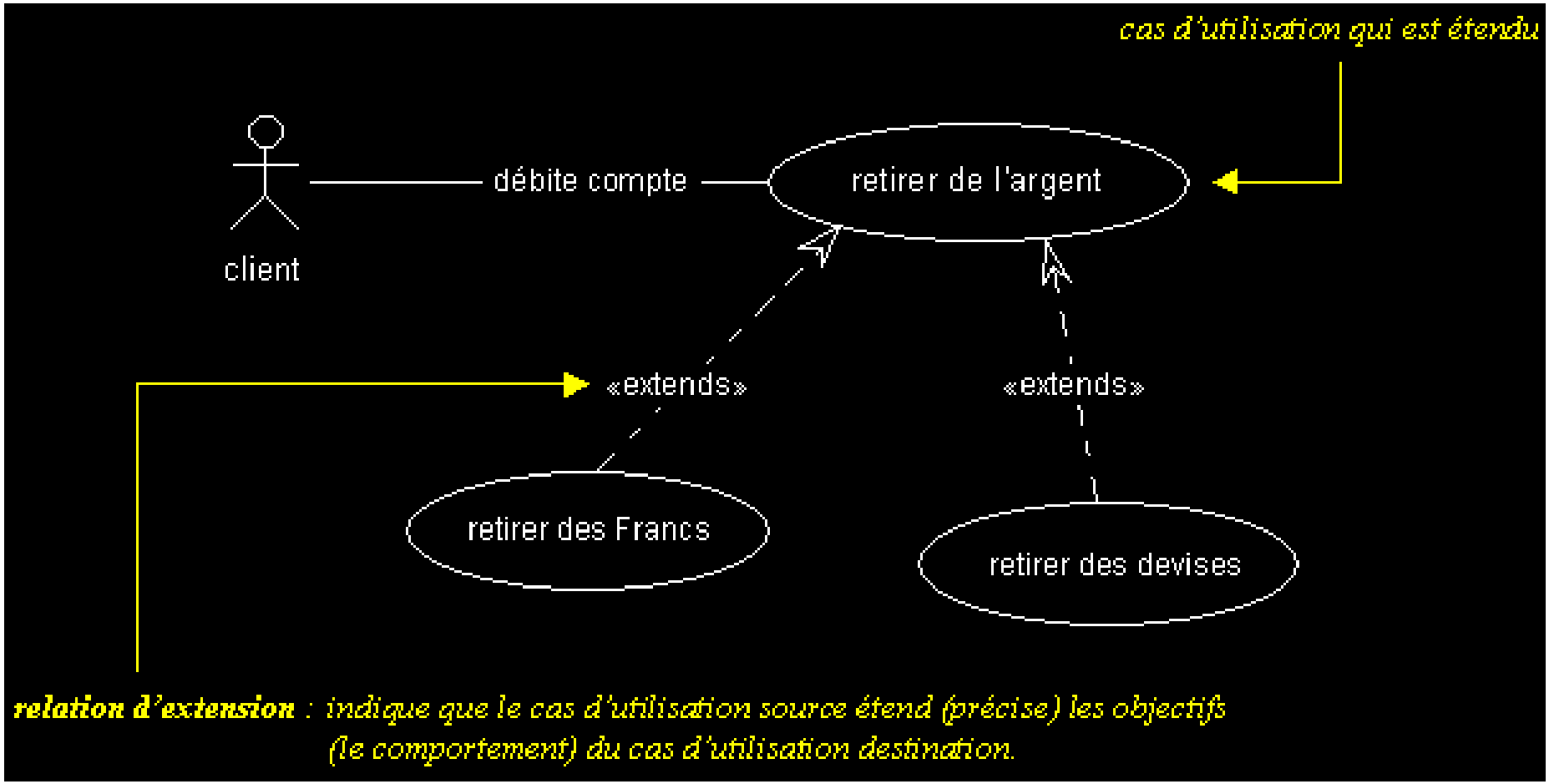
Use case : ensemble d'actions réalisées par le système, en réponse à une action d'un acteur.

- Les uses cases peuvent être structurés.
- Les uses cases peuvent être organisés en paquetages (packages).
- L'ensemble des use cases décrit les objectifs (le but) du système

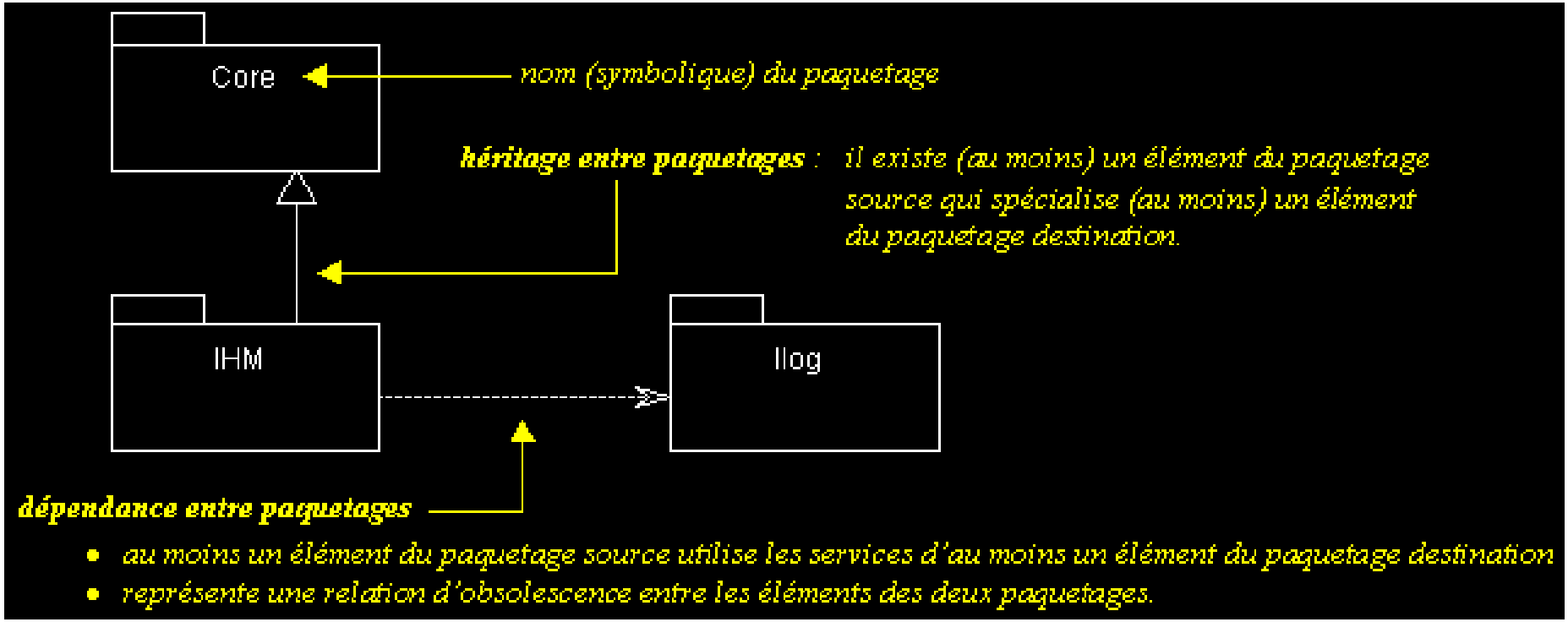
diagrammes de cas d'utilisation



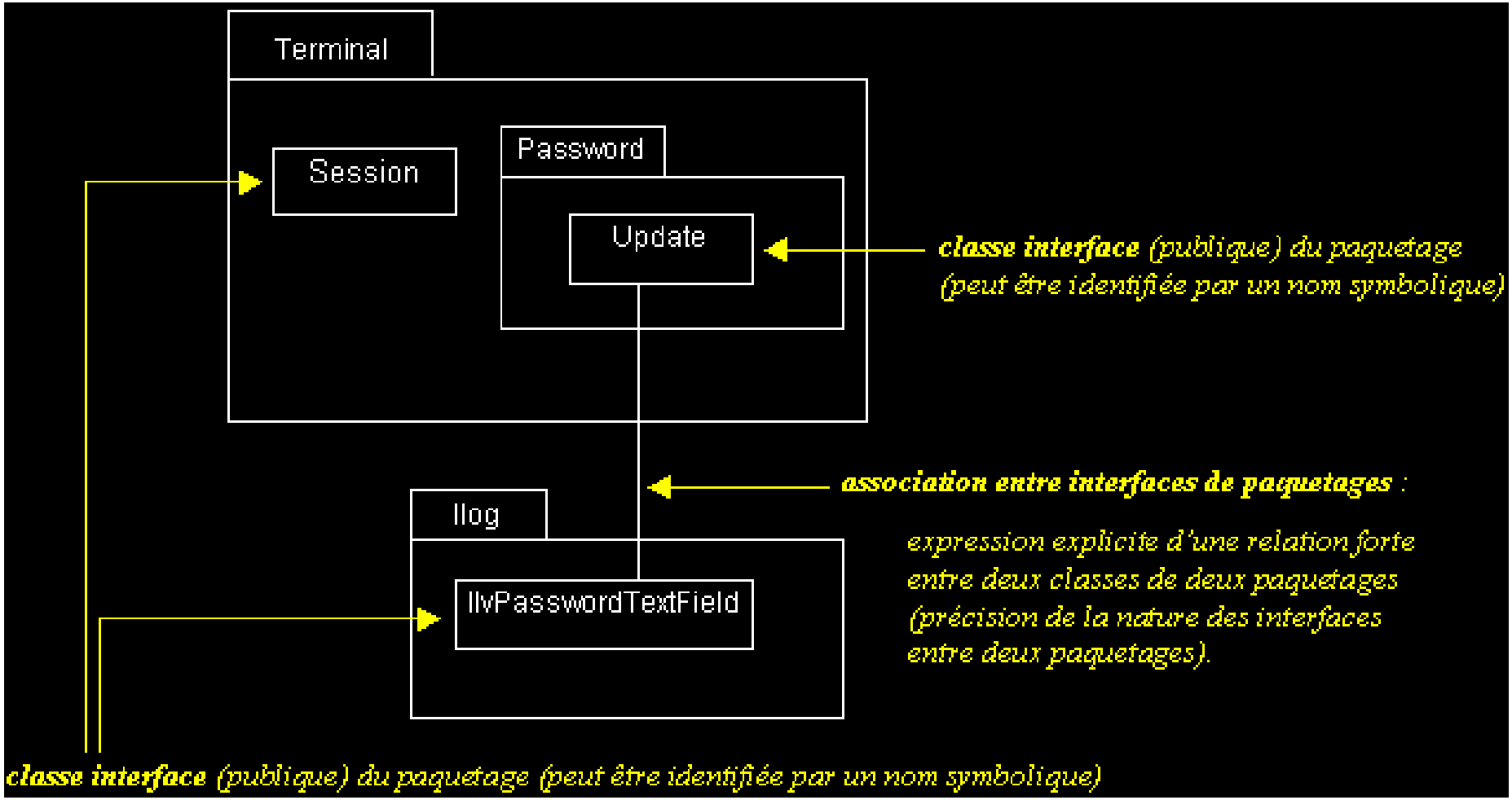
diagrammes de cas d'utilisation



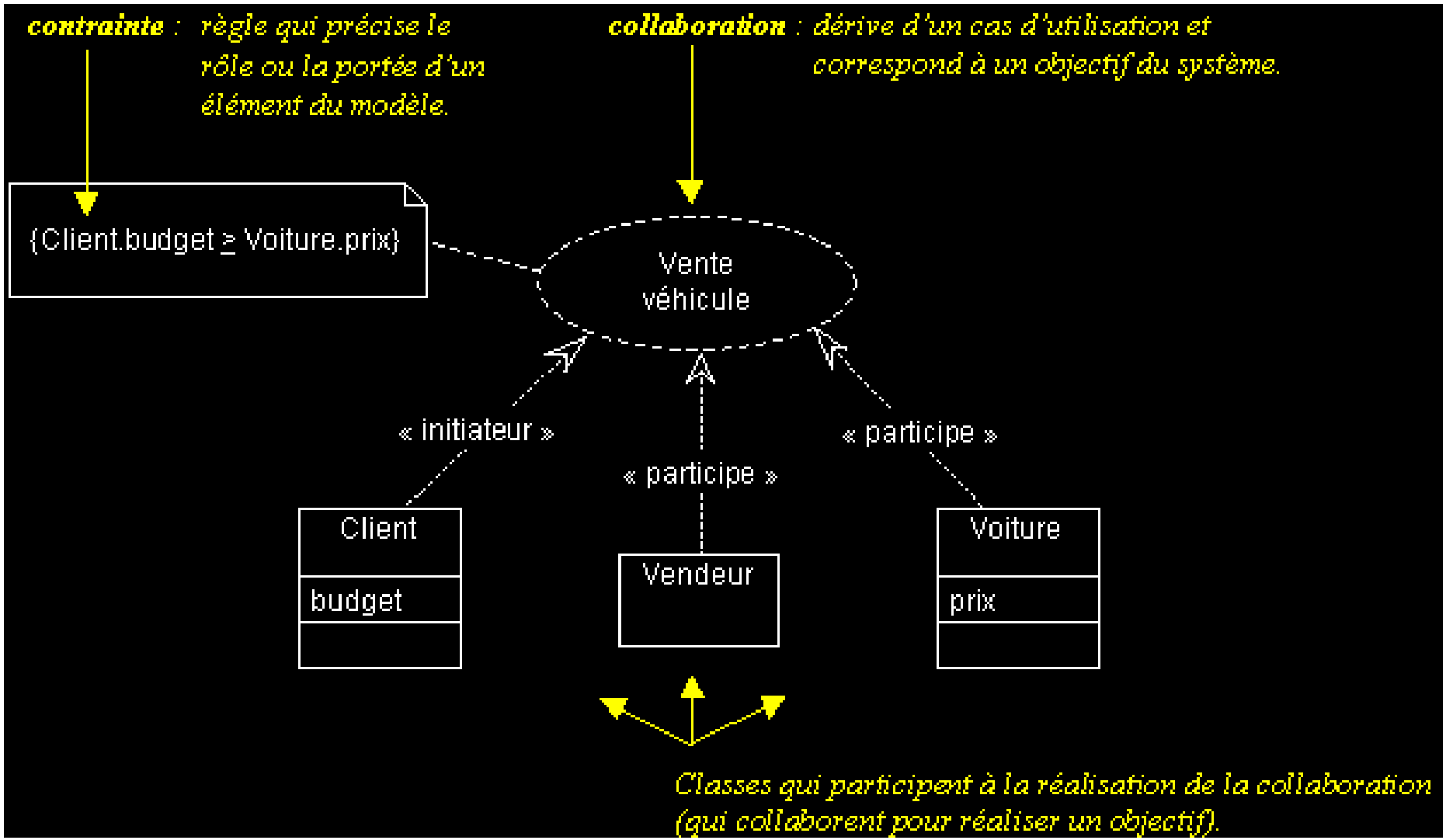
diagrammes de cas d'utilisation



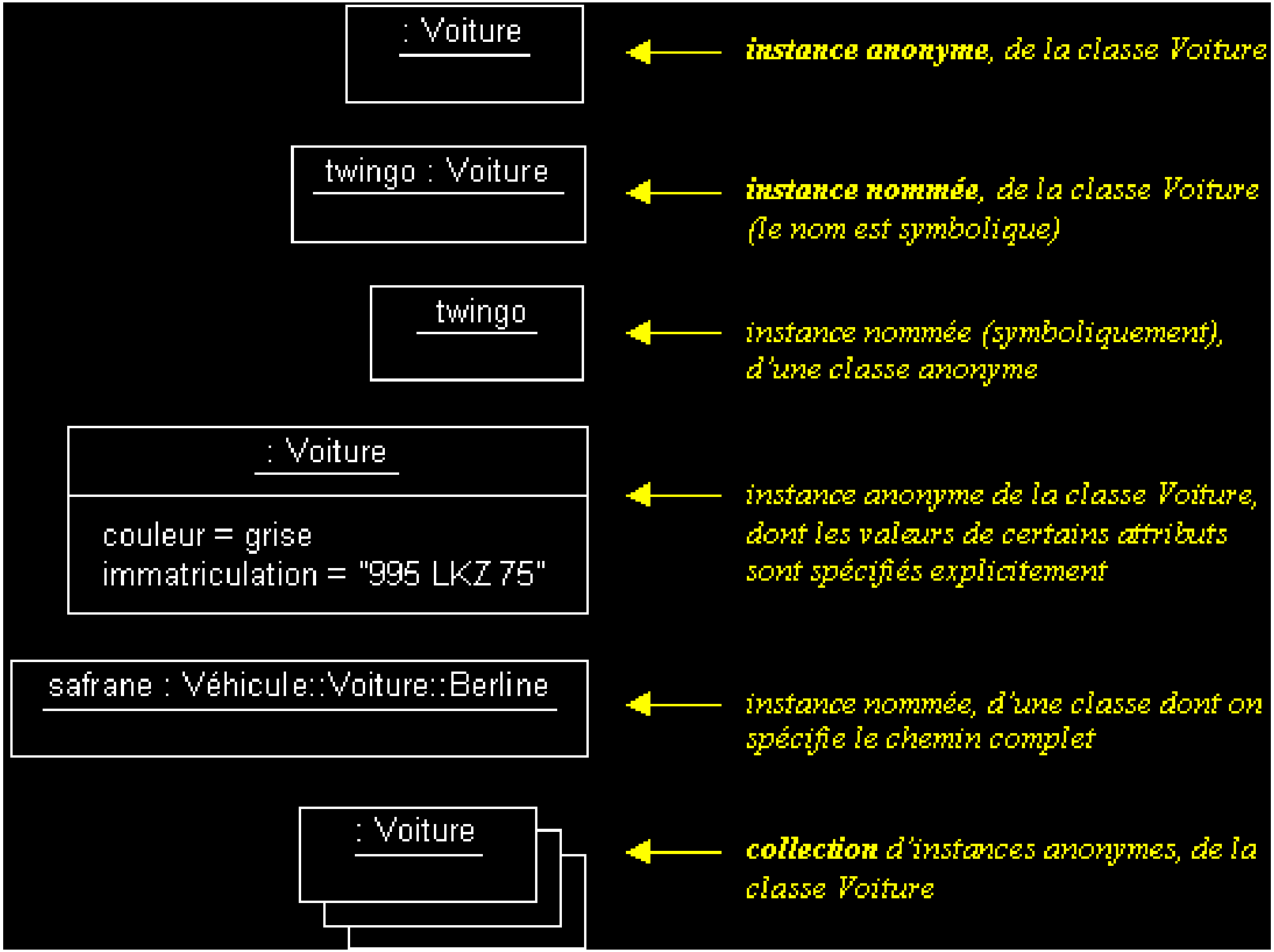
diagrammes de cas d'utilisation



diagrammes de collaboration

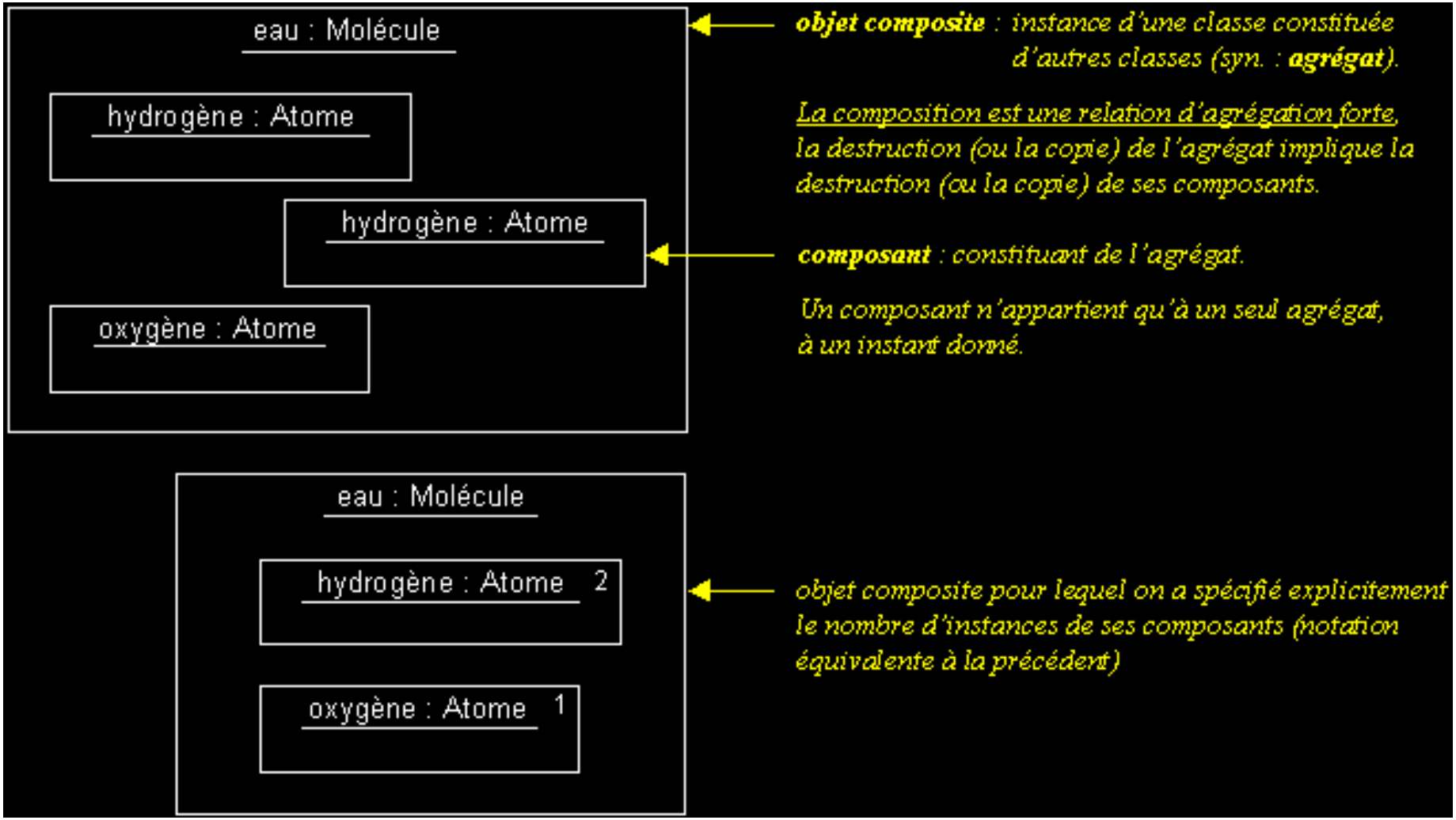


diagrammes d'objets



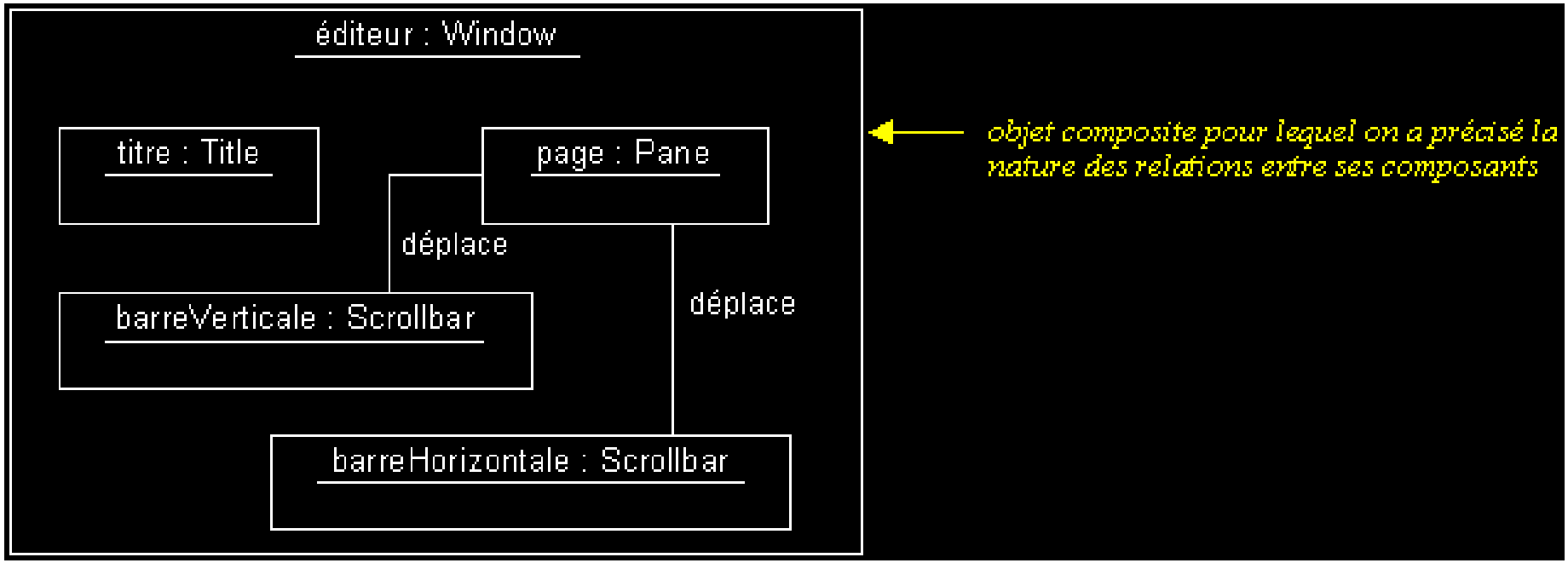
diagrammes d'objets

Objets composites

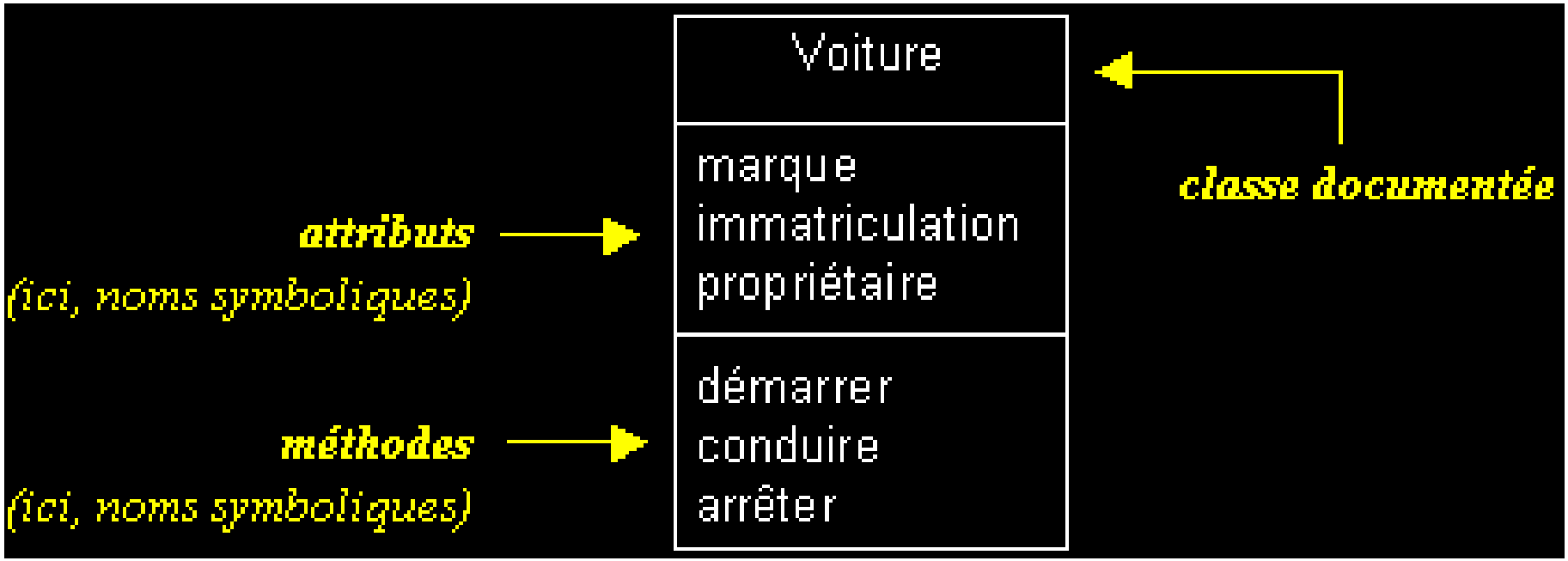


diagrammes d'objets

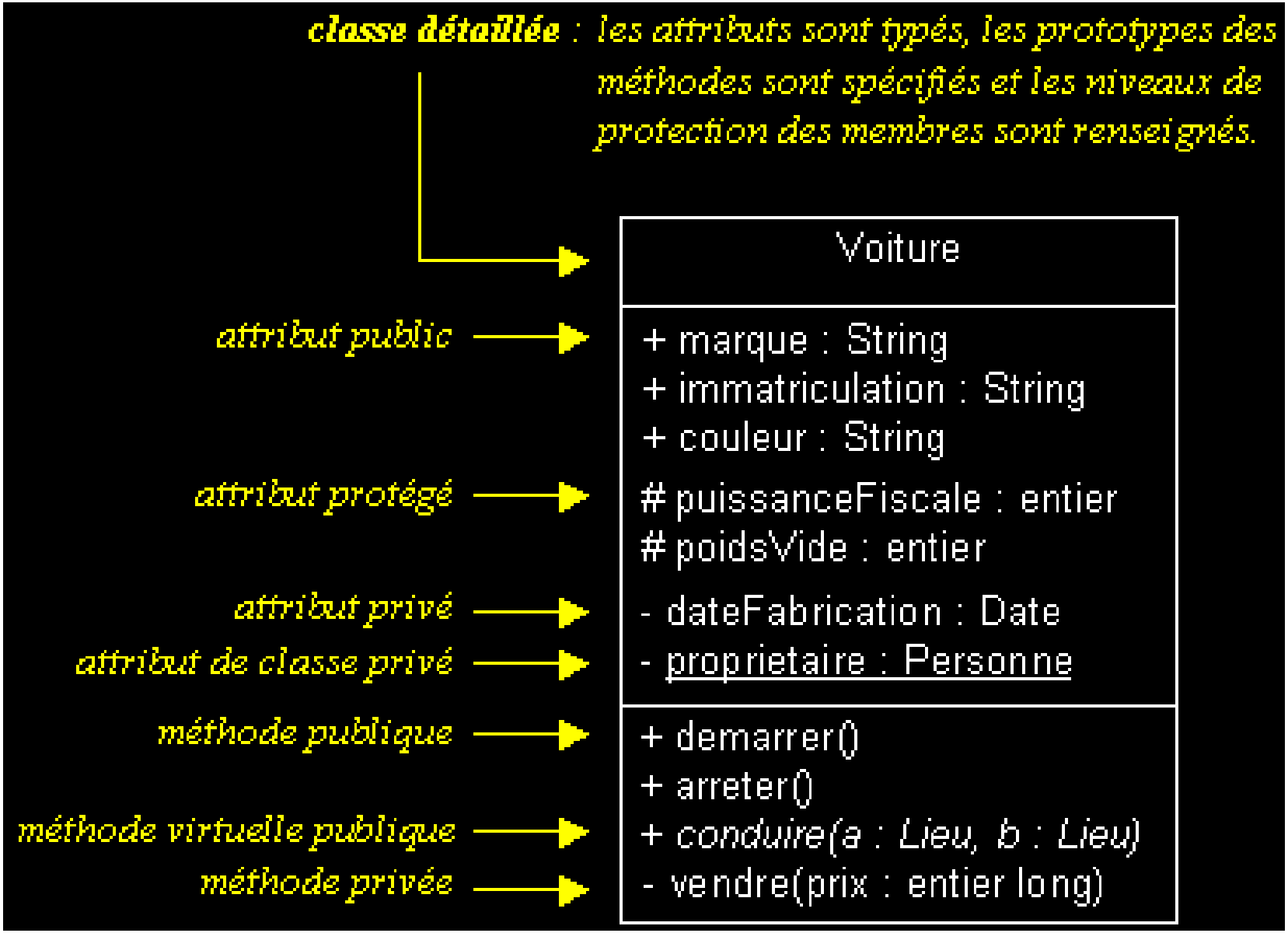
Objets composites



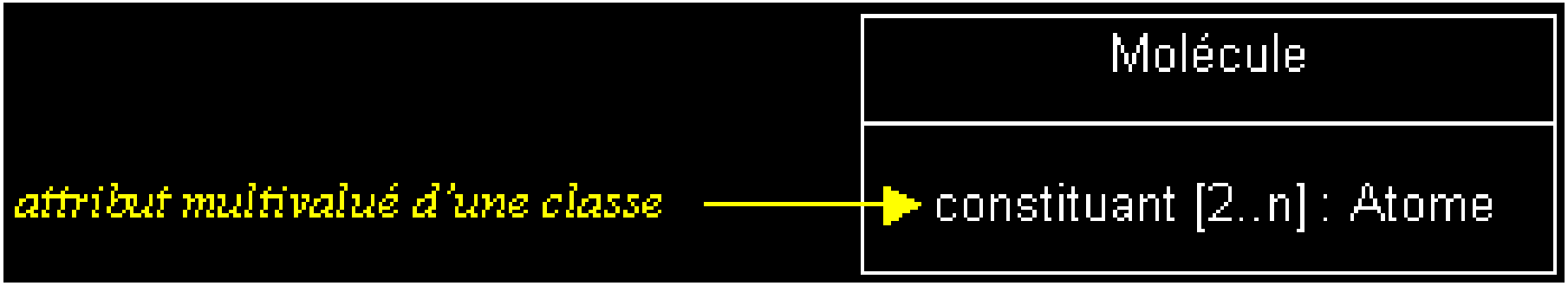
diagrammes de classes

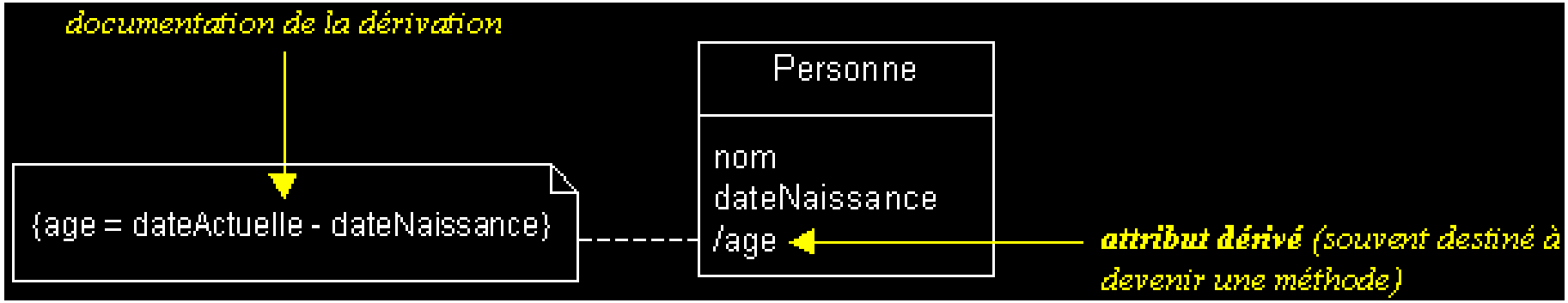


diagrammes de classes



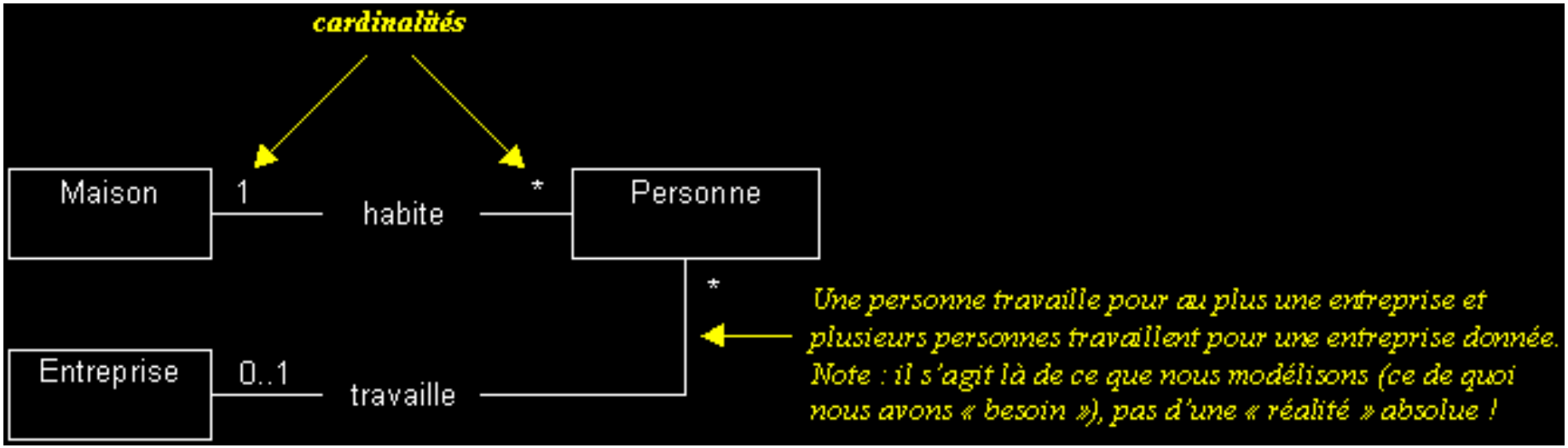
diagrammes de classes





diagrammes de classes

Associations



Expression des cardinalités d'une relation en UML :

n : exactement "n" (n, entier naturel > 0)
exemples : "1", "7"

n..m : de "n" à "m" (entiers naturels ou variables, m ≥ n)
exemples : "0..1", "3..n", "1..31"

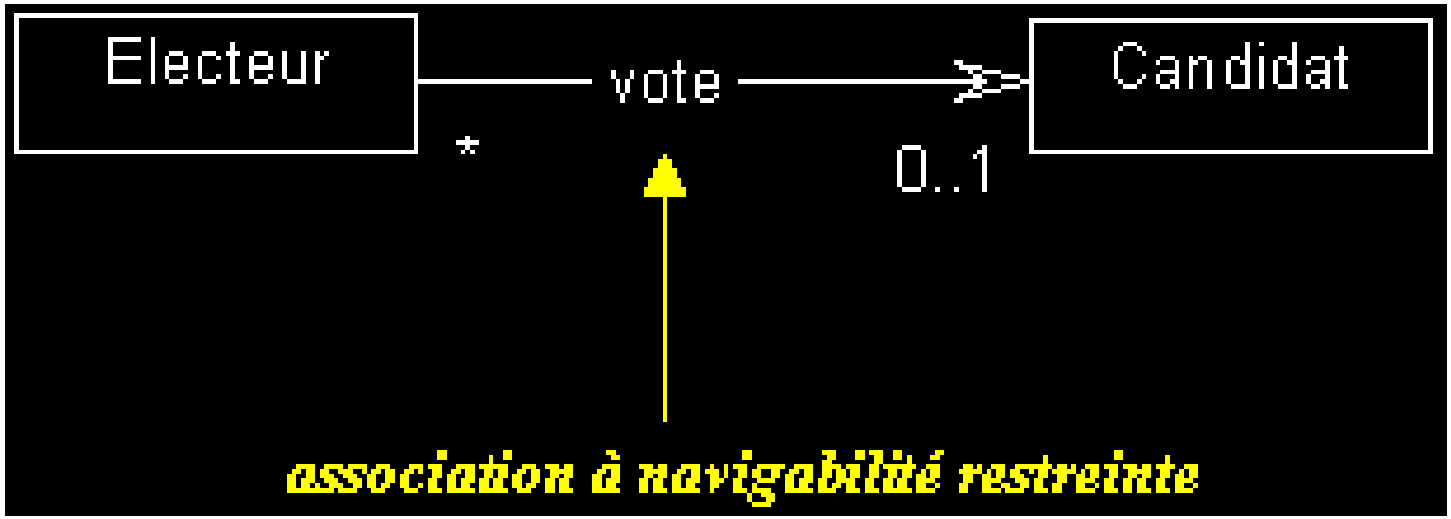
*** :** plusieurs (équivalent à "0..n" et "0..*")

n..* : "n" ou plus (n, entier naturel ou variable)
exemples : "0..*", "5..*"

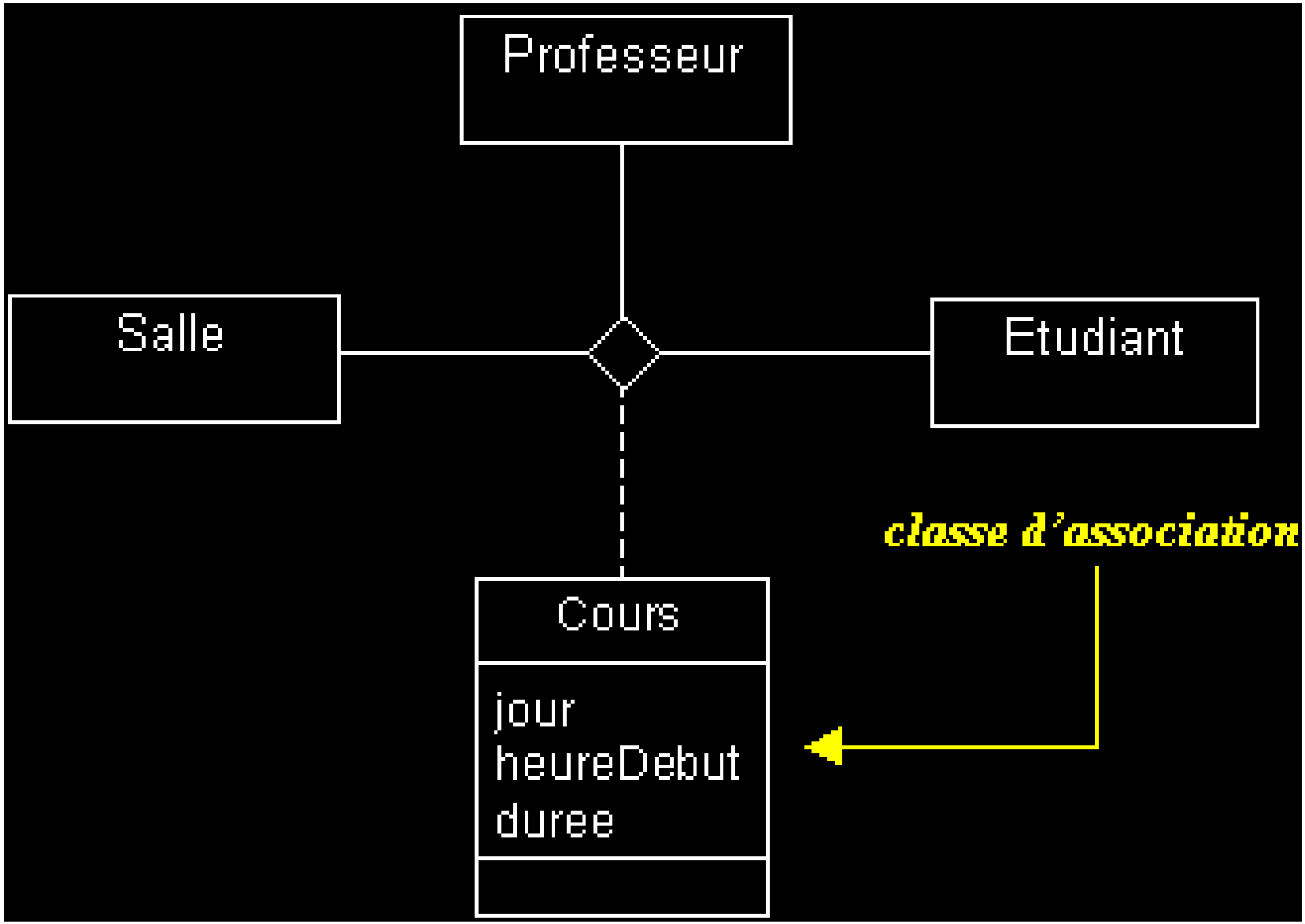
diagrammes de classes

Association à navigabilité restreinte

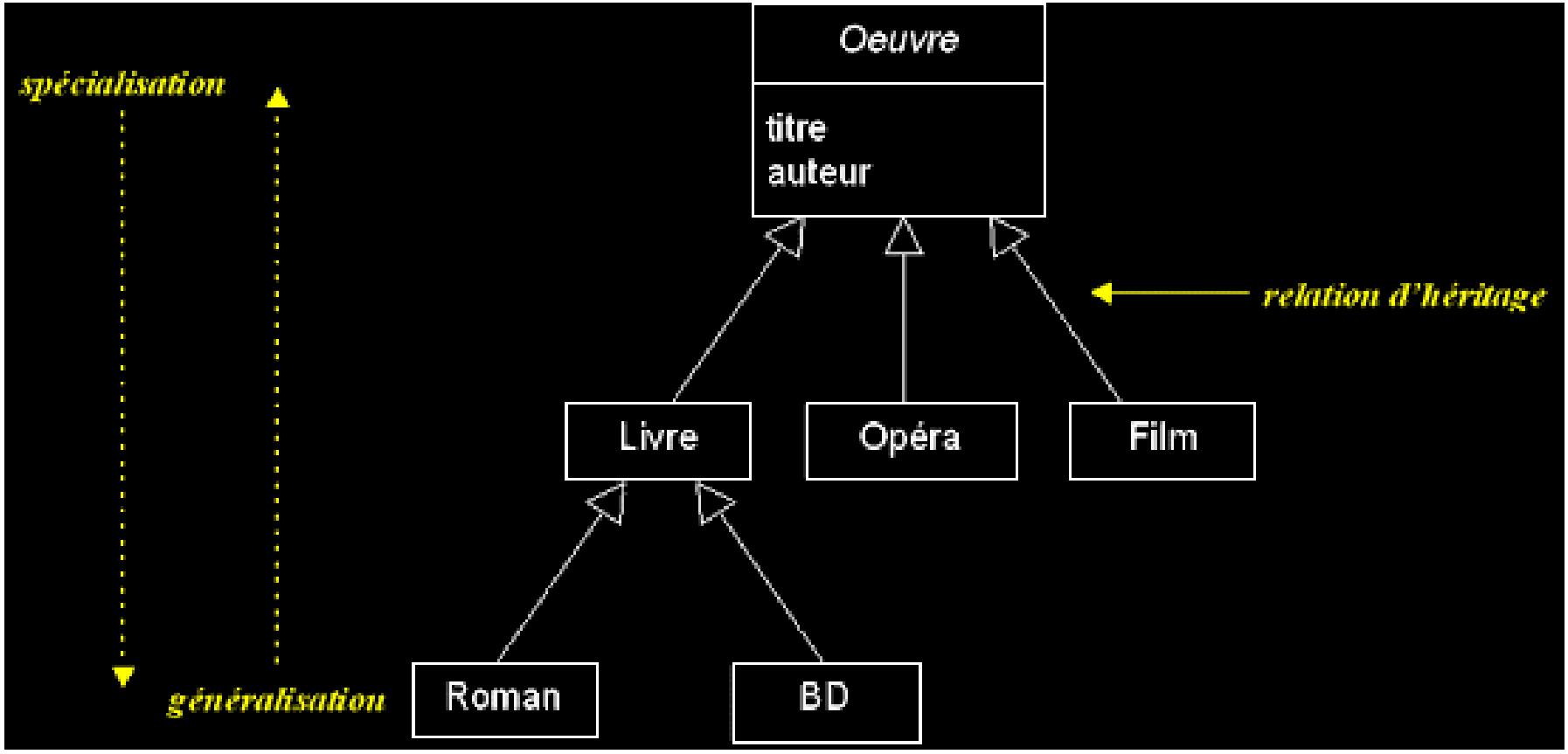
Par défaut, une association est navigable dans les deux sens. La réduction de la portée de l'association est souvent réalisée en phase d'implémentation, mais peut aussi être exprimée dans un modèle pour indiquer que les instances d'une classe ne "connaissent" pas les instances d'une autre.



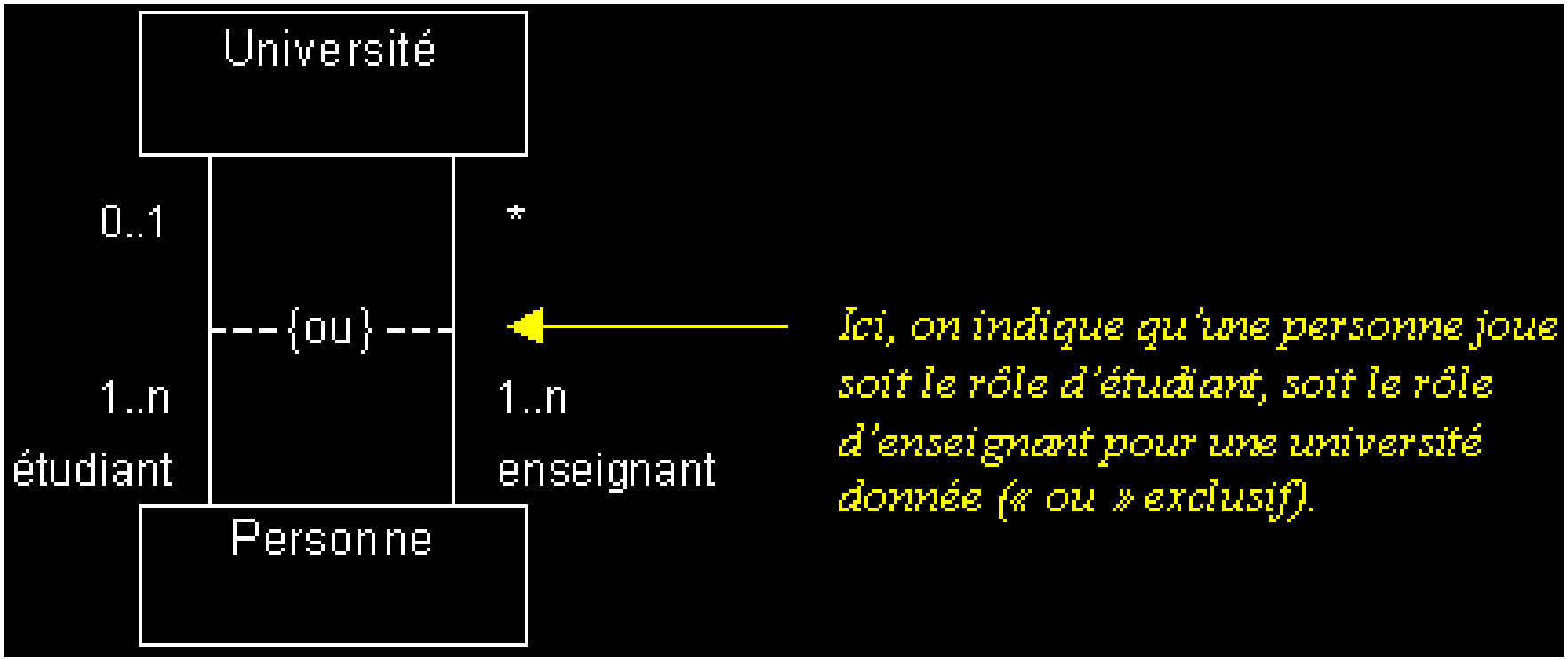
diagrammes de classes



diagrammes de classes



diagrammes de classes



stop

diagrammes de composants

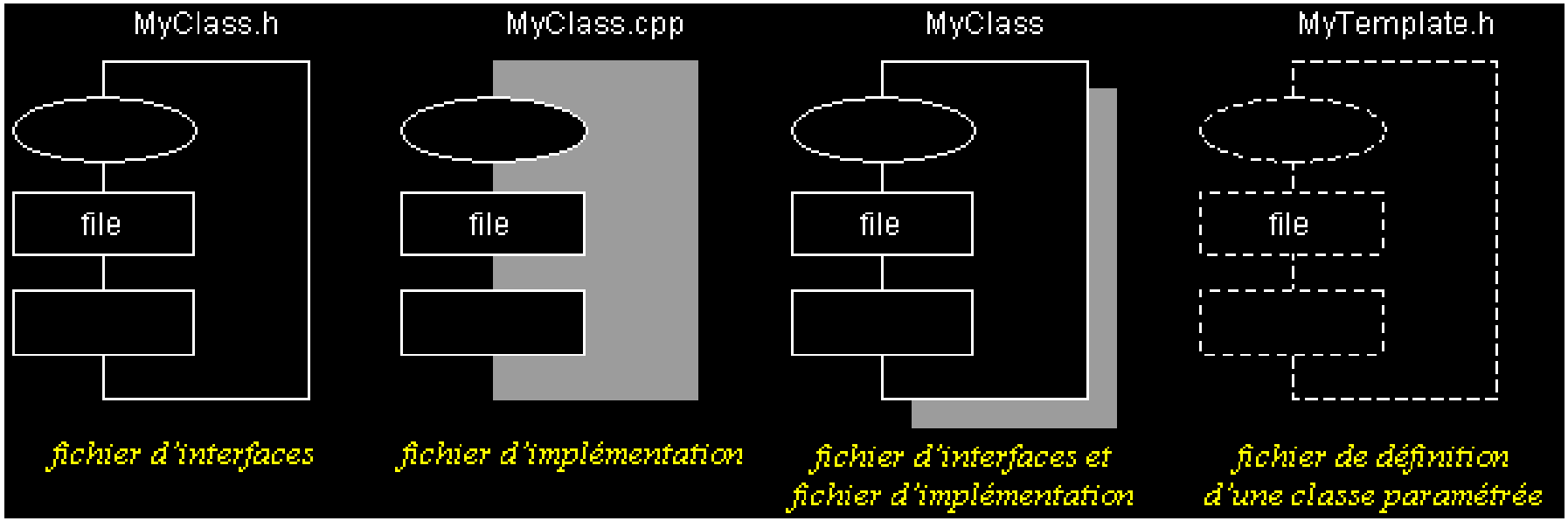
Les diagrammes de composants permettent de décrire **l'architecture physique et statique** d'une application en terme de modules : fichiers sources, librairies, exécutables, etc.

Les dépendances entre composants permettent notamment d'identifier les **contraintes de compilation** et de mettre en évidence la **réutilisation** de composants.

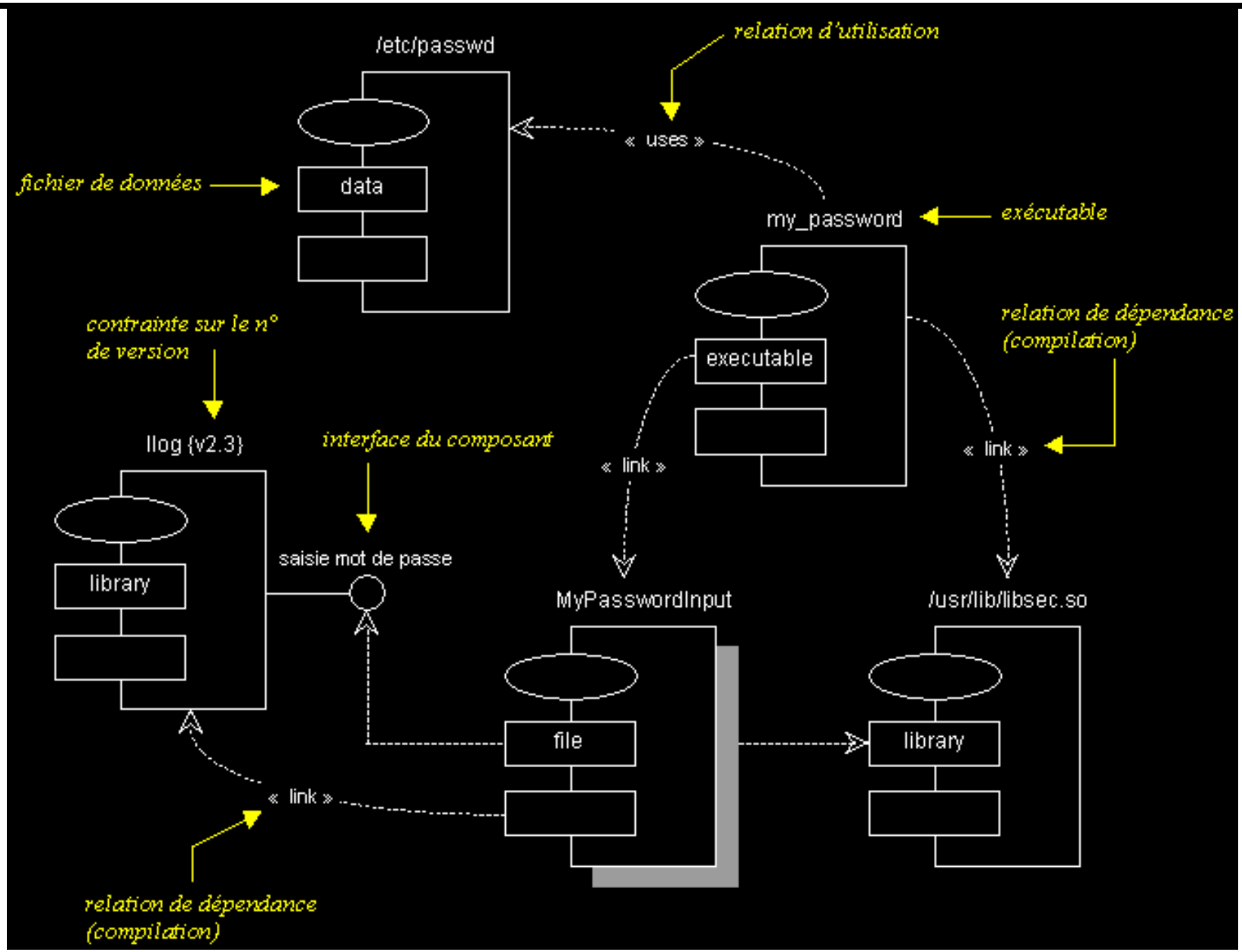
Le composants peuvent être organisés en paquetages, qui définissent des sous-systèmes. Les sous-systèmes organisent la vue des composants (de réalisation) d'un système.

Ils permettent de gérer la complexité, par encapsulation des détails d'implémentation.

diagrammes de composants



diagrammes de composants



diagrammes de déploiement

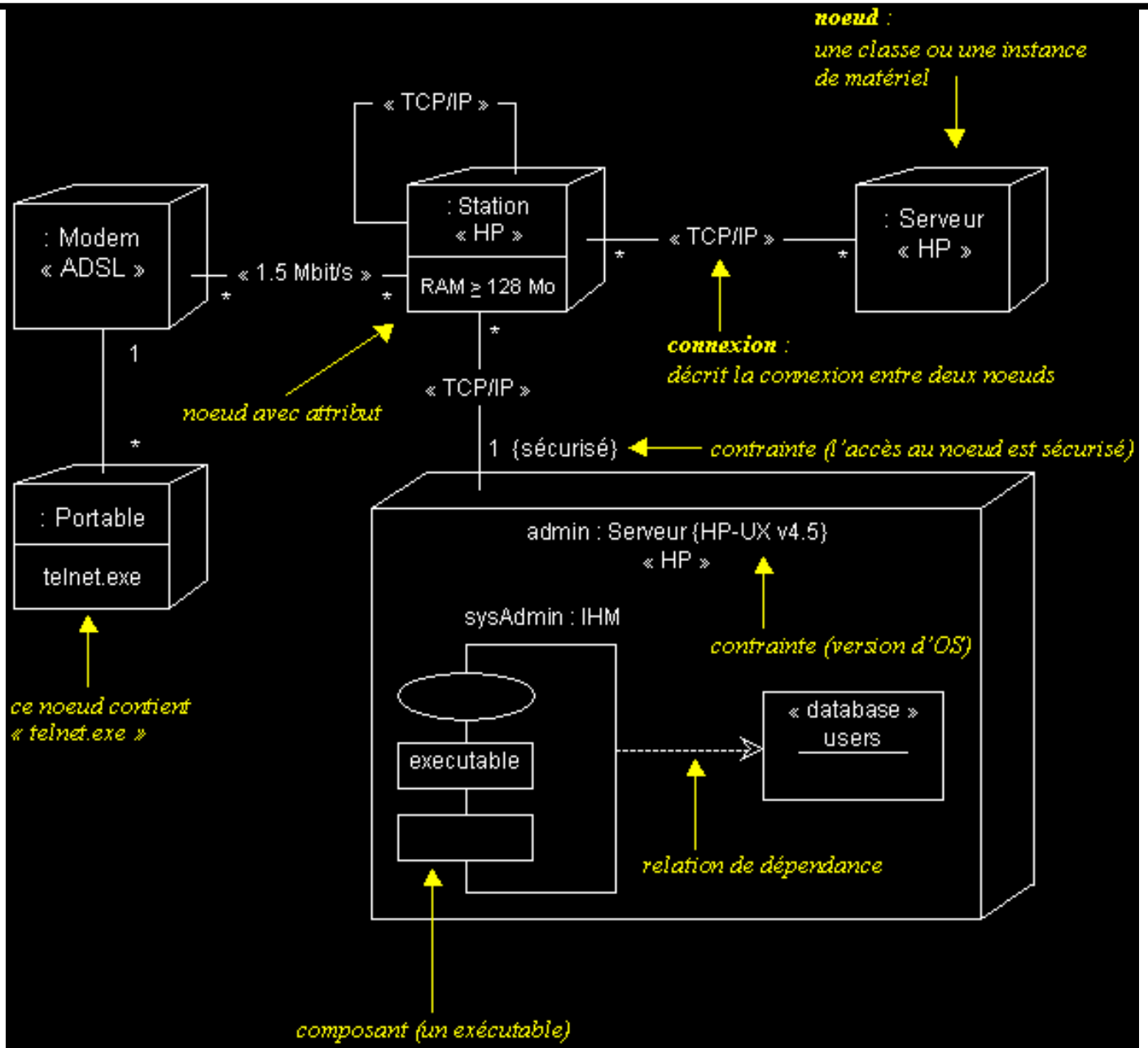
Les diagrammes de déploiement montrent la disposition physique des matériels qui composent le système et la répartition des composants sur ces matériels.

Les ressources matérielles sont représentées sous forme de noeuds.

Les noeuds sont connectés entre eux, à l'aide d'un support de communication. La nature des lignes de communication et leurs caractéristiques peuvent être précisées.

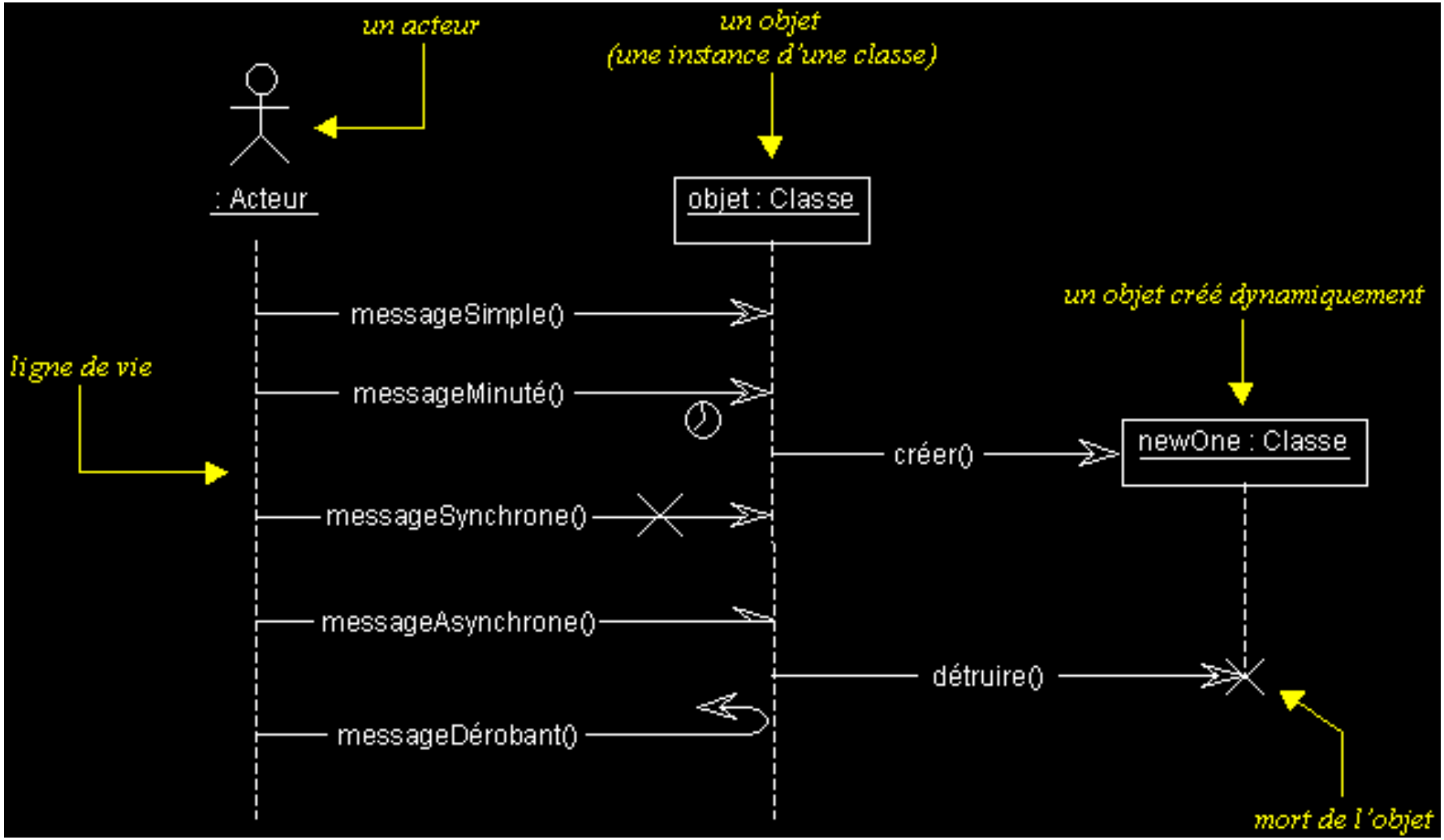
Les diagrammes de déploiement peuvent montrer des instances de noeuds (un matériel précis), ou des classes de noeuds.

diagrammes de déploiement



Modéliser les vues dynamiques d'un système

diagrammes de séquences



diagrammes de séquences

message simple : Message dont on ne spécifie aucune caractéristique d'envoi ou de réception particulière.

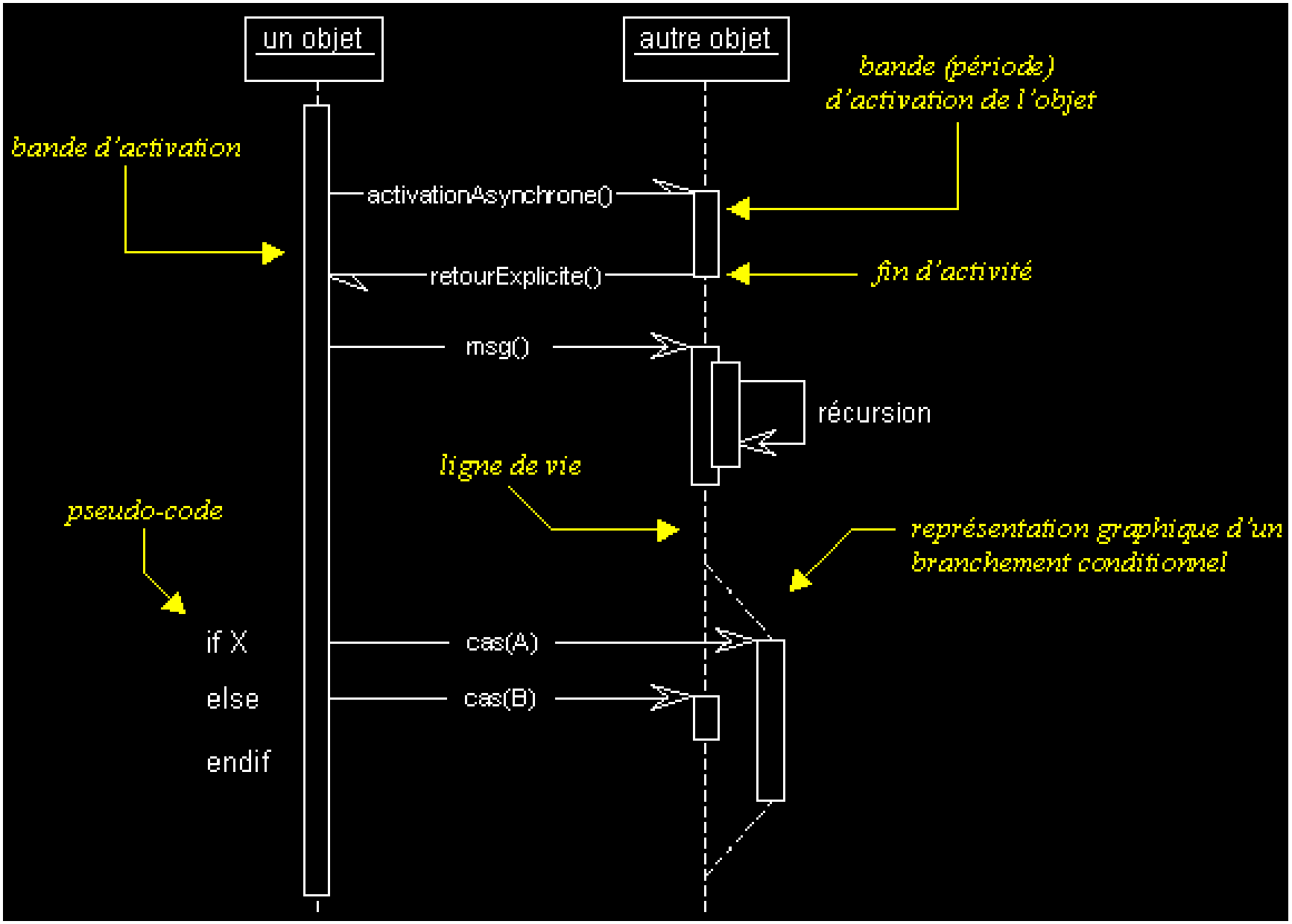
message minuté (timeout) : Bloque l'expéditeur pendant un temps donné (qui peut être spécifié dans une contrainte), en attendant la prise en compte du message par le récepteur. L'expéditeur est libéré si la prise en compte n'a pas eu lieu pendant le délai spécifié.

message synchrone : Bloque l'expéditeur jusqu'à prise en compte du message par le destinataire.

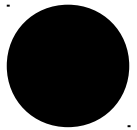
message asynchrone : N'interrompt pas l'exécution de l'expéditeur. Le message envoyé peut être pris en compte par le récepteur à tout moment ou ignoré (jamais traité).

message dérobant : N'interrompt pas l'exécution de l'expéditeur et ne déclenche une opération chez le récepteur que s'il s'est préalablement mis en attente de ce message.

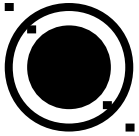
diagrammes de séquences



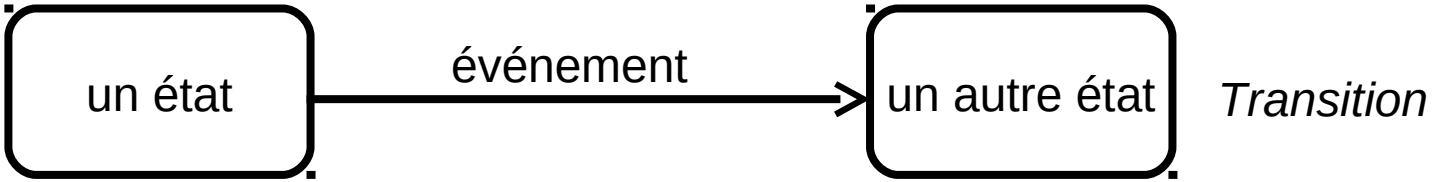
diagrammes d'états-transitions



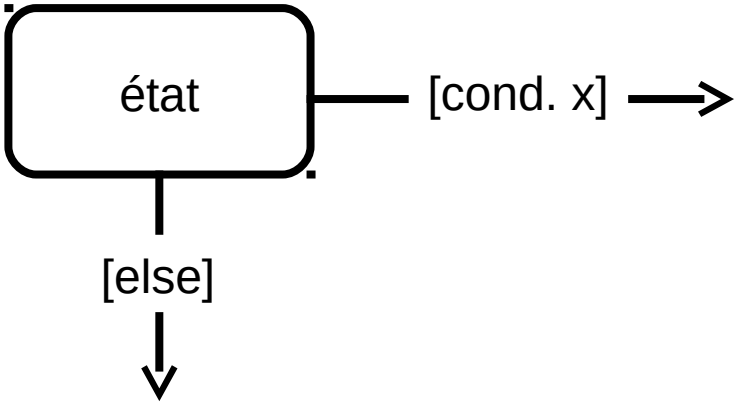
Début du scénario



Fin du scénario

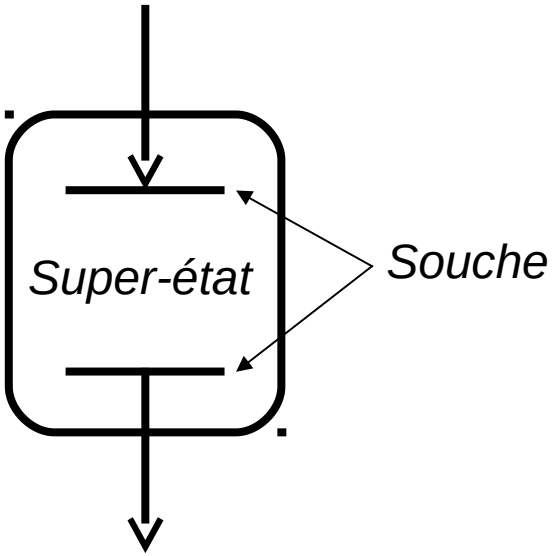


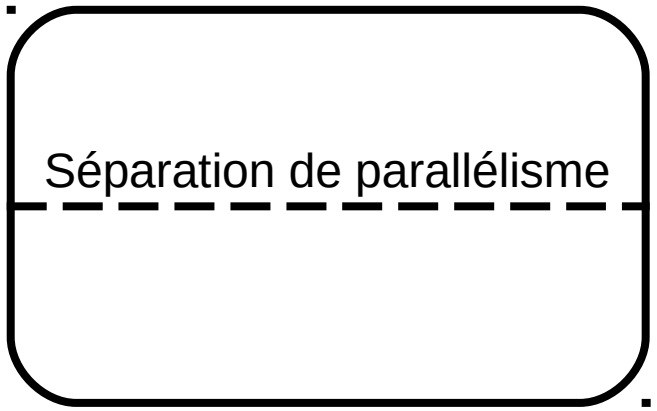
Transition

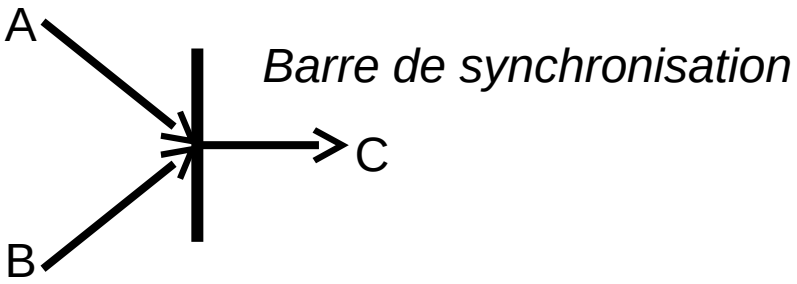


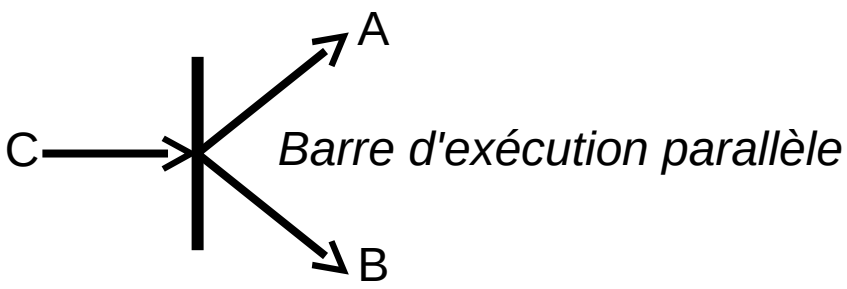
Conditionnelle

diagrammes d'états-transitions

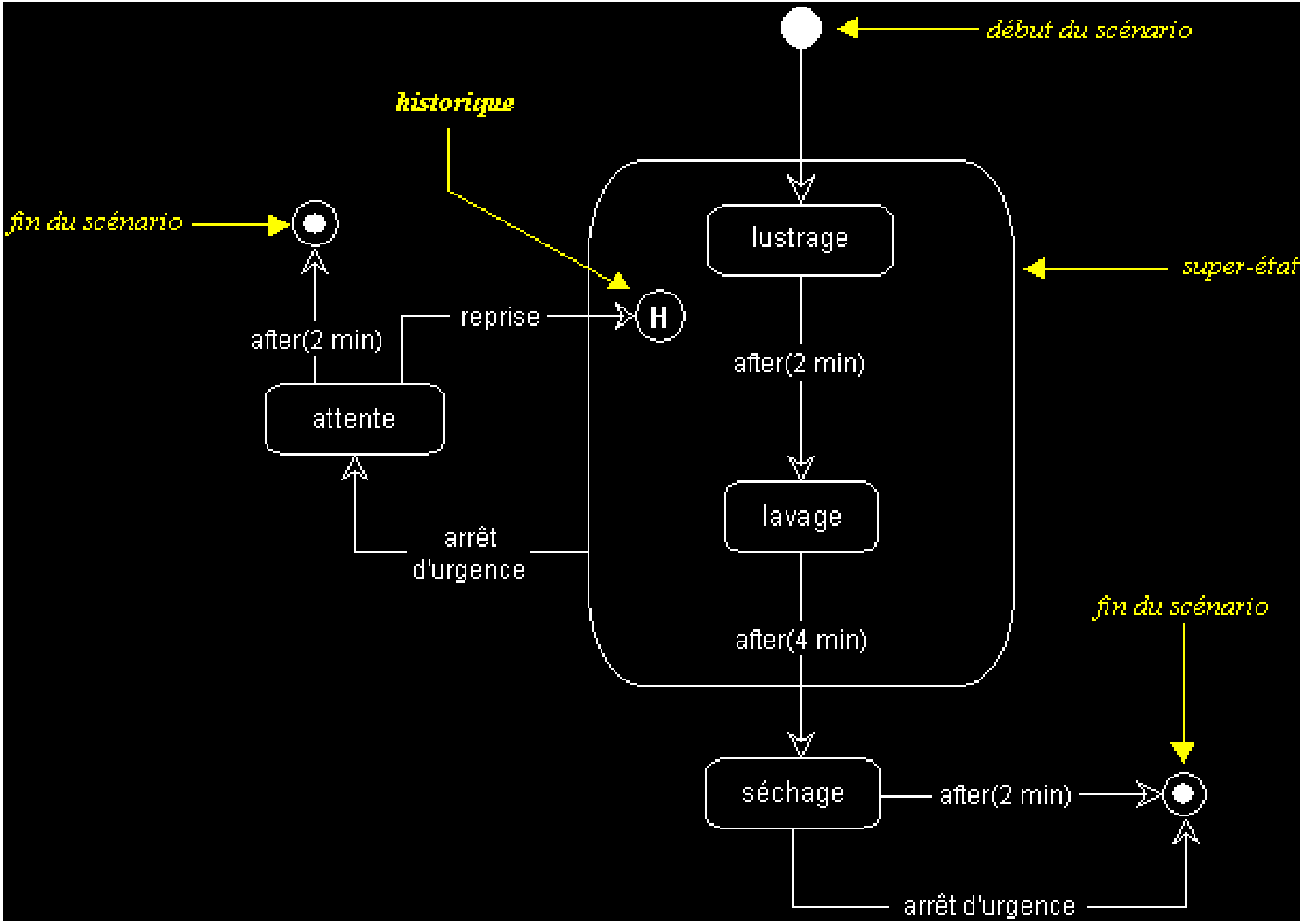




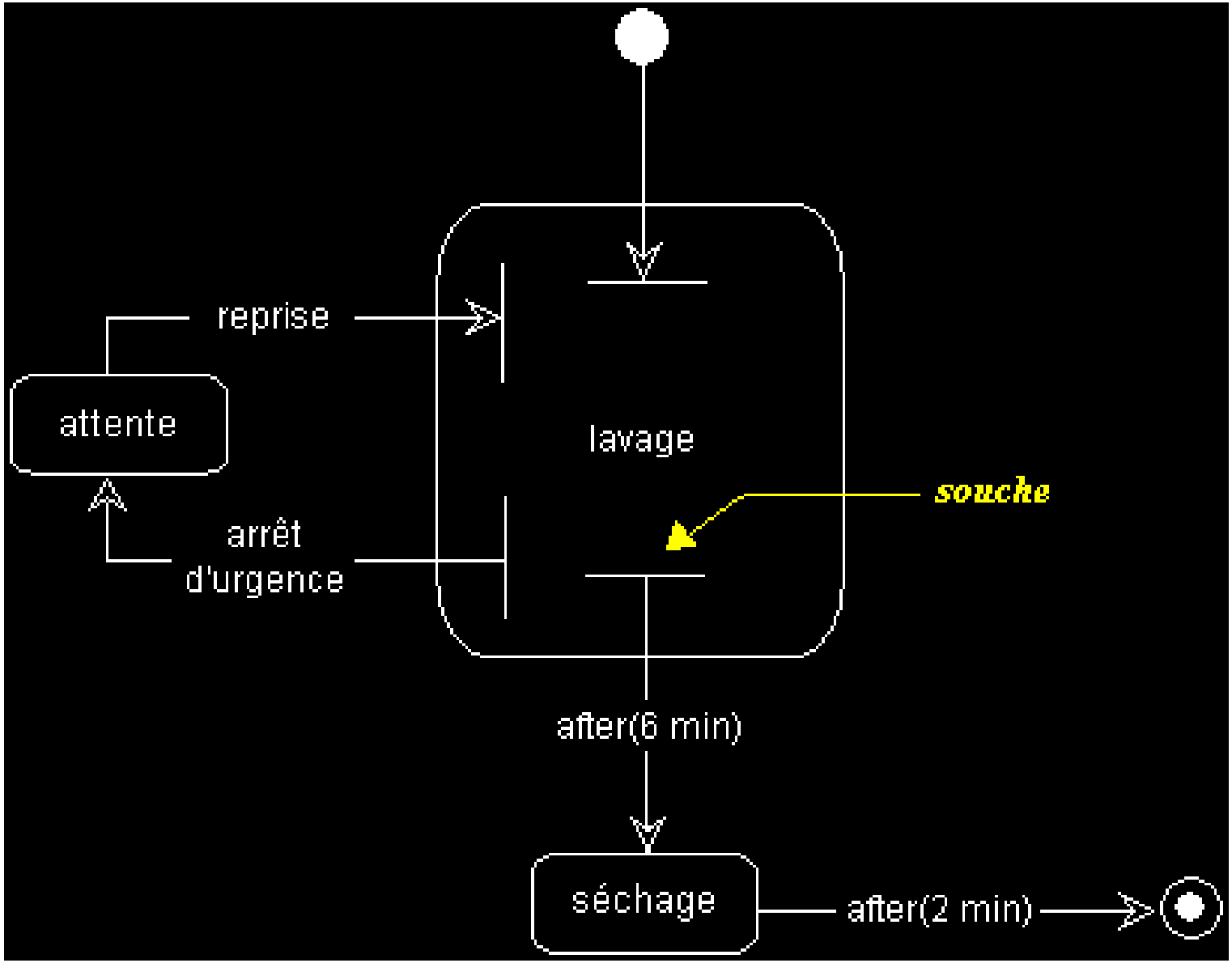




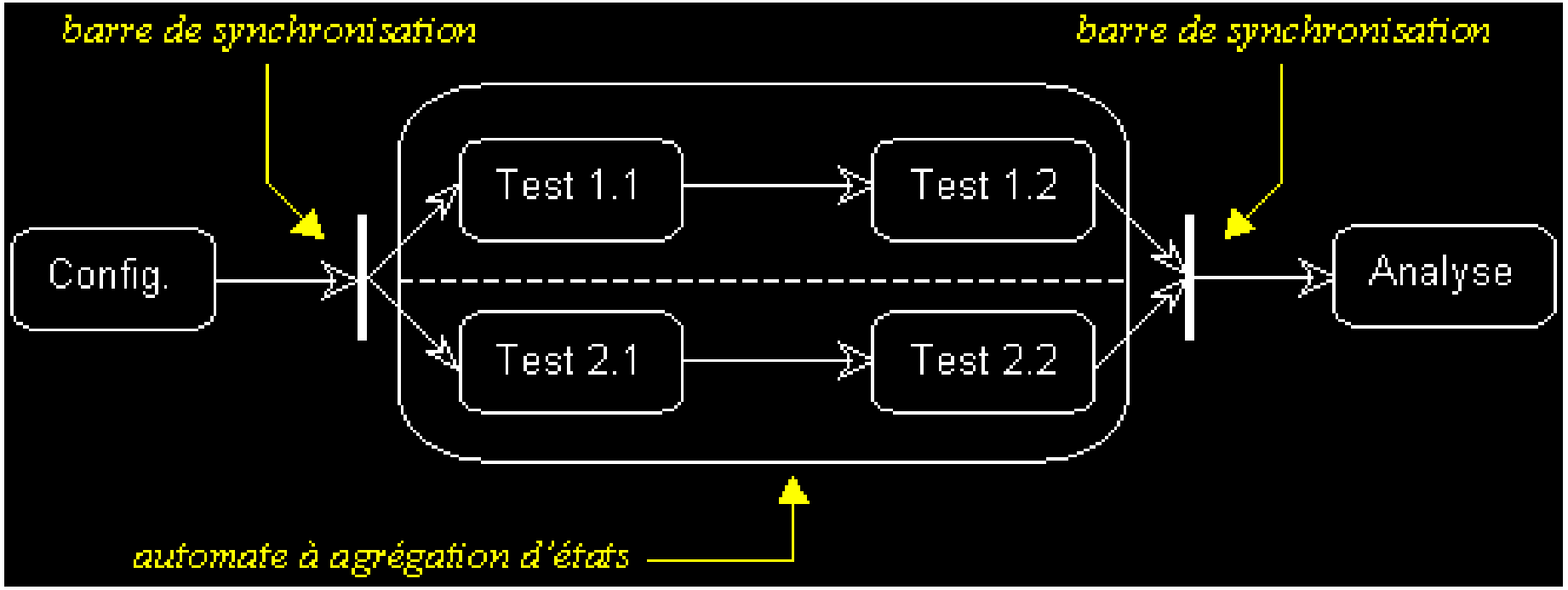
diagrammes d'états-transitions



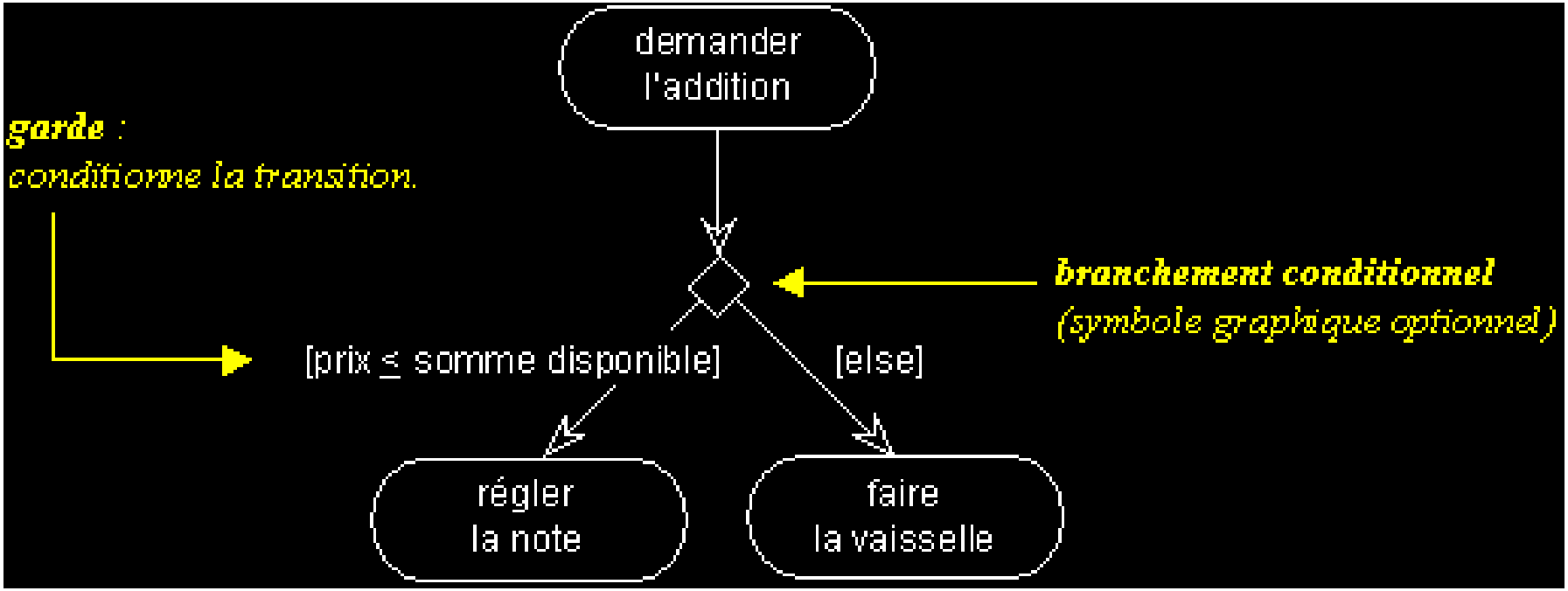
diagrammes d'états-transitions



diagrammes d'états-transitions

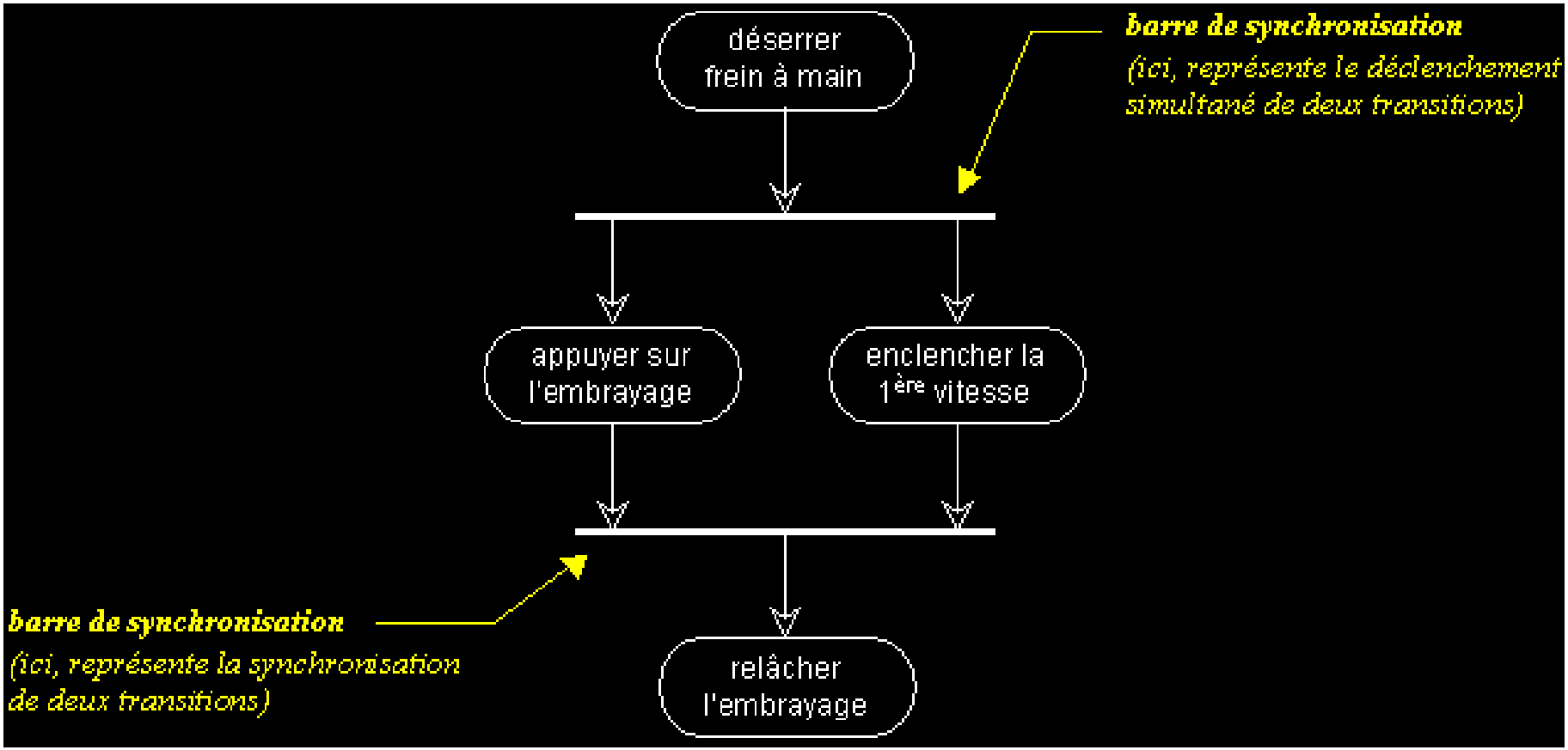


diagrammes d'activités

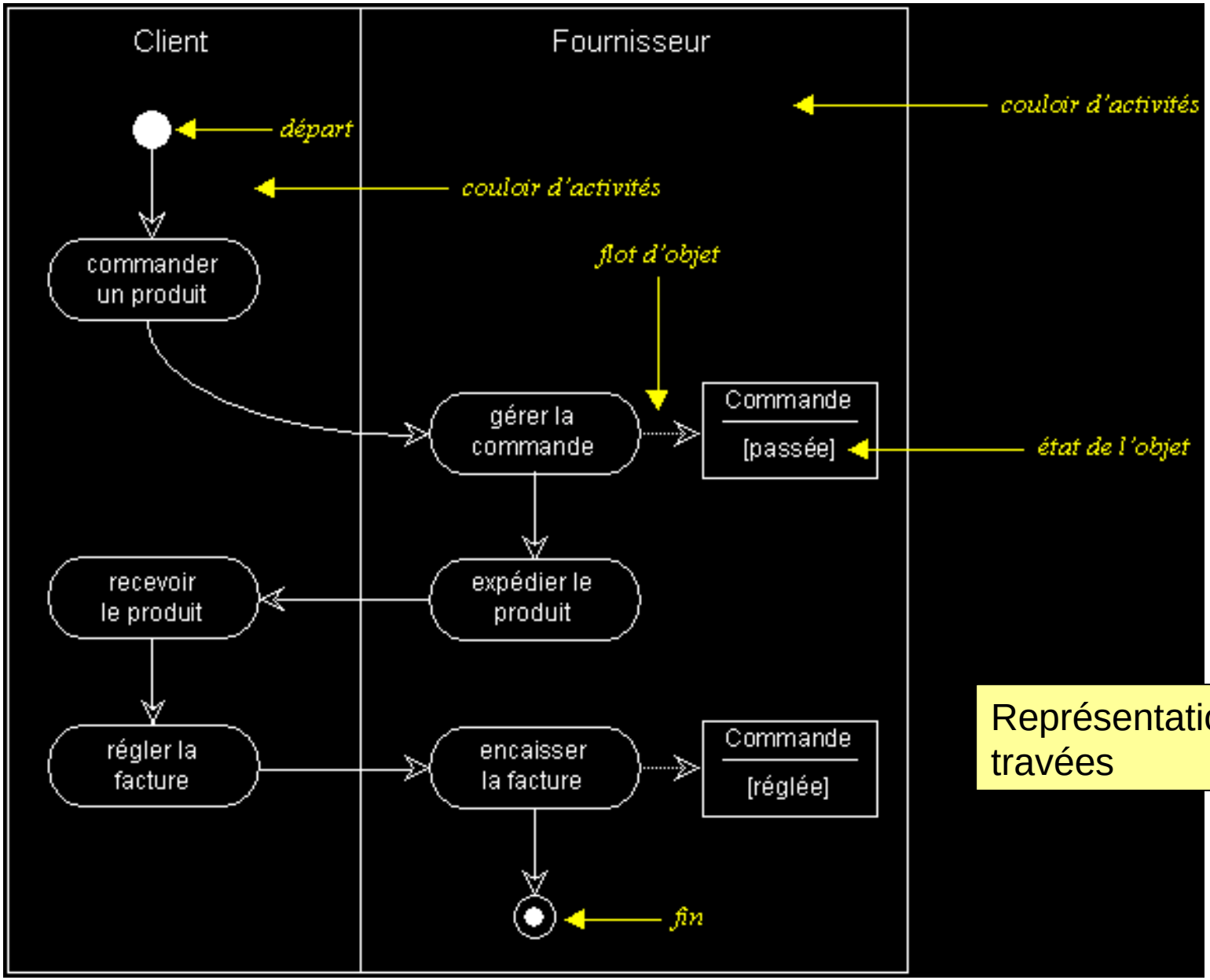


C'est une variante des diagrammes d'états-transitions avec une mise en avant des activités plutôt que des états dans l'automate fini.

diagrammes d'activités

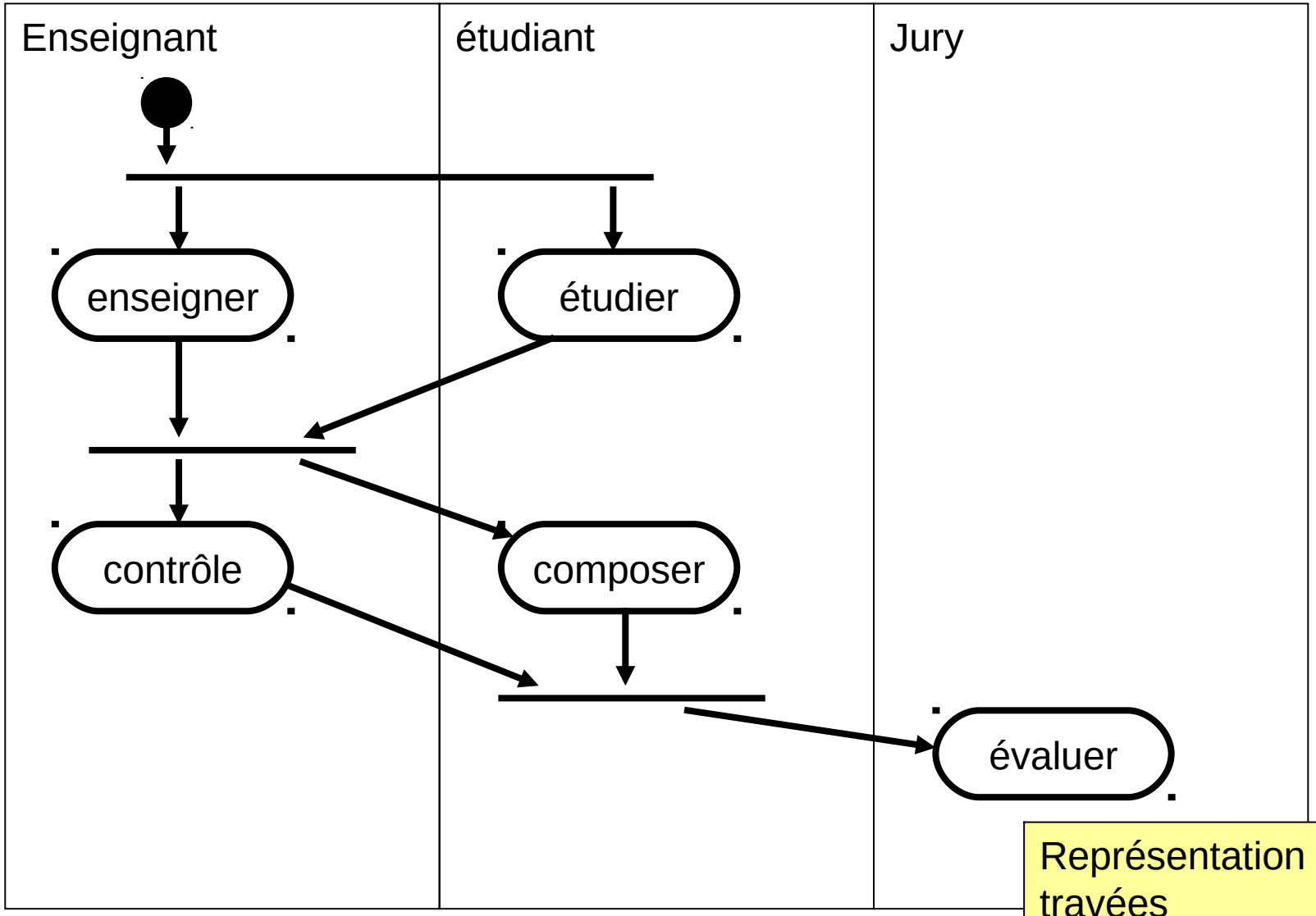


diagrammes d'activités



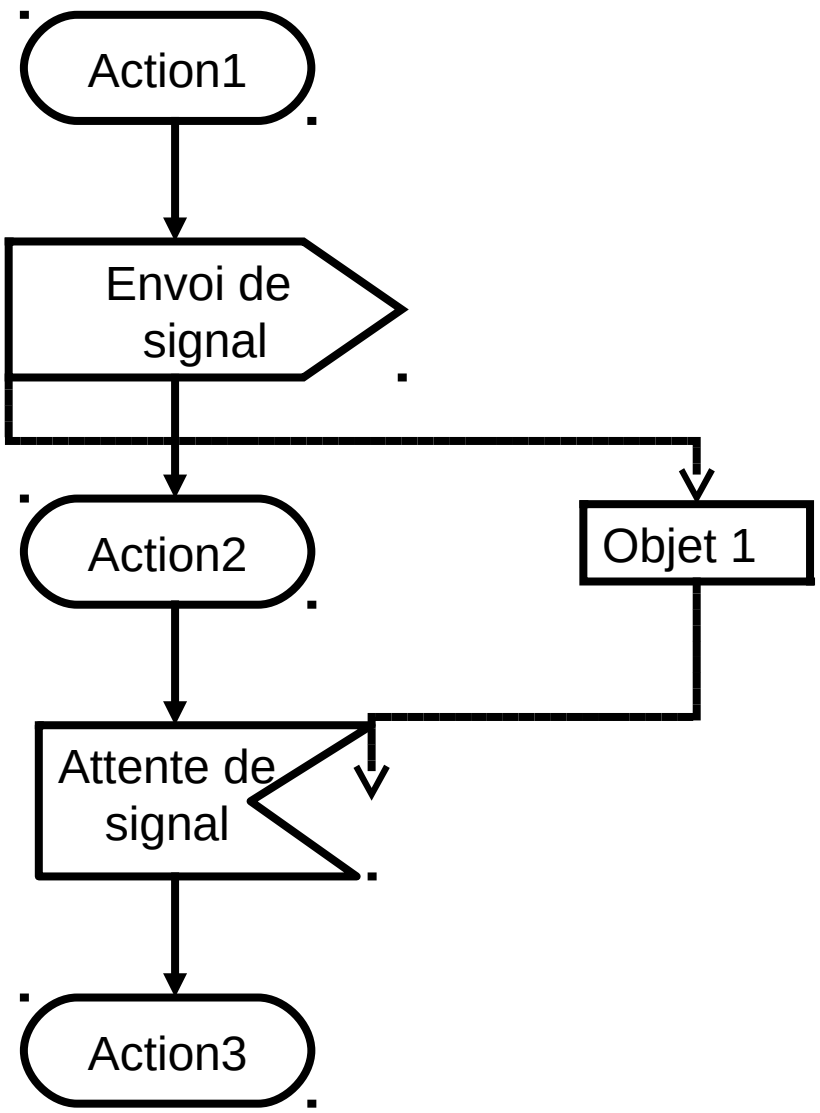
Représentation avec travées

diagrammes d'activités

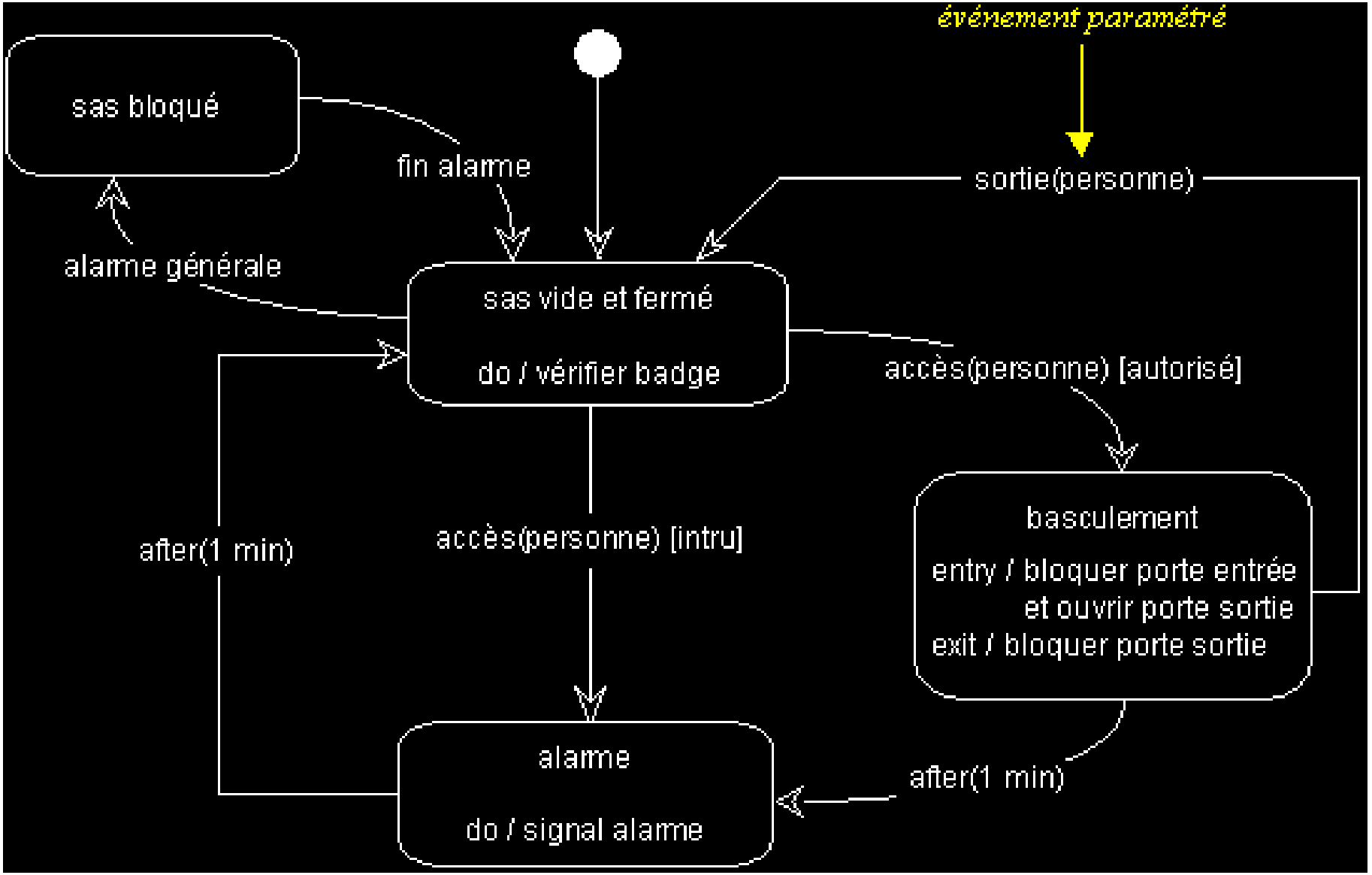


Représentation avec travées

diagrammes d'activités



diagrammes d'activités



Conclusion

Programmer objet ?

- **Programmer en Java n'est pas "concevoir objet" !**

  Seule une analyse objet conduit à une solution objet.

  Le langage de programmation est un moyen d'implémentation, il ne garantit pas le respect des concepts objet.

- **UML n'est qu'un support de communication !**

  L'utilisation d'UML ne garantit pas le respect des concepts objet : à vous de vous en servir à bon escient.

Conclusion

Utiliser UML ?

- **Multipliez les vues sur vos modèles !**

  Un diagramme n'offre qu'une vue très partielle et précise d'un modèle.

  Croisez les vues complémentaires (dynamiques / statiques).

- **Restez simple !**

  Utilisez les niveaux d'abstraction pour synthétiser vos modèles (ne vous limitez pas aux vues d'implémentation).

  Ne surchargez pas vos diagrammes.

  Commentez vos diagrammes (notes, texte...).

- **Utilisez des outils appropriés pour réaliser vos modèles !**

Les outils

www.objecteering.com

http://pyut.sourceforge.net

http://argouml.tigris.org/

http://www.gentleware.com