MODÉLISATION UML

N. FACI

MODÉLISATION UML

- **Objectifs de ce cours**
  - Comprendre comment représenter les vues statiques et dynamiques des systèmes (Diagrammes UML)
  - Maîtriser comment élaborer les modèles de ces vues
- **Organisation**
  - 44h Cours/TD
- **Modalités d'évaluation**
  - 1h*2 Interro, 1h30 DS

noura.faci@univ-lyon1.fr

UML EN BREF

- UML est une **notation graphique** permettant de **représenter des modèles** mais ne définit pas le **processus d'élaboration des modèles**.
- Un modèle est une abstraction de la réalité qui permet de mieux comprendre le système à développer.
- Les diagrammes constituent un outil de communication pour faire véhiculer des idées sur la conception de votre futur système.

LES DIAGRAMMES UML (1/2)

VUE STATIQUE

- **Le diagramme de Cas d'Utilisation (DCU)**
  - permet de capturer les exigences fonctionnelles d'un système, et les interactions types entre les utilisateurs d'un système (les acteurs) et le système lui-même.

- **Les diagrammes de classes (DCL) et d'Objets (DOB)**
  - Un diagramme de classes décrit les types d'objets qui composent un système et les différents types de relations statiques qui existent entre eux, représente les propriétés et les opérations des classes

LES DIAGRAMMES UML (2/2)

VUE DYNAMIQUE

- **Les diagrammes d'interaction** (e.g., diagrammes de séquence et de communication)
  - décrivent la façon dont des groupes d'objets collaborent pour la réalisation d'un comportement donné.

- **Le diagramme d'Etat-Transition (DET)**
  - permet de décrire le comportement d'un système.

- **Le diagramme d'activités (DIT)**
  - permet de décrire la logique procédurale, les processus métiers et les enchaînements d'activités (workflows).

Diagramme de cas d'Utilisation (DCU)

DIAGRAMME DE CAS D'UTILISATION

Les cas d'utilisation permettent de:

- Décrire le comportement d'un système d'un point de vue de l'utilisateur,
- Représenter les grandes fonctionnalités du système sans spécifier comment ces fonctionnalités seront réalisées :

  → Description fonctionnelle de haut niveau (description du QUOI et non du COMMENT)

Diagramme de cas d'Utilisation

Le diagramme de cas d'utilisation définit un système en spécifiant:

- Les fonctions métiers principales du système: ***les cas d'utilisation*** (en anglais, use cases)
- Les ***acteurs*** qui interagissent avec les use cases
- Les ***relations*** (ou associations) entre acteur- use case, acteur-acteur, et use case- use case

ELÉMENTS DE BASE

Système

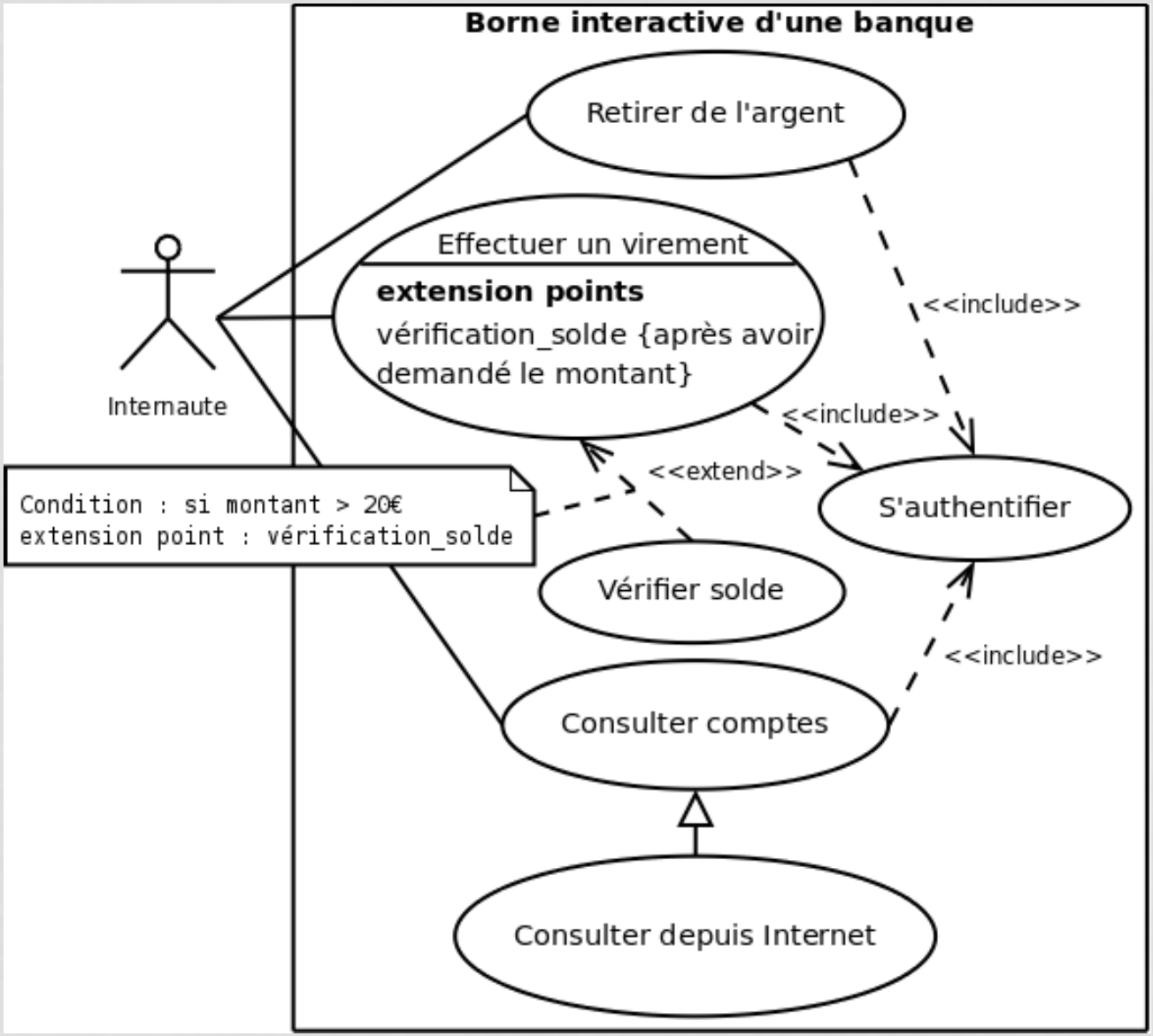
Acteur

Cas d'utilisation

Système définit l'application informatique, il contient les use cases et leurs associations.

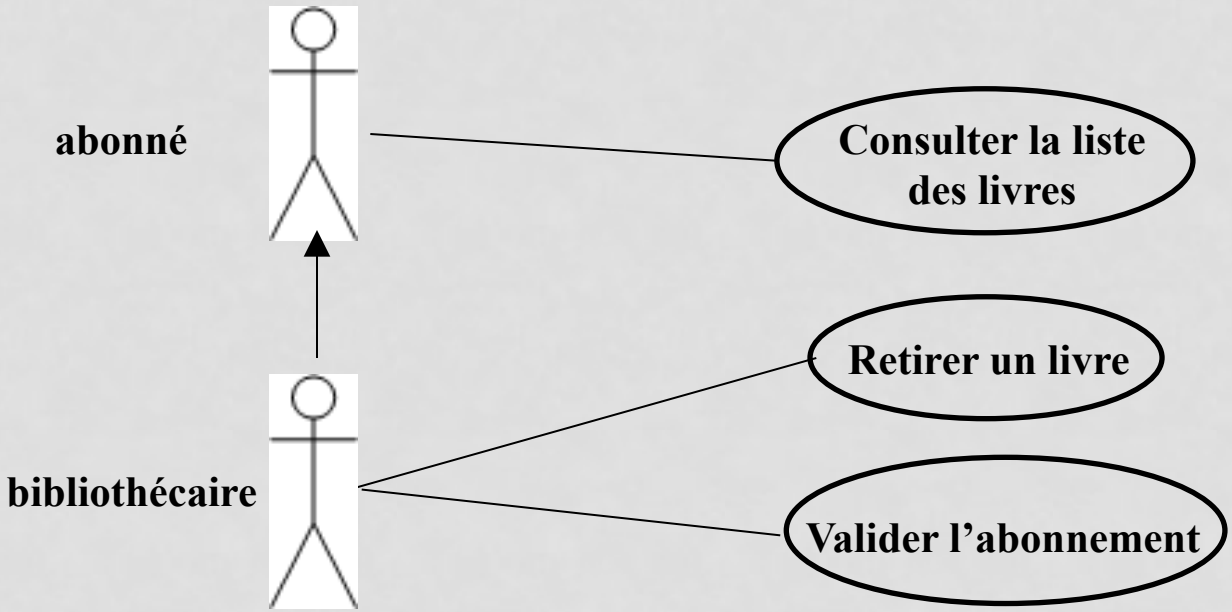
Acteur représente une personne ou un périphérique jouant un rôle avec le système.

Use case représente une fonctionnalité du système visible de l'extérieur et utilisable par un acteur. Son nom est un verbe à l'infinitif.


Borne interactive d'une banque
Retirer de l'argent
Effectuer un virement
extension points
vérification_solde {après avoir demandé le montant}
<<include>>
Internaute
<<include>>
<<extend>>
Condition : si montant > 20€
extension point : vérification_solde
S'authentifier
Vérifier solde
<<include>>
Consulter comptes
Consulter depuis Internet

RELATION ENTRE ACTEURS

- Généralisation (héritage)



CAS D'UTILISATION

- Il exprime une suite d'interactions entre un acteur et le système,
- Correspond à
  - un scenario nominal (déroulement sans erreurs),

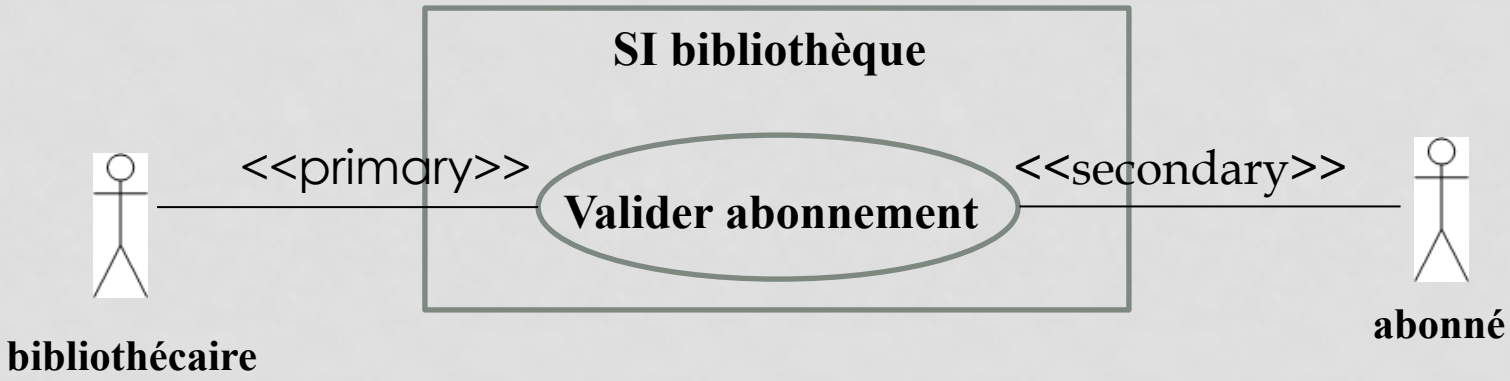
  Exemple: cas d'utilisation "*Retirer de l'argent*": le client introduit sa CB, saisit les codes d'accès et le montant souhaité, récupère l'argent et la carte.

  - et des scénarios alternatifs (qui se terminent de façon normale) et/ou d'erreur (qui se terminent en échec)

  Exemples: - le client saisit un faux code plus de 3 fois,
  - le client saisit un montant supérieur au solde

RELATION ACTEUR-USE CASE

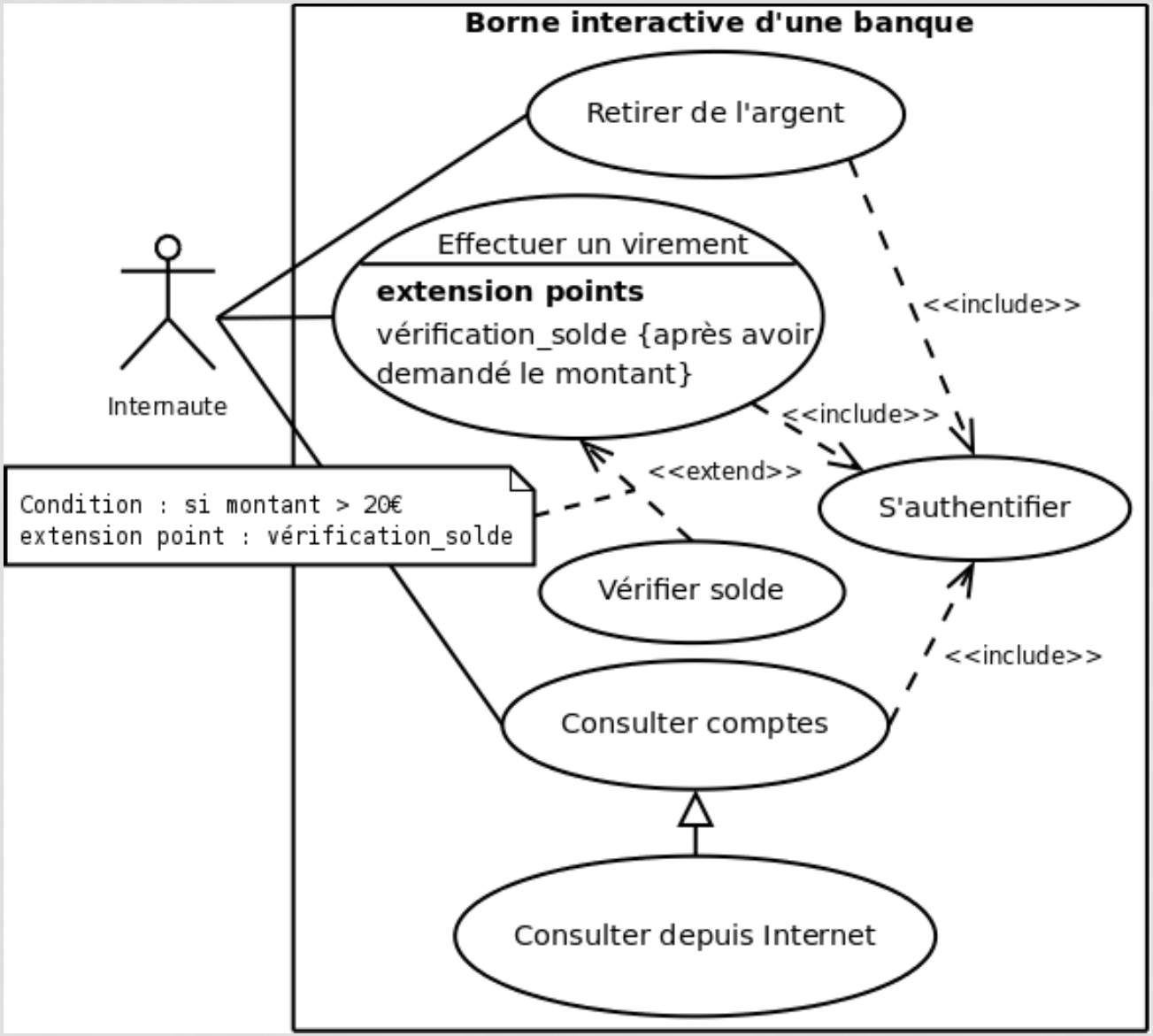
- **Acteur principal** (<<primary>> sur l'association): déclenche le use case et reçoit un résultat observable.
- **Acteur secondaire** (<<secondary>> sur l'association): sollicité pour des informations ou bien informé par le use case.



RELATIONS ENTRE CAS D'UTILISATION

Relation <<include>>

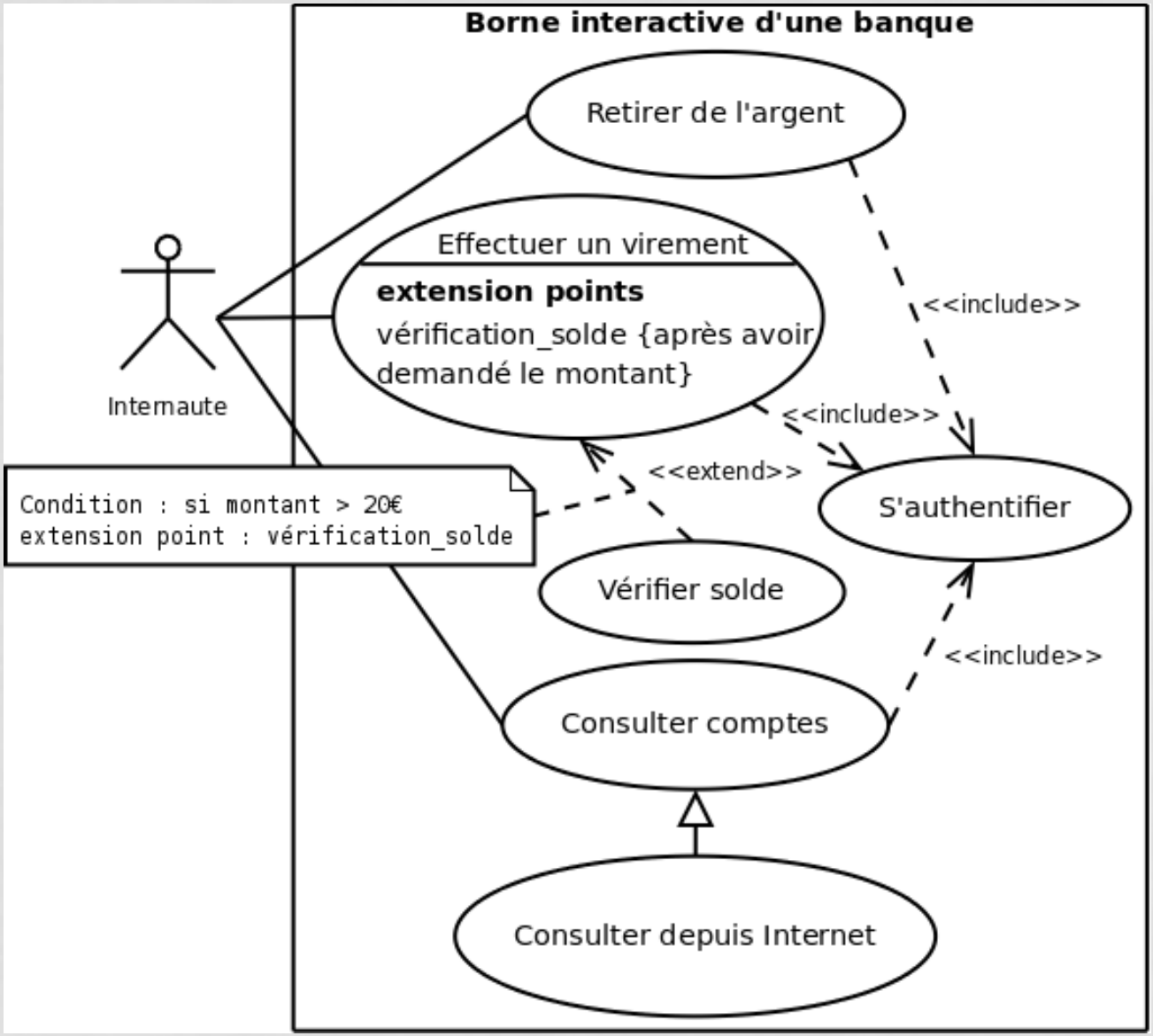
- Elle précise qu'un cas d'utilisation contient le comportement défini dans un autre cas d'utilisation
- Elle permet de mettre en commun des comportements communs à plusieurs cas d'utilisation
- Le cas inclus est ajouté **obligatoirement au cas de base**


Borne interactive d'une banque
Retirer de l'argent
Effectuer un virement
extension points
vérification_solde {après avoir demandé le montant}
<<include>>
Internaute
<<include>>
<<extend>>
Condition : si montant > 20€
extension point : vérification_solde
S'authentifier
Vérifier solde
<<include>>
Consulter comptes
Consulter depuis Internet

RELATIONS ENTRE CAS D'UTILISATION

Relation <<extend>>

- Elle précise qu'un cas d'utilisation peut dans certains cas augmenter le comportement d'un autre cas d'utilisation.
- Une condition devra valider cette augmentation définie dans un "point d'extension".

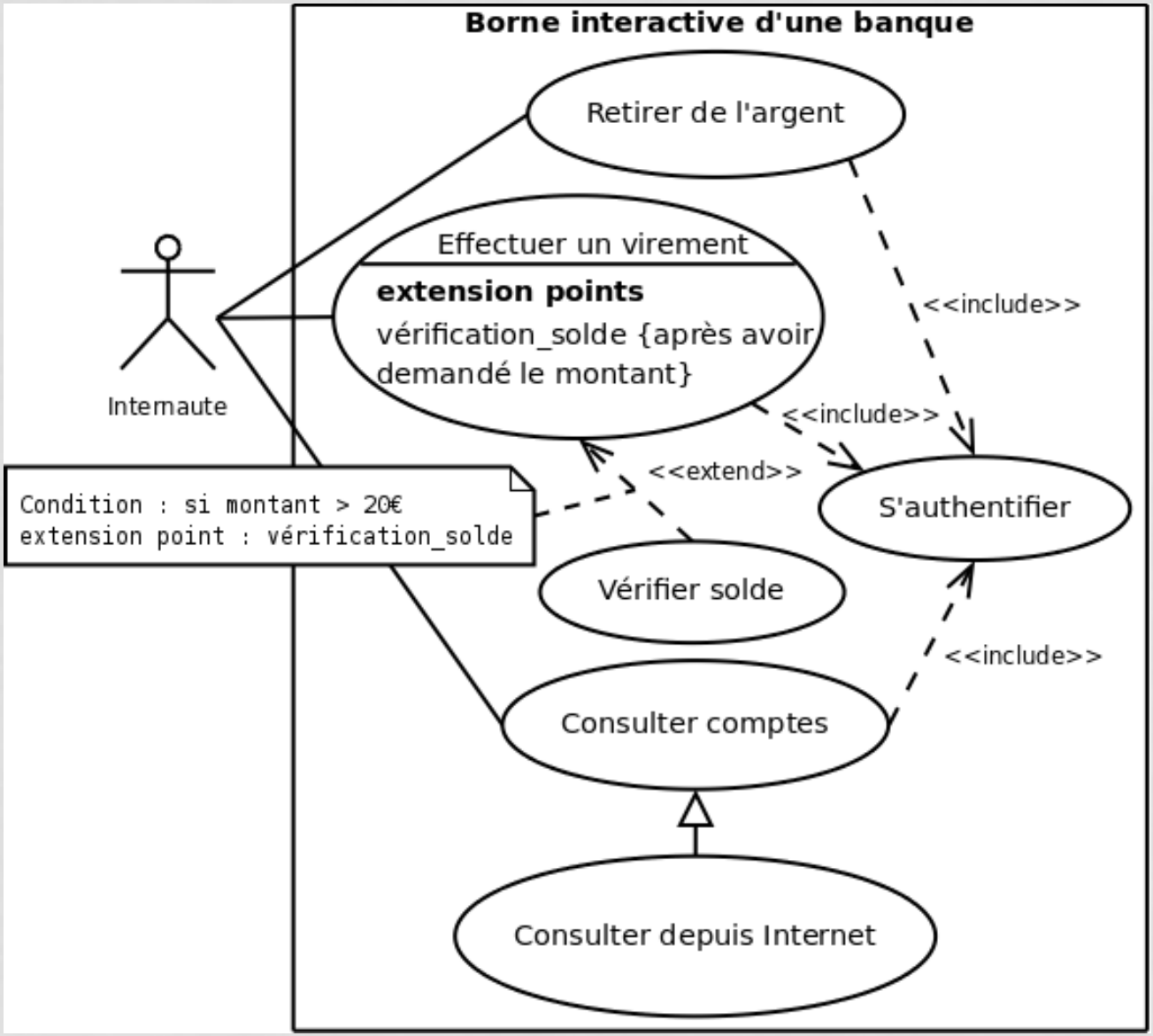

Borne interactive d'une banque
Retirer de l'argent
Effectuer un virement
extension points
vérification_solde {après avoir demandé le montant}
<<include>>
Internaute
<<include>>
<<extend>>
Condition : si montant > 20€
extension point : vérification_solde
S'authentifier
Vérifier solde
<<include>>
Consulter comptes
Consulter depuis Internet

RELATIONS ENTRE CAS D'UTILISATION

Relation de généralisation/spécialisation

- permet d'exprimer que les cas d'utilisation descendants héritent de la description de leur parent.

NB: Les descendants peuvent comprendre des interactions spécifiques supplémentaires, ou modifier les interactions dont ils ont hérité.


Borne interactive d'une banque
Retirer de l'argent
Effectuer un virement
extension points
vérification_solde {après avoir demandé le montant}
<<include>>
Internaute
<<include>>
<<extend>>
Condition : si montant > 20€
extension point : vérification_solde
S'authentifier
Vérifier solde
<<include>>
Consulter comptes
Consulter depuis Internet

DÉMARCHES POUR CONSTRUITE UN DCU

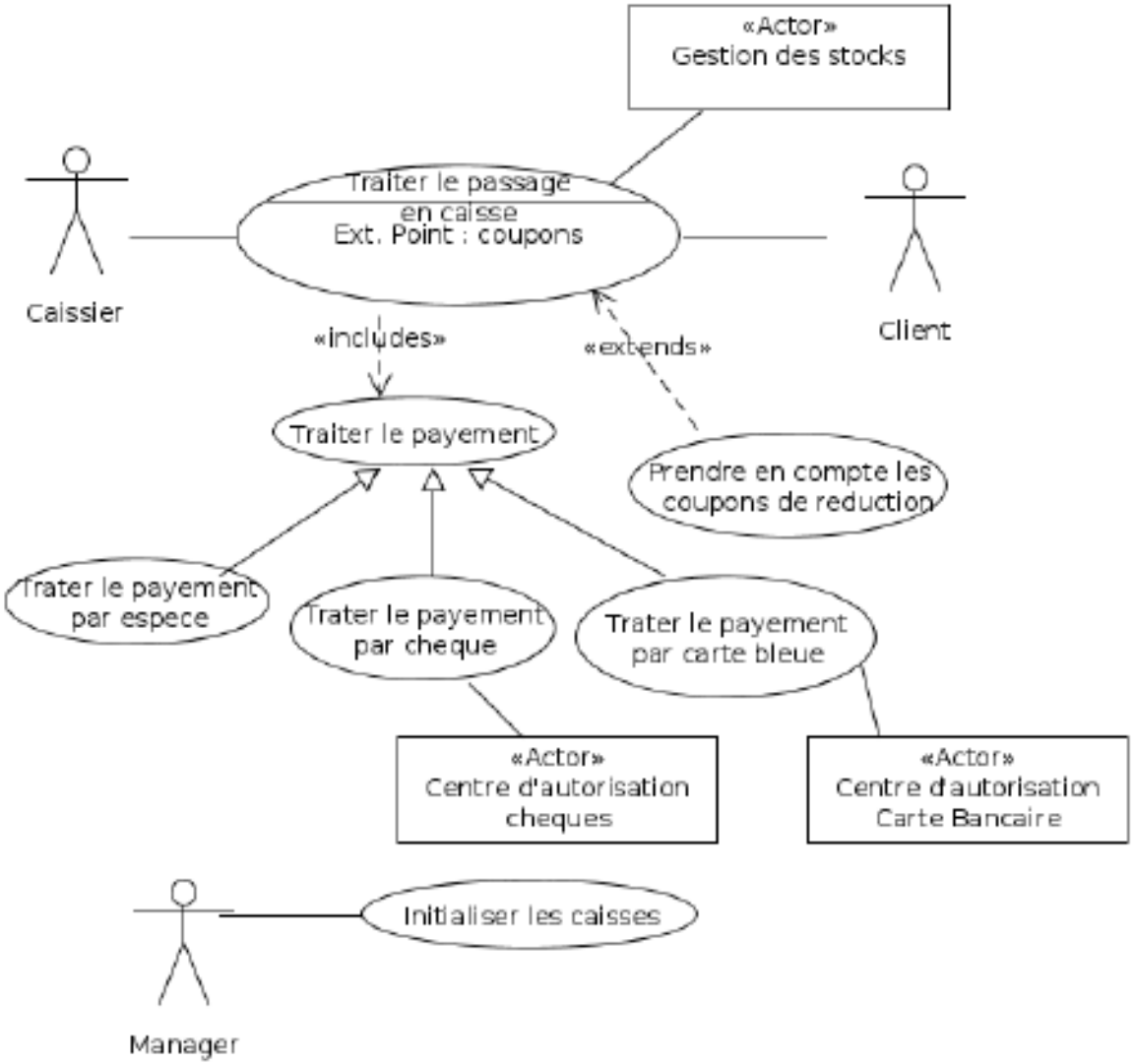
Etape 1. Identifier les acteurs

Etape 2. Identifier les cas d'utilisation: pour chaque acteur, déterminer comment il se sert du système. Se poser des questions:

- Quelles sont les tâches que l'acteur veut faire faire au système?
- Est-ce que l'acteur crée/modifie/supprime des informations dans le système?
- Est-ce que l'acteur a besoin d'informer les système de changements externes?

Etape 3. Ajouter les relations entre les cas d'utilisation

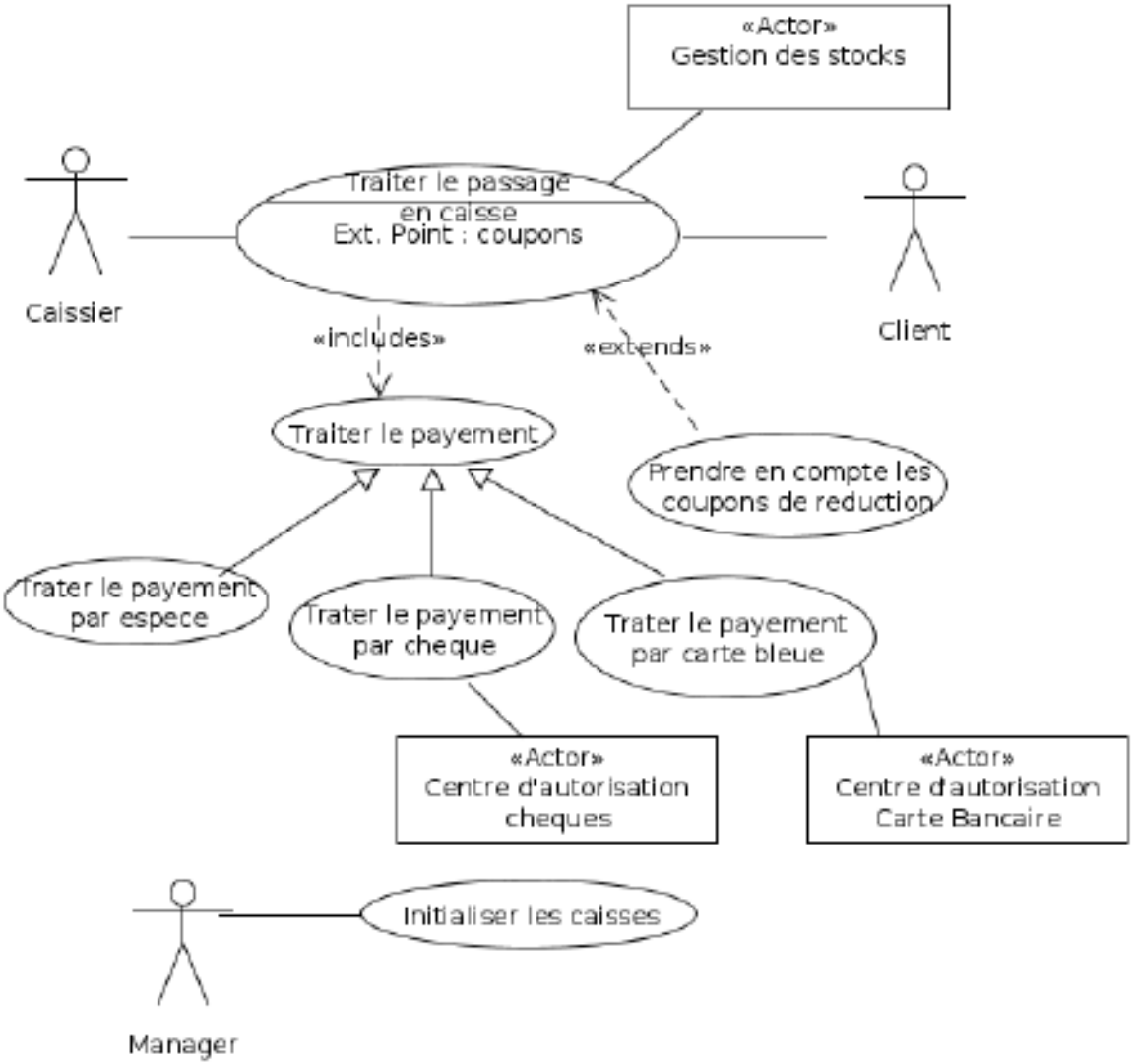
EXEMPLE: TERMINAL DE POINT DE VENTE



DESCRIPTION TEXTUELLE D'UN CAS D'UTILISATION

- **Titre**
- **Objectifs**: le contexte et les résultats attendus du cas d'utilisation
- **Acteurs**
- **Date de création**
- **Pré conditions**: les conditions requises avant l'exécution du cas
- **Post conditions**: les conditions a réunir après l'exécution du cas
- **Scénario nominal**: déroulement sans erreurs
- **Scénarios alternatifs**: variantes du scenario nominal
- **Scénarios d'exception**: décrivent les cas d'erreurs

EXEMPLE: TERMINAL DE POINT DE



EXEMPLE: TERMINAL DE POINT DE VENTE

Titre : Traiter le passage en caisse

Résumé : un client arrive à une caisse avec des articles qu'il souhaite acheter. Le caissier enregistre les achats et récupère le paiement. A la fin de l'opération, le client part avec les articles.

Acteurs : caissier (principal), client (secondaire).

Pré conditions : Le TPV est en service, un caissier y est connecté, le catalogue produit est disponible

Post conditions: La vente est enregistrée dans le TPV.

EXEMPLE: TERMINAL DE POINT DE VENTE

Scenario Nominal (1/3)

1. Ce cas d'utilisation commence quand un client arrive à la caisse avec des articles qu'il souhaite acheter.
2. Le caissier enregistre chaque article. S'il y a plus d'un exemplaire par article, le caissier indique également la quantité.
3. Le TPV valide le code de l'article et détermine le prix de l'article. Le TPV affiche la description et le prix de l'article en question.
4. Après avoir enregistré tous les articles, le caissier indique que la vente est terminée.

EXEMPLE: TERMINAL DE POINT DE VENTE

Scenario Nominal (2/3)

5. Le TPV calcule et affiche le montant total de la vente.
6. Le caissier annonce le montant total au client.
7. Le client choisit le type de paiement :
   a) En cas de paiement cash, exécuter l'UC "*Traiter le payement en cash*"
   b) En cas de paiement par carte bancaire, exécuter "*Traiter le payement par CB*"
   c) En cas de paiement par chèque, exécuter "*Traiter le payement par chèque* "

EXEMPLE: TERMINAL DE POINT DE VENTE

Scenario Nominal (3/3)

8. Le TPV enregistre la vente effectuée et imprime un ticket.
9. Le caissier donne le ticket de caisse au client.
10. Le client s'en va avec les articles qu'il a achetés.

EXEMPLE: TERMINAL DE POINT DE VENTE

Enchainements alternatifs ou d'erreur

A 1: numéro d'identification inconnu

Cette alternative démarre au point 3 du scénario nominal.

3. La caisse indique au caissier que le numéro d'identification de l'article est inconnu. L'article ne peut alors être pris en compte dans la vente en cours.

Le scénario nominal reprend au point 2.

A2: demande d'annulation d'un article

…

E1: annulation de la vente

L'enchainement E1 peut démarrer du point 2 au point 7 du scenario nominal

2→7 Le caissier annule l'ensemble de la vente et le cas d'utilisation se termine en échec

Les diagrammes de classe (DCL)

LE DCL EN BREF

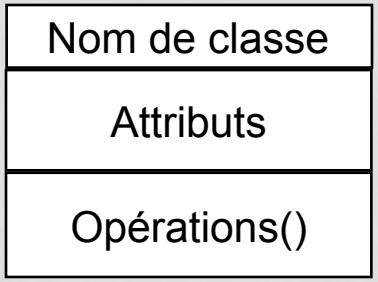
- DCL est une représentation de la structure statique d'un système, en terme de classes et des relations entre ces classes.

- Il permet de fournir une représentation abstraite des objets du système qui vont interagir pour réaliser les cas d'utilisation.

- Il existe de deux types de DCL:
  - d'analyse lié à aucun langage de programmation
  - de conception lié à un langage de programmation et enrichi d'éléments de conception tels que des patterns de conception (MVC)

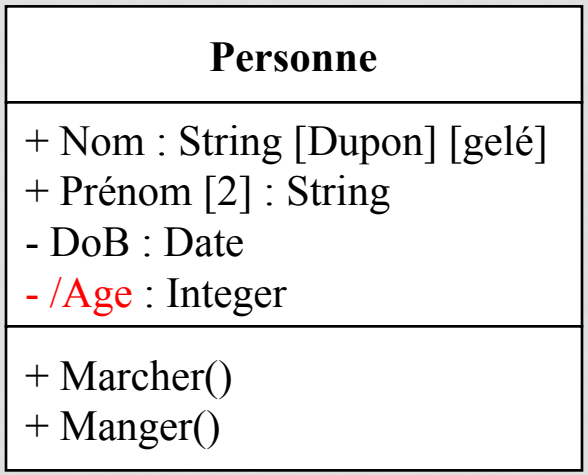
CLASSES

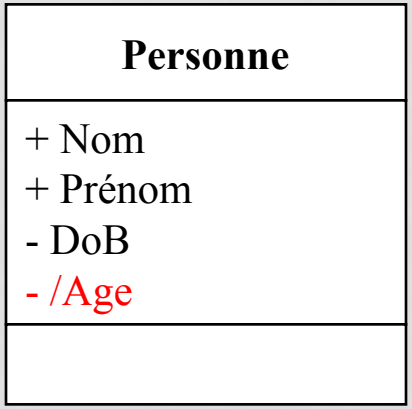
- Une classe est une description abstraite d'un ensemble d'objets qui partagent les mêmes *Attributs*, *Opérations* et *Relations*

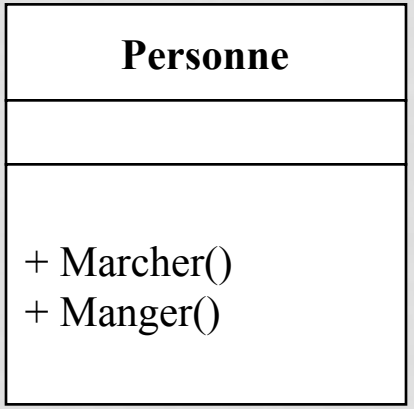


- Syntaxe de déclaration:
  - Attributs: Visibilité Nom_d'attribut [Multiplicité] : Type [Valeur initiale] [{Propriété}]
  - Operations: Visibilité Nom_d'opération ()

EXEMPLES DE CLASSES





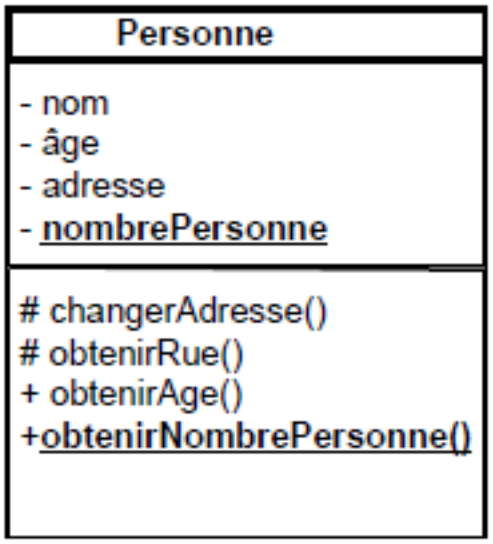


NB: Nom est un attribut non modifiable (ou constante),
Age est un attribut dérivé (ou calculé)

- **Protection des attributs et des opérations: l'encapsulation**
  - Privé (-): l'attribut (ou l'opération) est visible par sa classe seulement
  - Protégé (#): l'attribut (ou l'opération) est visible par sa classe et ses classes dérivées
  - Public (+): l'attribut (ou l'opération) est visible pour toutes les classes

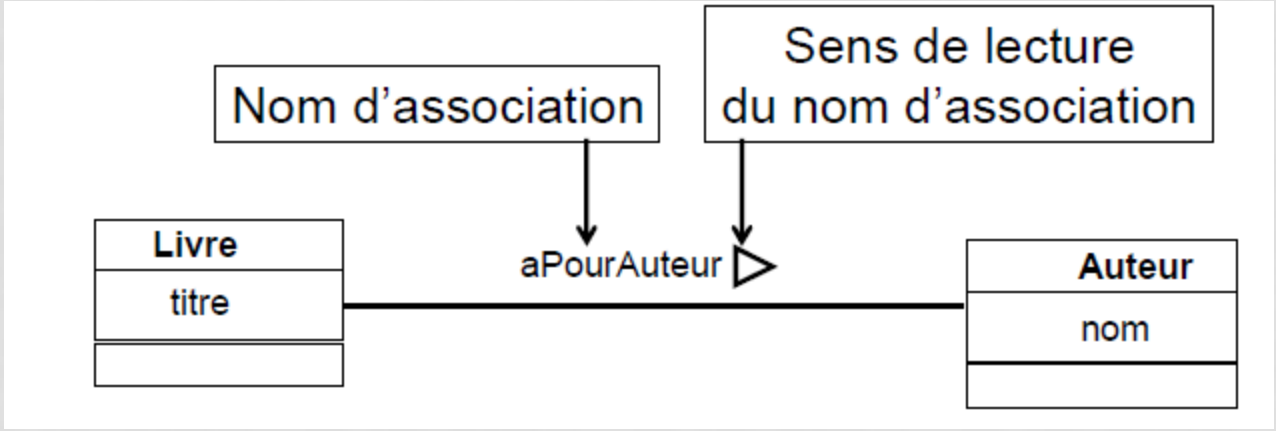
- **Attributs de niveau Classe**
  - Attributs dont la valeur est constante pour toutes les instances (les objets)
  - Représentation : nomAttributSouligné

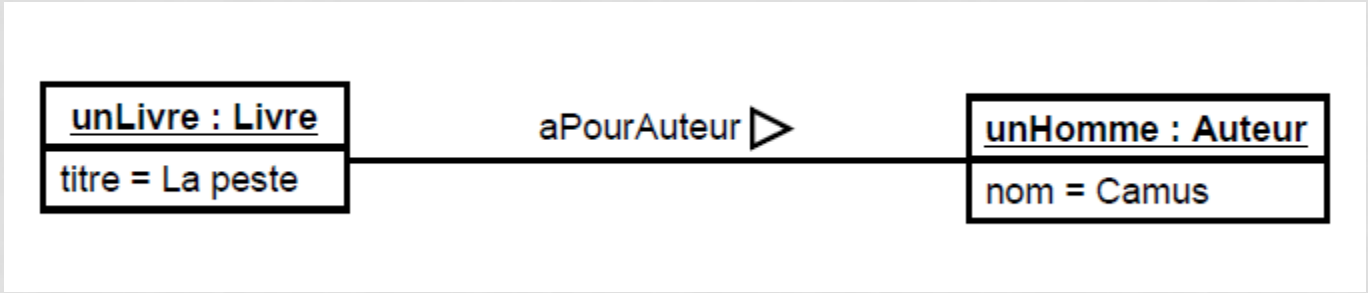
- **Opération de niveau classe**
  - Opération qui ne s 'applique pas aux objets de la cla
  - Opération qui s 'applique à la classe
  - Représentation : nomOpérationSoulignée



RELATIONS ENTRE CLASSES: LES ASSOCIATIONS

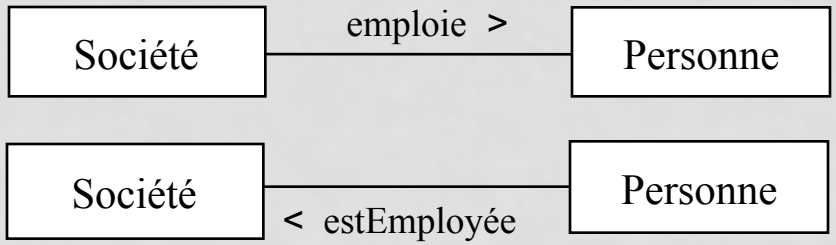
- Une Association précise que les objets d'une classe peuvent être reliés aux objets d'une autre classe.





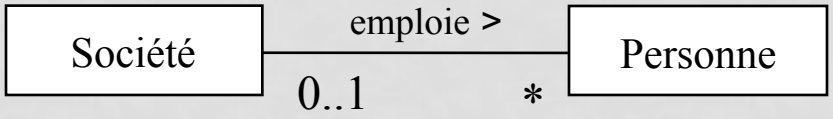
NOM D'ASSOCIATION, MULTIPLICITÉ

Le nom d'une association: le nommage des associations facilite la compréhension des modèles



Multiplicité

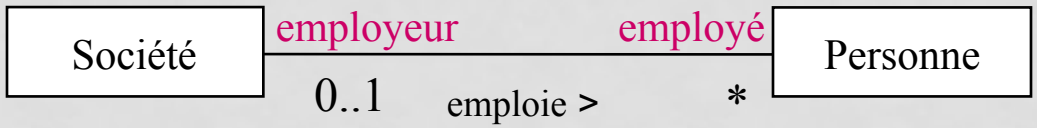
1	Un et un seul
0..1	Zéro ou un
M..N	De M à N (entiers)
M	M (entier)
* ou 0..*	de zéro à plusieurs
1..*	d'un à plusieurs

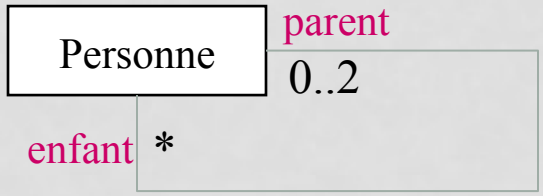


NOM DE RÔLE D'UNE ASSOCIATION

Rôle d'une association

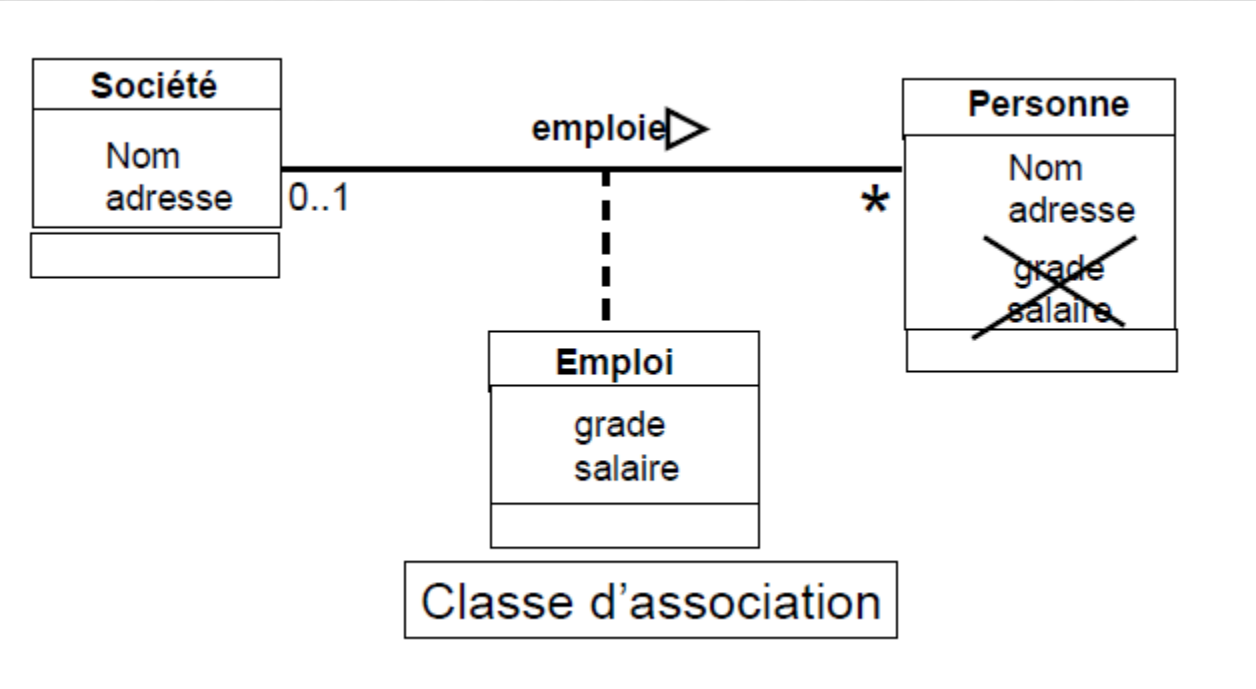
- Décrit comment une classe participe dans une association
- Devient obligatoire lorsque plusieurs associations existent entre 2 classes ou quand l'association est réflexive





LES CLASSES-ASSOCIATIONS

- Une association peut être représentée par une classe pour ajouter des attributs et des opérations

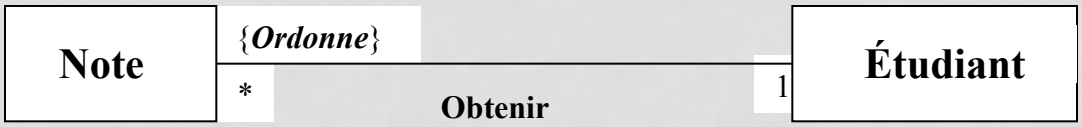


LES CONTRAINTES SUR LES ASSOCIATIONS

- Les contraintes permettent d'apporter plus de **précisions** à une association
- Peut être définie sur 1 ou plusieurs associations
- Notation **{expression}**

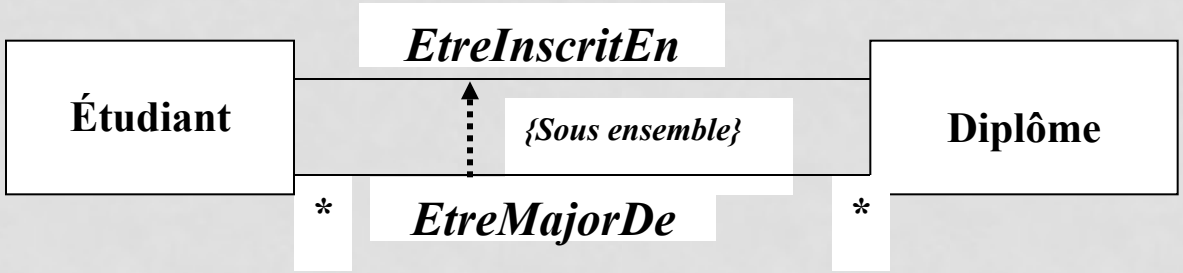
Exemple de contraintes:

- **{ordonnée}**: spécifie qu'il y a une relation d'ordre entre les objets d'une collection, mais ne spécifie pas comment ils sont ordonnés.

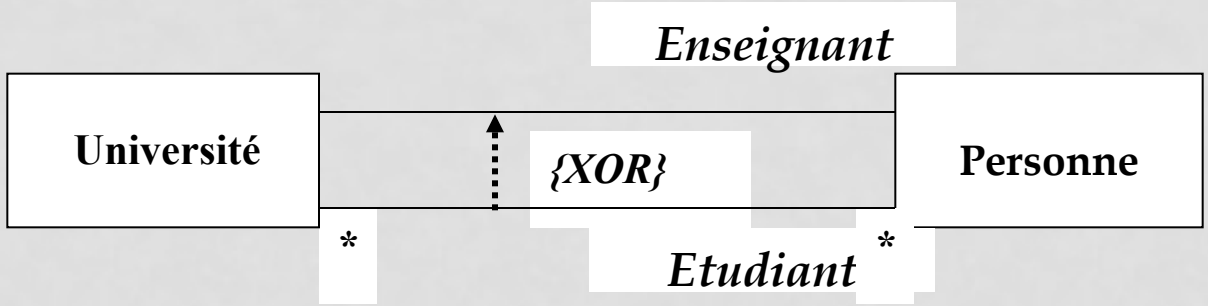


LES CONTRAINTES SUR LES ASSOCIATIONS

- **{sous-ensemble}:** une relation est incluse dans une autre.

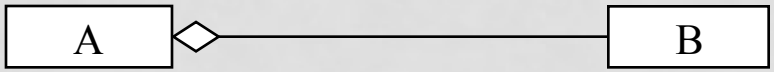


- **{ou-Excusif} ou {XOR}:** une seule association est valide pour un objet donné.

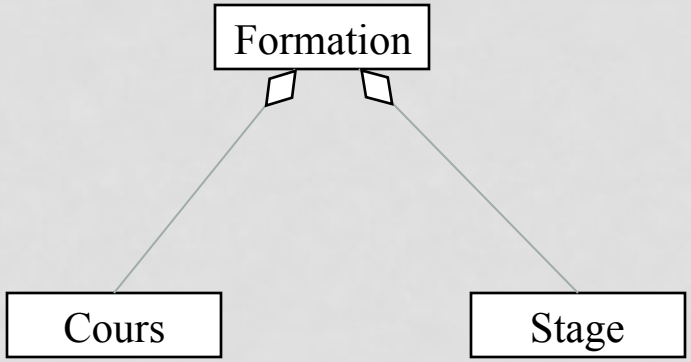


RELATIONS ENTRE CLASSES: AGRÉGATION

- L'agrégation est une association non symétrique, qui exprime un couplage fort et une relation de subordination.
- Elle représente une relation "*fait-partie-de*"
- Représentation:

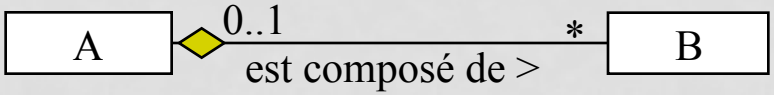


  - Les instances de B peuvent exister indépendamment de celles de A
  - Une instance de B peut éventuellement appartenir à plusieurs instances de A en même temps.

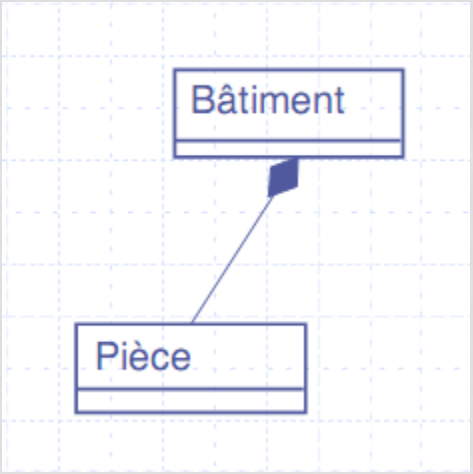


RELATIONS ENTRE CLASSES: COMPOSITION

- La composition est une agrégation forte qui porte la sémantique "est composé de"
- Représentation:



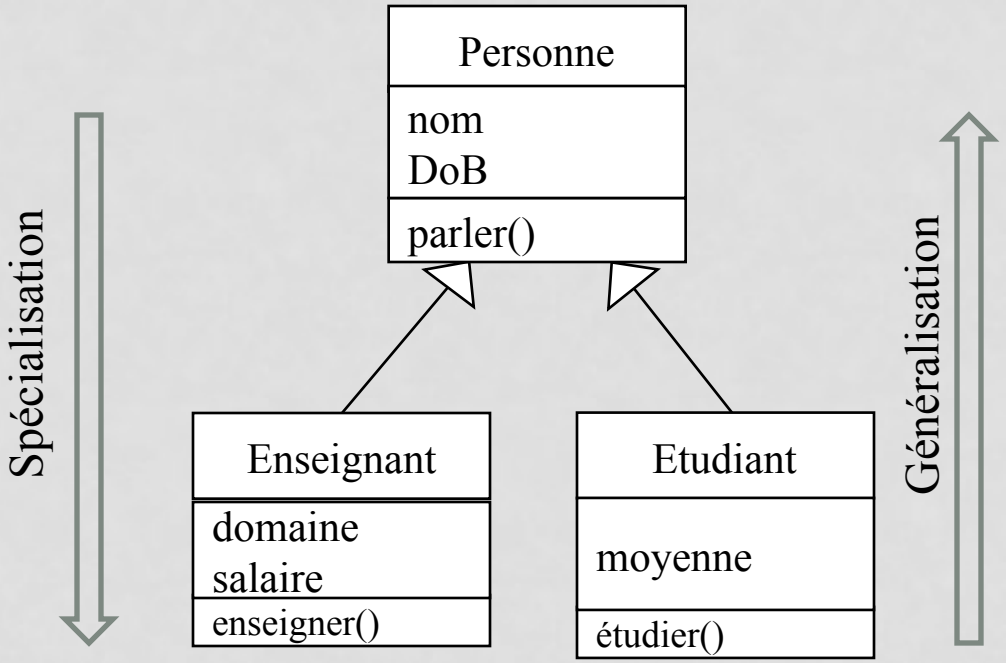
- A est composé de un ou plusieurs B
- Un B ne peut exister tout seul (dans le système)
- Un B n'appartient qu'à un seul A



RELATIONS ENTRE CLASSES: GÉNÉRALISATION/ SPÉCIALISATION

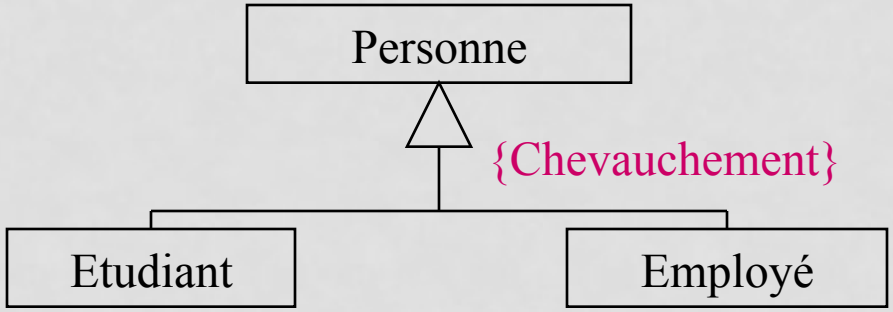
- La généralisation est une relation entre une classe et une ou plusieurs versions affinées de la classe (classes plus spécifiques).
- Permet une factorisation des attributs et des comportements communs à plusieurs classes

Enseignant *et* ***Etudiant*** sont des *spécialisations* (ou sous classes) de la classe ***Personne***.

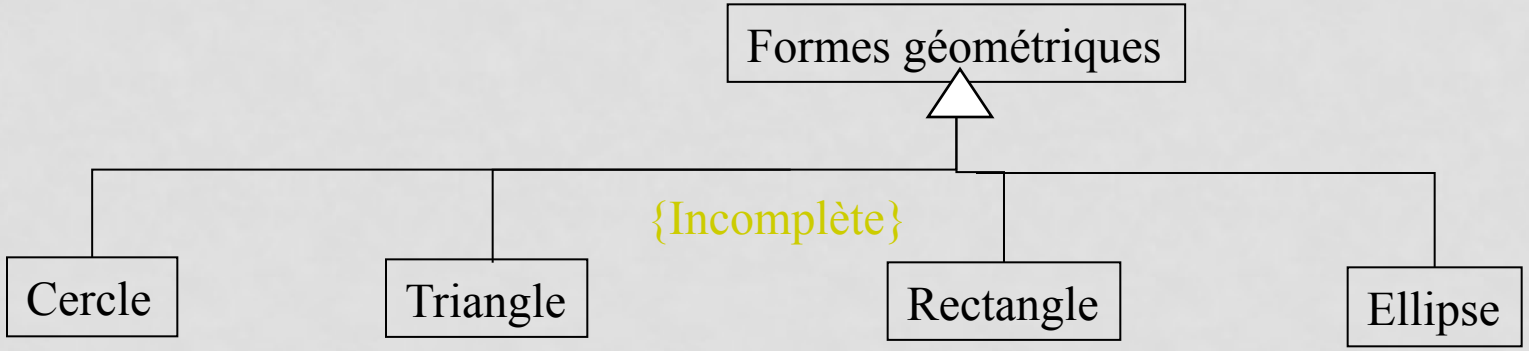


CONTRAINTES SUR LES GÉNÉRALISATIONS

- {chevauchement} ou {inclusif} : 2 sous-classes peuvent avoir, parmi leurs instances, des instances identiques,
- {Exclusif} : un objet est au plus une instance d'une seule sous classe

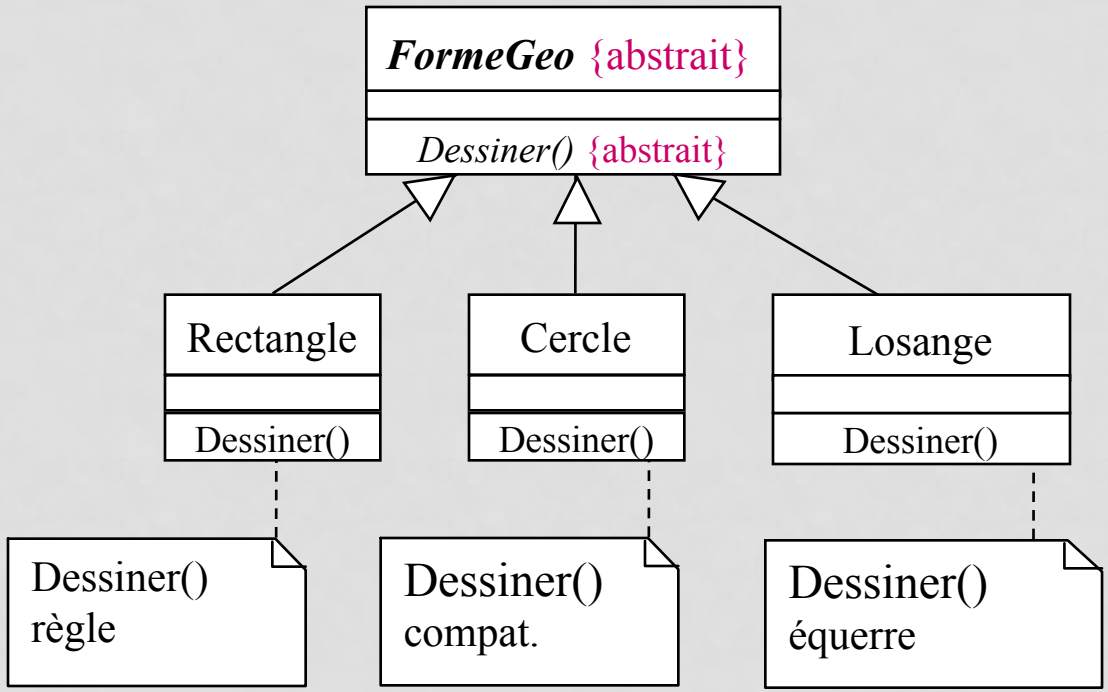


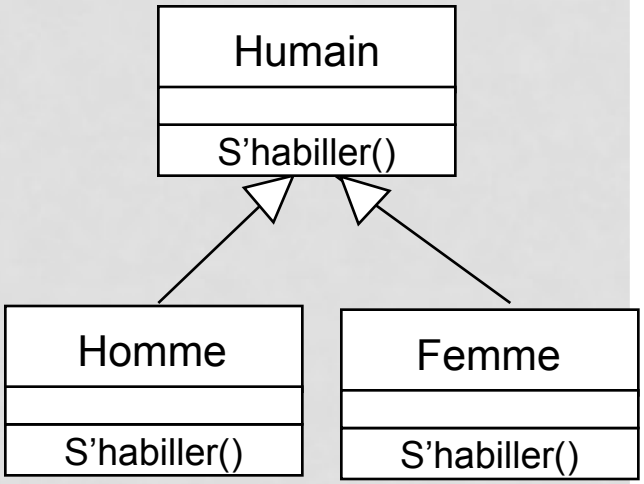
- *{Complète} : existence de toutes les sous-classes*
- *{Incomplète}* : généralisation extensible



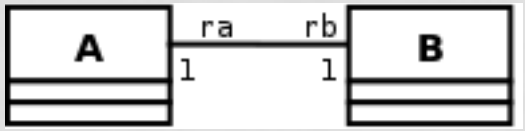
LES CLASSES ABSTRAITES

- Classe non instanciable,
- Classe de spécification générale: elle permet de déclarer le comportement que les sous-classes doivent implémenter.
- Représentation: nom de classe en *italique*, ou utilisation de <<abstrait>>





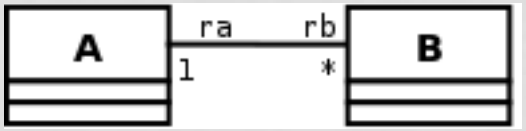
Implémentation Java-SQL




```
public class A {
  private B rb;
  public void addB( B b ) {
    if( b != null ){
      if ( b.getA() != null ) {   // si b est déjà connecté à un autre A
        b.getA().setB(null);    // cet autre A doit se déconnecter
      }
      this.setB( b );
      b.setA( this );
    }
  }
  public B getB() { return( rb ); }
  public void setB( B b ) { this.rb=b; }
}
```

```
public class B {
  private A ra;
  public void addA( A a ) {
    if( a != null ) {
      if (a.getB() != null) {   // si a est déjà connecté à un autre B
        a.getB().setA( null );  // cet autre B doit se déconnecter
      }
      this.setA( a );
      a.setB( this );
    }
  }
  public void setA(A a){ this.ra=a; }
  public A getA(){ return(ra); }
}
```

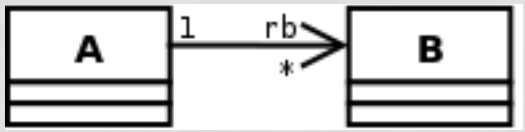
Implémentation Java-SQL




```
public class A {
  private ArrayList <B> rb;
  public A() { rb = new  ArrayList<B>(); }
  public ArrayList <B> getArray()
{return(rb);}
  public void remove(B b)
{rb.remove(b);}
  public void addB(B b){
    if( !rb.contains(b) ){
      if (b.getA()!=null)
b.getA().remove(b);
      b.setA(this);
      rb.add(b);
    }
  }
}
```

```
public class B {
  private  A ra;
  public B() {}
  public A getA() { return (ra); }
  public void setA(A a){ this.ra=a; }
  public void addA(A a){
    if( a != null ) {
      if( !a.getArray().contains(this)) {
        if (ra != null) ra.remove(this);
        this.setA(a);
        ra.getArray().add(this);
      }
    }
  }
}
```

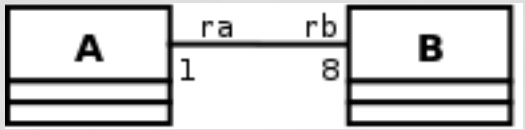
Implémentation Java-SQL




```
public class A {
  private ArrayList <B> rb;
  public A() { rb = new  ArrayList<B>(); }
  public void addB(B b){
    if( !rb.contains( b ) ) {
      rb.add(b);
    }
  }
}
```

```
public class B {
 ... // B ne connaît pas l'existence de A
}
```

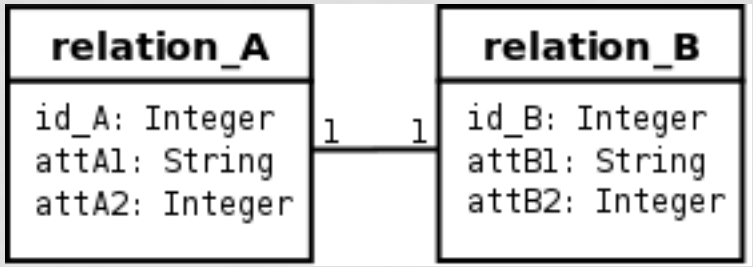
Implémentation Java-SQL



Utiliser un tableau au lieu d'un vecteur

Agrégation s'implémente comme une association

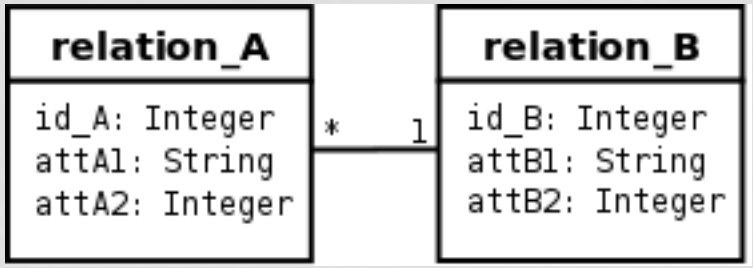
Implémentation Java-SQL




```
create table relation_A (
    id_A integer primary key,
    attA1 text,
    attA2 integer);

create table relation_B (
    id_B integer primary key,
    num_A integer references relation_A,
    attB1 text,
    attB2 integer);
```

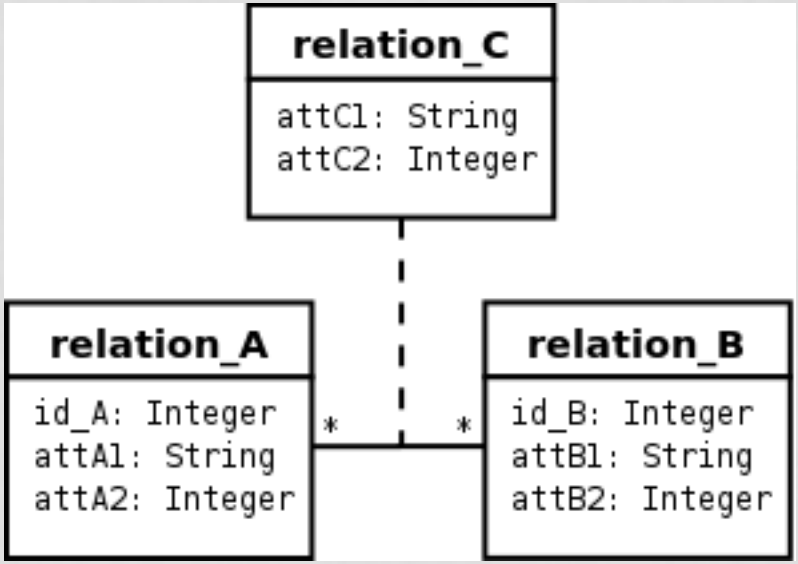
Implémentation Java-SQL




```
create table relation_A (
    id_A integer primary key,
    num_B integer references relation_B,
    attA1 text,
    attA2 integer);

create table relation_B (
    id_B integer primary key,
    attB1 text,
    attB2 integer);
```

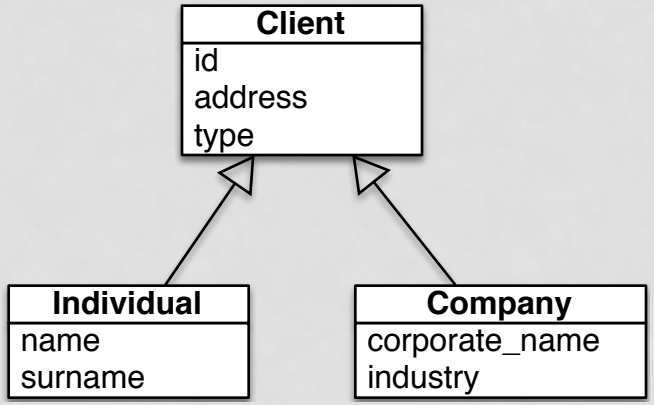
Implémentation Java-SQL




```
create table relation_A (
    id_A integer primary key,
    attA1 text,
    attA2 integer);
```

```
create table relation_B (
    id_B integer primary key,
    attB1 text,
    attB2 integer);
```

```
create table relation_C (
    num_A integer references relation_A, num_B integer
references relation_B,  attC1 text, attC2 integer,
    primary key (num_A, num_B));
```




```
create table client_t (
    id integer primary key,
    address text,
    type text);

create table individual_t (
    id_B references client_t,
    name text,
    surname integer,
    primary key (id_B));
```

```
create table company_t (
    id_A references client_t,
    corporate_name text,
    industry text,
    primary key (id_A));
```

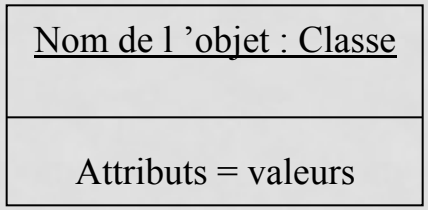
```
CREATE VIEW individual AS
select
  client_t.id,
  client_t.address,
  individual_t.name,
  individual_t.surname
  from client_t, individual_t
  where client_t.id =
individual_t.client_id and
client_t.type='i';
```

Le diagramme d’objets (DOB)

Diagramme d'Objets

Structure statique d'un système, en termes d'**objets** et de **liens** entre ces objets.

Un objet est une *instance* de classe et un lien est une instance de relation.



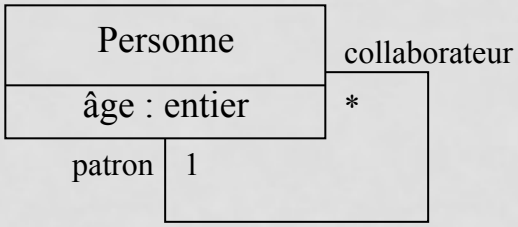
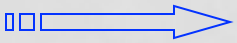


Diagramme de classes



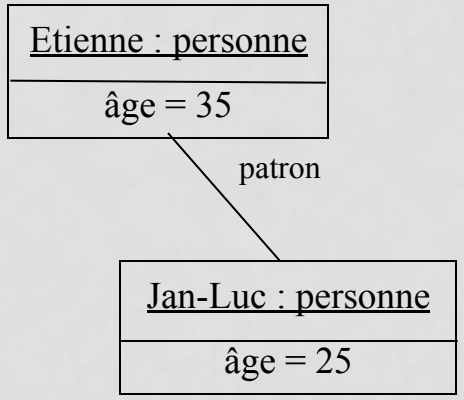
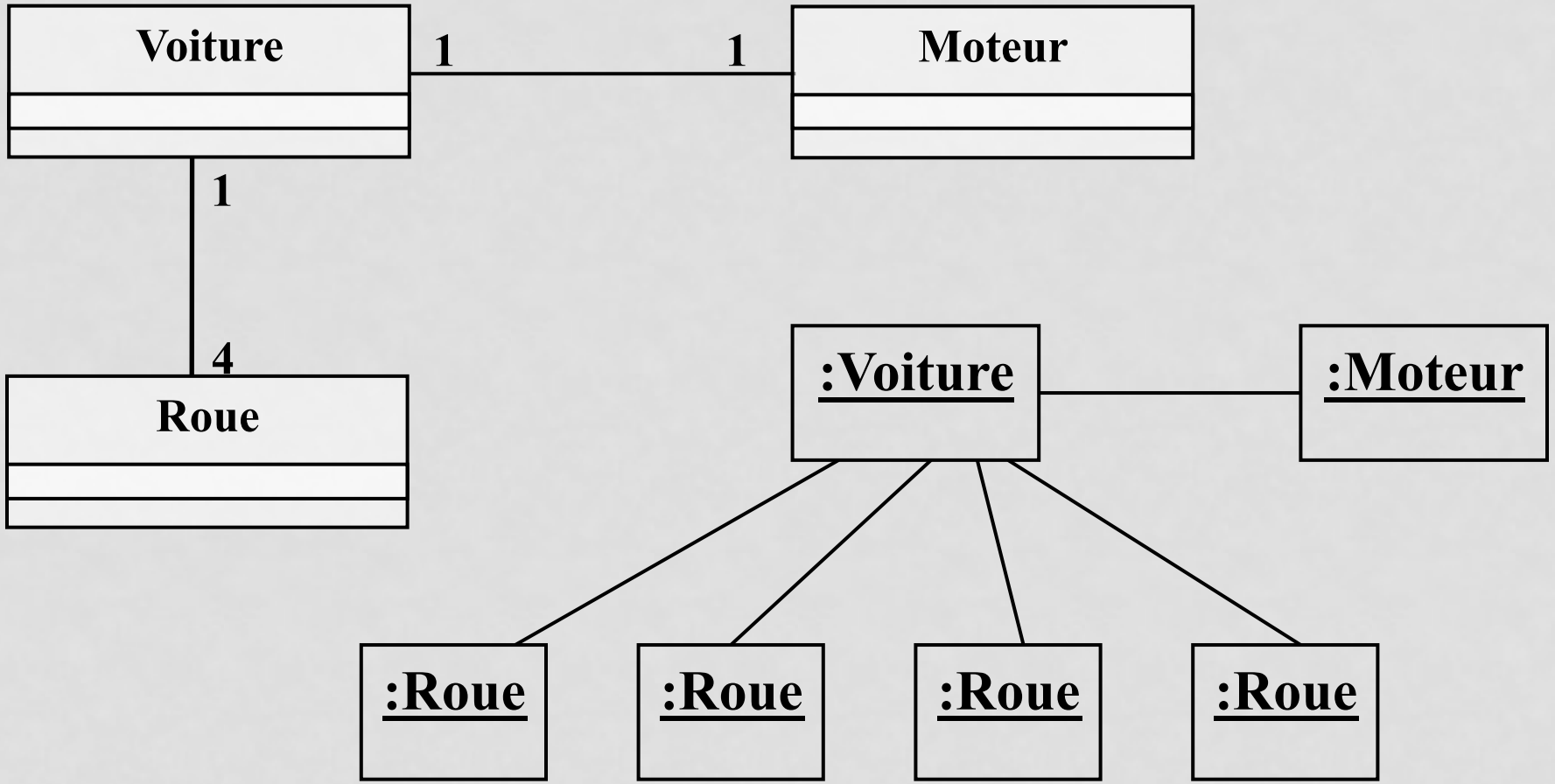
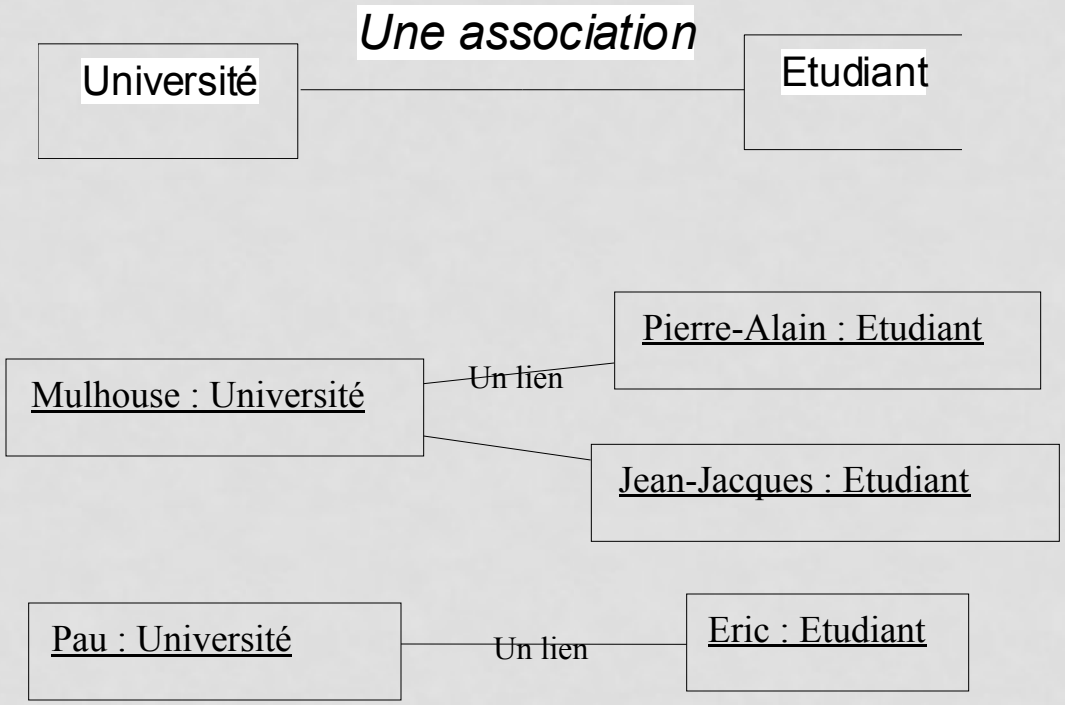


Diagramme d 'objets

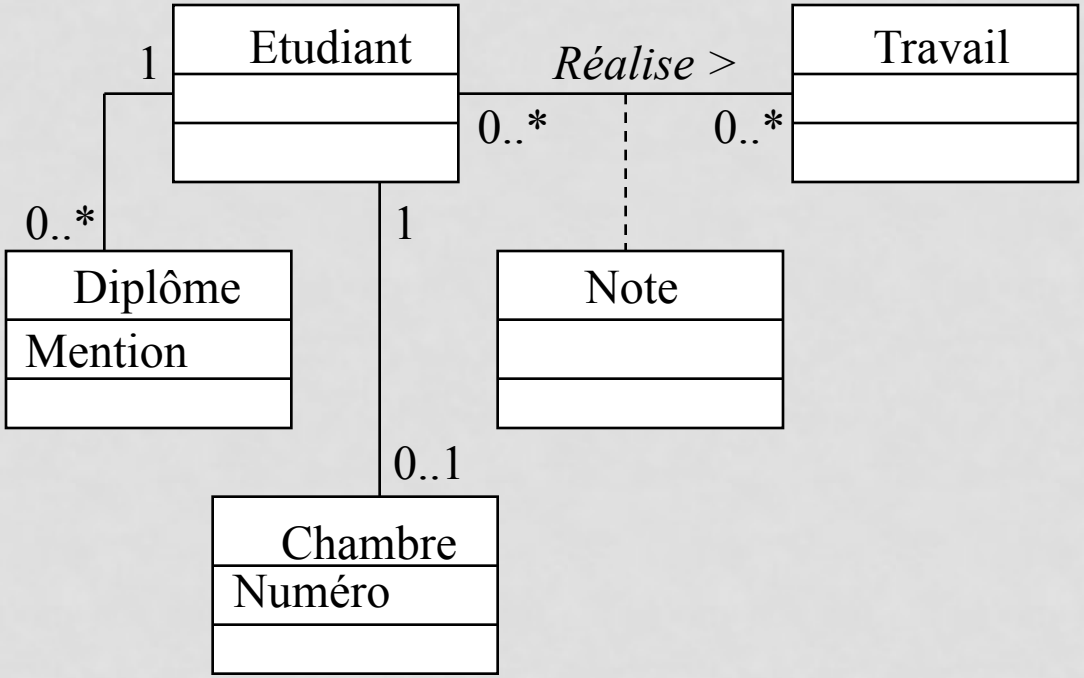
Exemple


Voiture
1
1
Moteur
1
4
Roue
:Voiture
:Moteur
:Roue
:Roue
:Roue
:Roue

AUTRE EXEMPLE

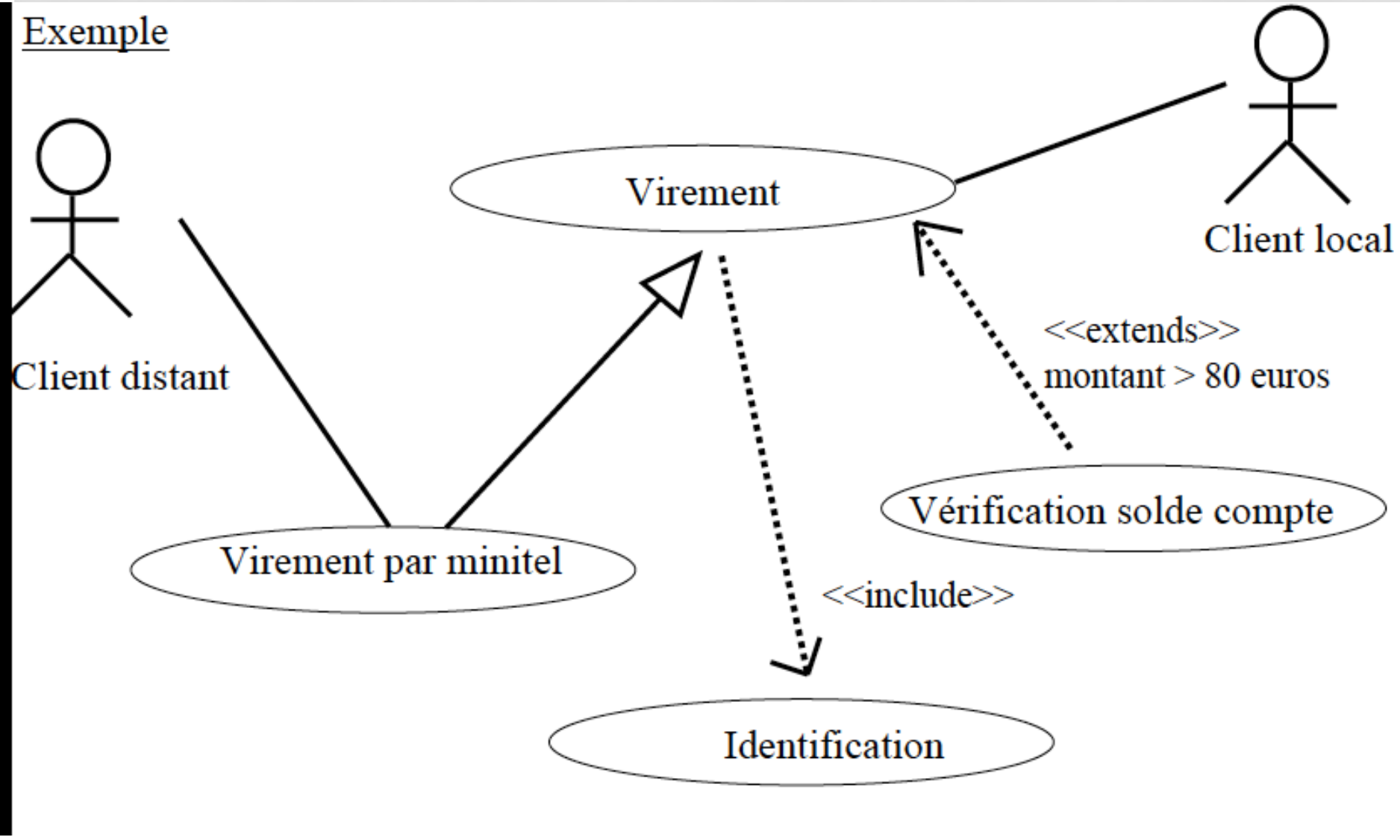


EXERCICE:
FAIRE CORRESPONDRE LE DCL SUIVANT EN UN DOB.



Passage du fonctionnel à l'objet

Transition vers les objets: Exemple



Transition vers les objets: Exemple

Description des enchaînements

- **Scénario nominal :**

1. Le guichetier saisit le numéro du compte du client
2. L'application valide le compte auprès du système central
3. Le guichetier vérifie l'identité du client (d'après la carte d'identité)
4. L'application demande le montant du retrait au guichetier
5. Le guichetier sélectionne un retrait d'espèces de 20 euros
6. Le système guichet interroge le système central pour s'assurer que le compte est suffisamment approvisionné
7. Le système central effectue le débit du compte
8. Le système notifie au guichetier qu'il peut délivrer le montant demandé

- **Enchaînements alternatifs** :

A1. Numéro de compte non valide : l'enchaînement A1 démarre au point 1 du scénario nominal

2. Le système central indique que le compte n'est pas valide

Le scénario nominal reprend au point 1.

A2. Compte non suffisamment approvisionné : l'enchaînement A2 démarre au point 6 du scénario nominal

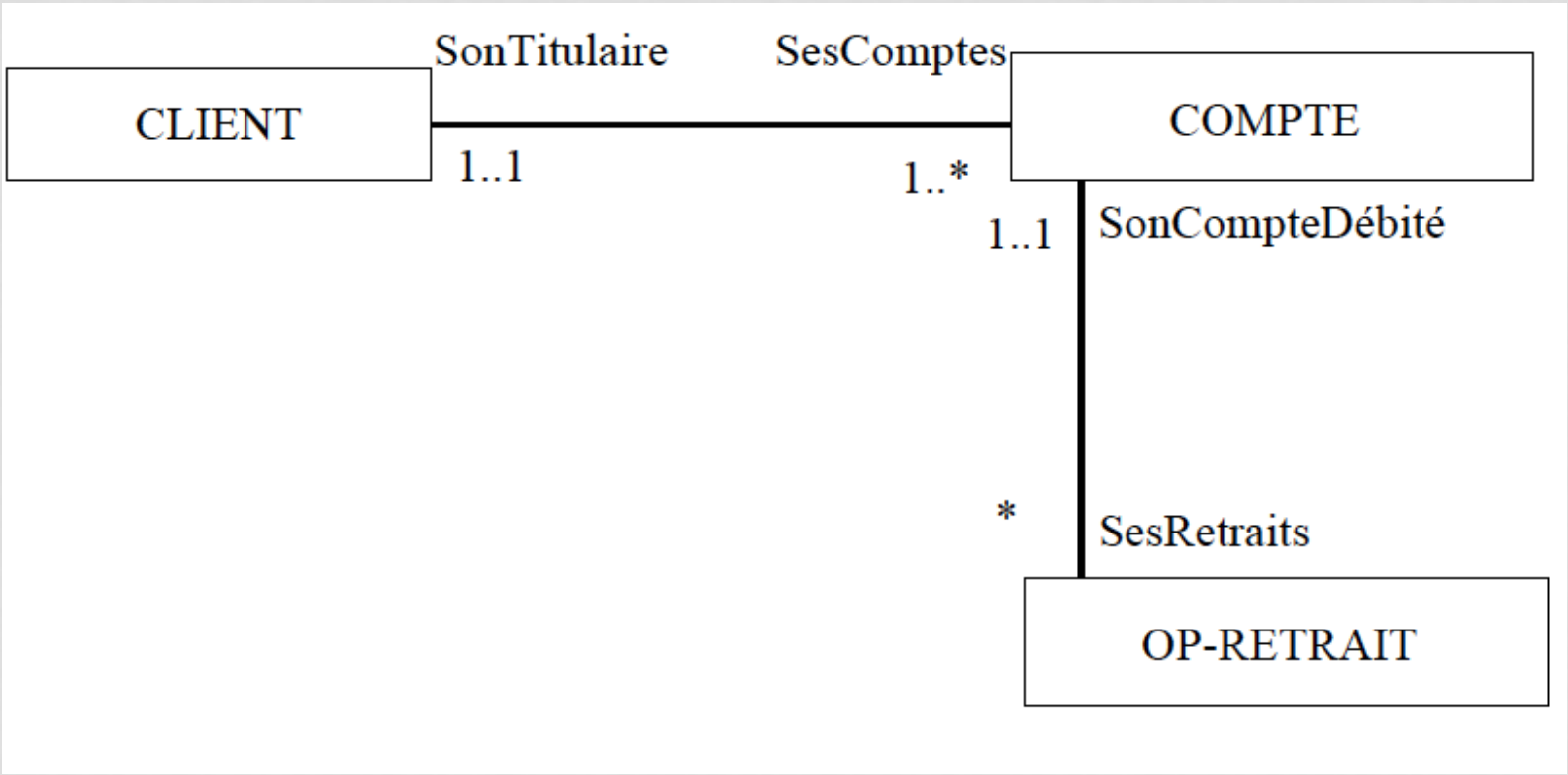
7. Le système central indique que le montant demandé est supérieur au solde du compte.

Le scénario nominal reprend au point 4.

- **Postconditions** : montant demandé délivré & fin du retrait.

Transition vers les objets: Exemple

Premier jet de DCL



Transition vers les objets: Exemple

Analyse plus poussée

- **Scénario nominal :**

1. Le guichetier saisit le *numéro* du compte du client
2. L'application valide le compte auprès du système central qui retourne *le nom, prénom* du client associé au compte
3. Le guichetier vérifie l'identité du client d'après la carte d'identité
4. L'application demande le *montant* du retrait
5. Le guichetier sélectionne un retrait d'espèces de 20 euros
6. Le système guichet interroge le système central *pour s'assurer que le compte est suffisamment approvisionné*
7. Le système central effectue le débit du compte : *le solde du compte est mis à jour*
8. Le système notifie au guichetier qu'il peut délivrer le montant demandé : *une instance de op-retrait est créée*

- **Enchaînements alternatifs** :

A1. Numéro de compte non valide : l'enchaînement A1 démarre au point 1 du scénario nominal

2. Le système central indique que le compte n'est pas valide

Le scénario nominal reprend au point 1.

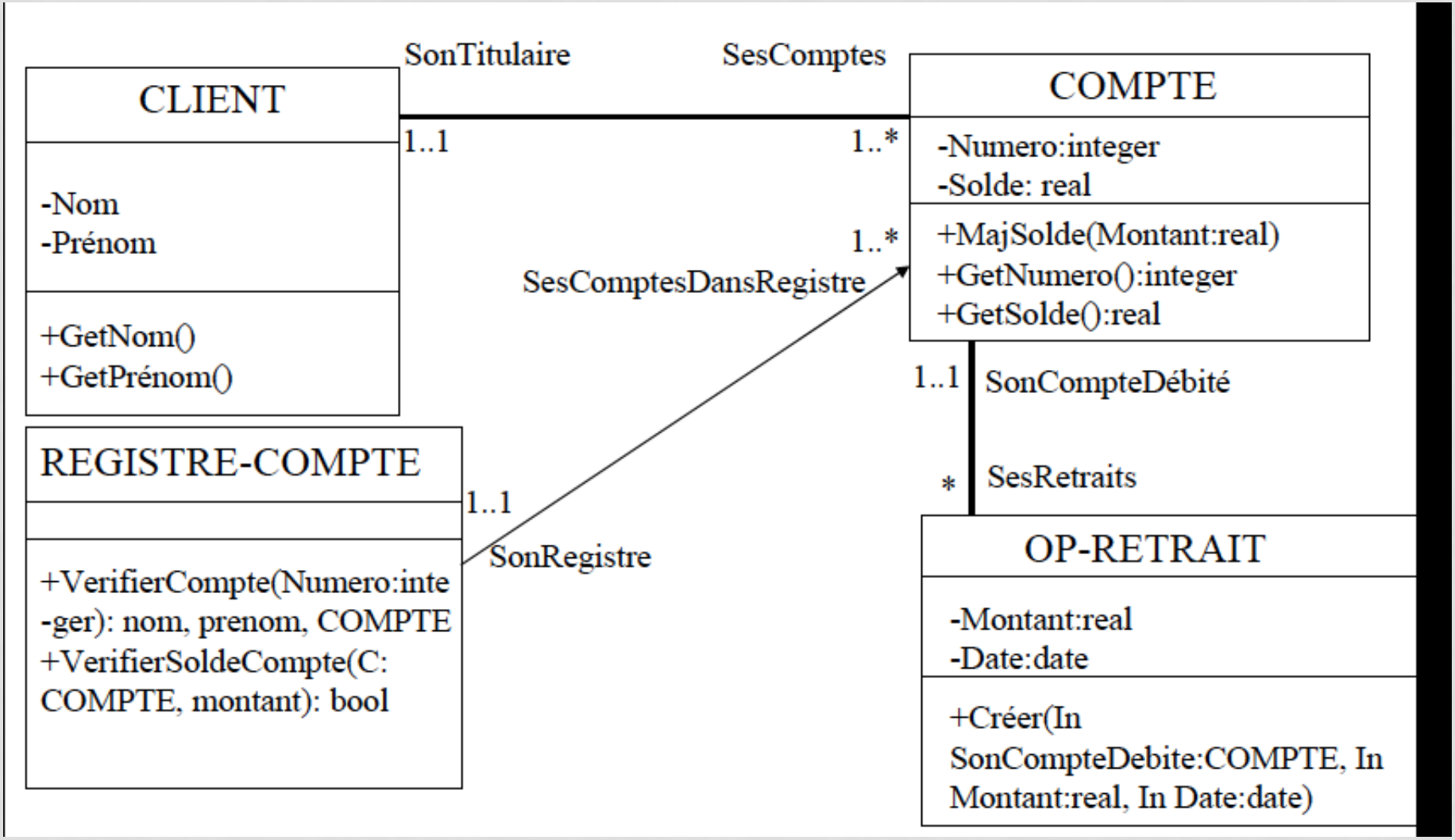
A2. Compte non suffisamment approvisionné : l'enchaînement A2 démarre au point 6 du scénario nominal

7. Le système central indique que le montant demandé est supérieur au solde du compte.

Le scénario nominal reprend au point 4.

- **Postconditions** : montant demandé délivré & fin du retrait.

Transition vers les objets: Exemple



Description graphique des Use Cases

POURQUOI?

- Pour documenter les cas d'utilisation, la description textuelle est indispensable car elle permet de communiquer facilement avec les utilisateurs finaux et de s'entendre sur la terminologie métier employée.

- En revanche, il est difficile de montrer comment les enchaînements se succèdent, ou à quel moment les acteurs secondaires sont sollicités.

- Il est alors **recommandé** de la compléter par un ou plusieurs diagrammes UML.

DIAGRAMMES DÉCRIVANT DES SCÉNARIOS DE USE CASES

- En Analyse, pour décrire dynamiquement un scénario, on utilise deux diagramme UML:
  - Un diagramme d'activités, qui permet de décrire les activités d'un CU, ou d'un ensemble de CU en restant à un **haut niveau d'abstraction** (besoins en IHM). Il peut aussi être utilisé pour décrire des **processus métier** (besoins fonctionnels).
  - Un diagramme de séquence système, qui montre les échanges avec les acteurs externes (besoins en IHM (un CU)).

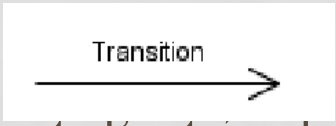
- En Analyse détaillée pour un scénario, on utilise le diagramme de séquence de conception.

Diagramme d’activité (DIT)

ELÉMENTS DE NOTATION (1)

- **Etat d'activité**: Marque une action faite par une entité (e.g., Acteur, Système)

- **Transition**: Quand un état d'activité est accompli, le traitement passe à un autre état d'activité. Les transitions sont utilisées pour marquer ce passage.



- **Etat initial**: Marque le point d'entrée de la première activité. Il ne peut exister qu'un seul dans le diagramme

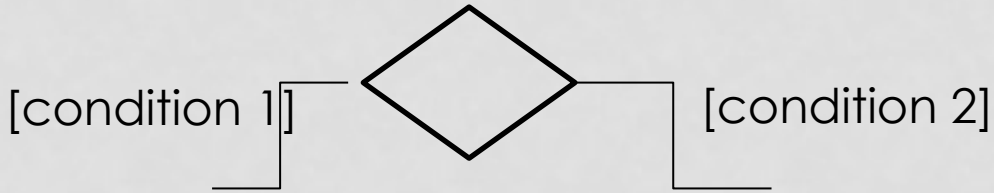
- **Etat final**: Marque la fin du déroulement des opérations. Il peut exister un ou plusieurs états finaux.

Eléments de notation (2)

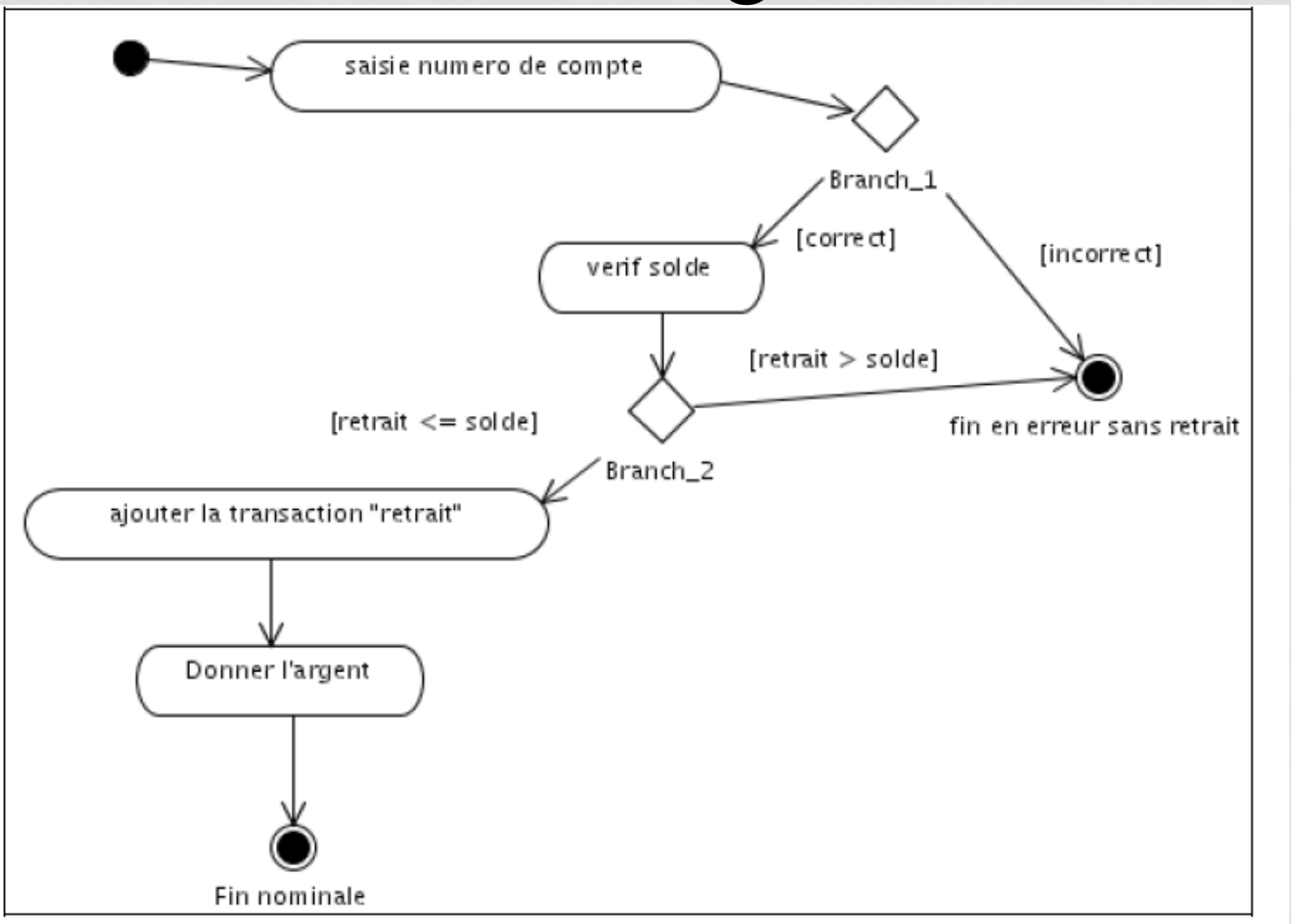
- **Barre de synchronisation**: Souvent certaines activités peuvent être synchronisées (**join**) ou bien faites en parallèle (**fork**).

- **Couloir**: Les activités peuvent être placées dans des couloirs, représentant des entités (e.g., Acteur, Système)

- Décision : Branchement conditionnel

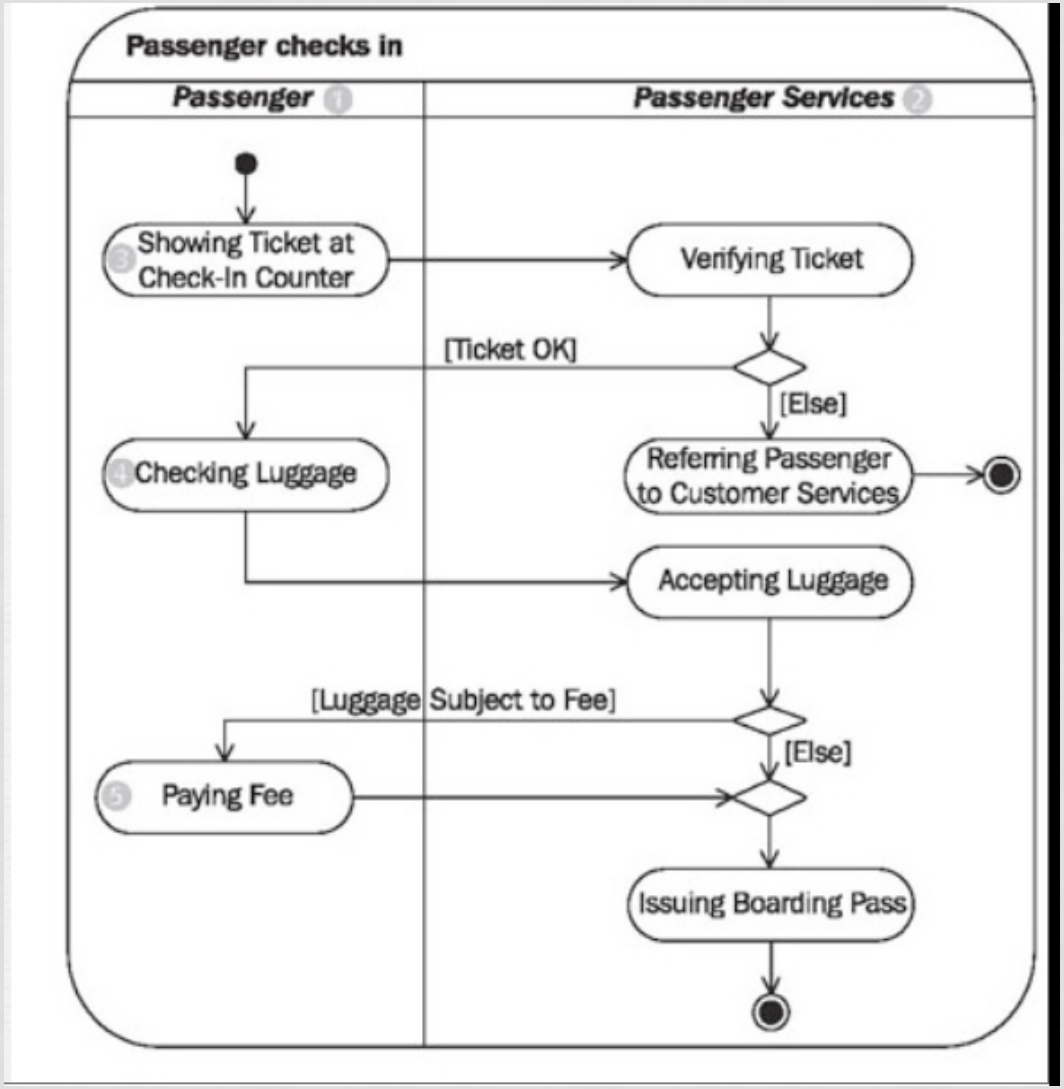


DIT « Retrait d'argent » GAB



DIT Enregistrer Passager

- **Couloir ou Swinlane**

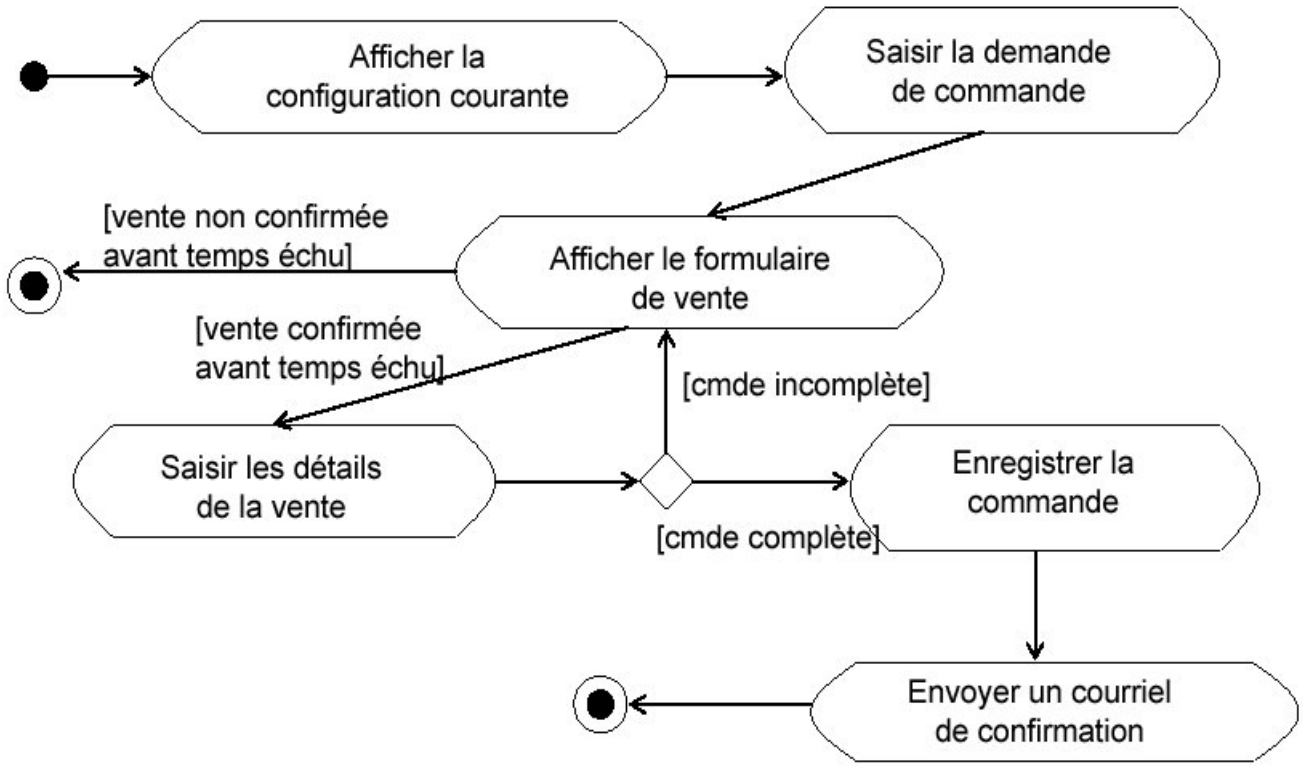


Construction un diagramme d'activité

1. **Identifiez la portée (« scope ») du diagramme d'activité**
   Commencez en identifiant ce que vous allez modéliser. Un seul use case? Une partie d'un use case ? Un « workflow » qui inclut plusieurs use cases ? Une méthode de classe ?
2. **Ajouter l'état de *départ* et de *terminaison***
3. **Ajouter les actions**
   Si vous modélisez un « workflow », introduisez une activité pour chaque processus principal, souvent un use case. Enfin, si vous modélisez une méthode, il est souvent nécessaire d'avoir une action pour chaque grande étape de la méthode.
4. **Ajouter des transitions (séquentielles), des transitions alternatives (conditionnelles), des synchronisations entre des actions, des itérations.**
5. **Identifier des partitions et répartir des actions identifiées dans ces partitions.**

Système de vente en ligne

Cas d'utilisation: Commander ordinateur (à configurer)



DIT système « Retrait d'argent »

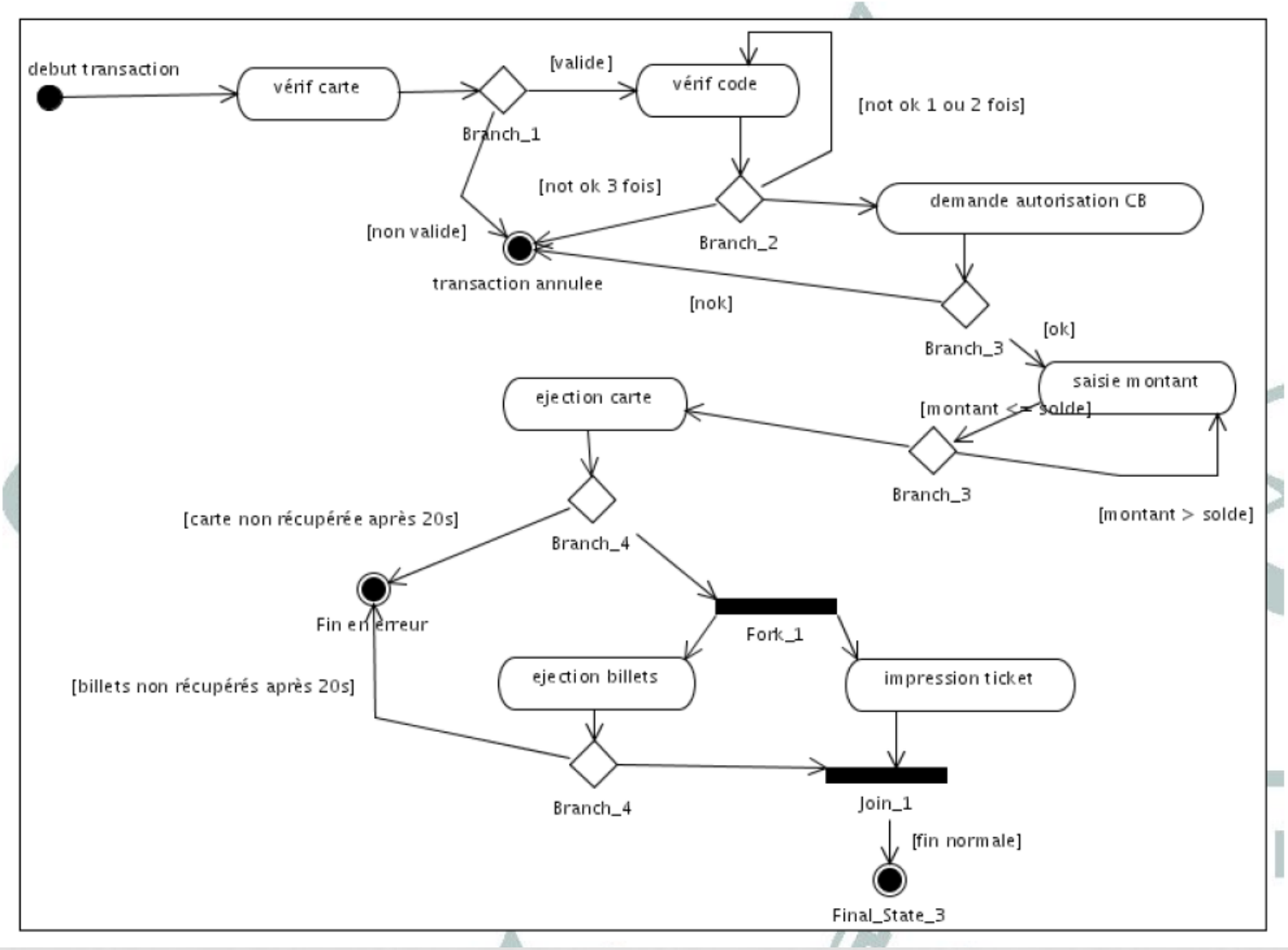


Diagramme de séquence système (DSEQ)

DIAGRAMME DE SÉQUENCE « SYSTÈME»

- **Documenter les interactions mis en œuvre entre les acteurs et le système (IHM) pour réaliser un cas d'utilisation**.

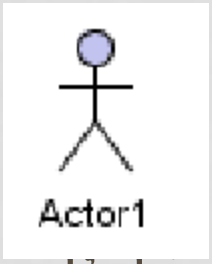
- Comme UML étant conçu pour la programmation orientée objet, ces communications sont reconnues comme des messages.

- Les acteurs et le système (IHM) sont énumérés en colonne, avec leurs lignes de vie verticales indiquant le cycle de vie d'un acteur ou système.

ELÉMENTS DE NOTATION

- **Acteurs** Entité virtuelle ou physique

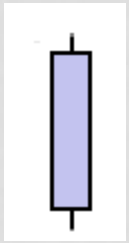
«Système»
Nom



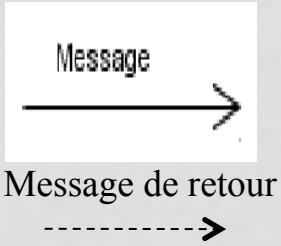
- **Ligne de vie**: identifie l'existence de l'objet par rapport au temps.

ELÉMENTS DE NOTATION

- **Activation**: indique quand un acteur effectue une action.

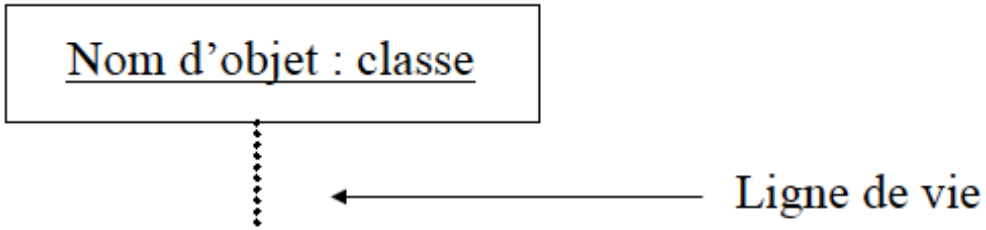


- **Message/Message de retour**: indique une communication entre les acteurs.

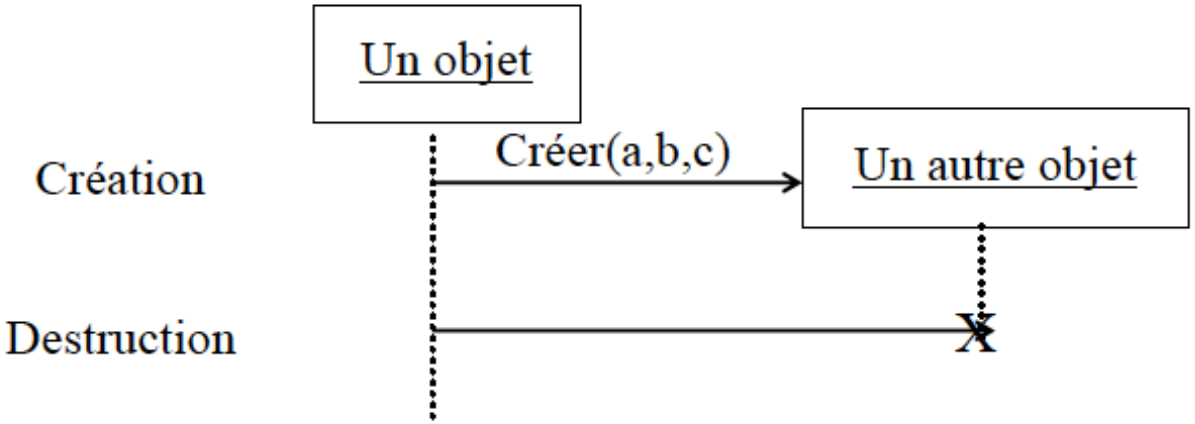


DSEQ représentant les interactions entre les objets

- La représentation des objets

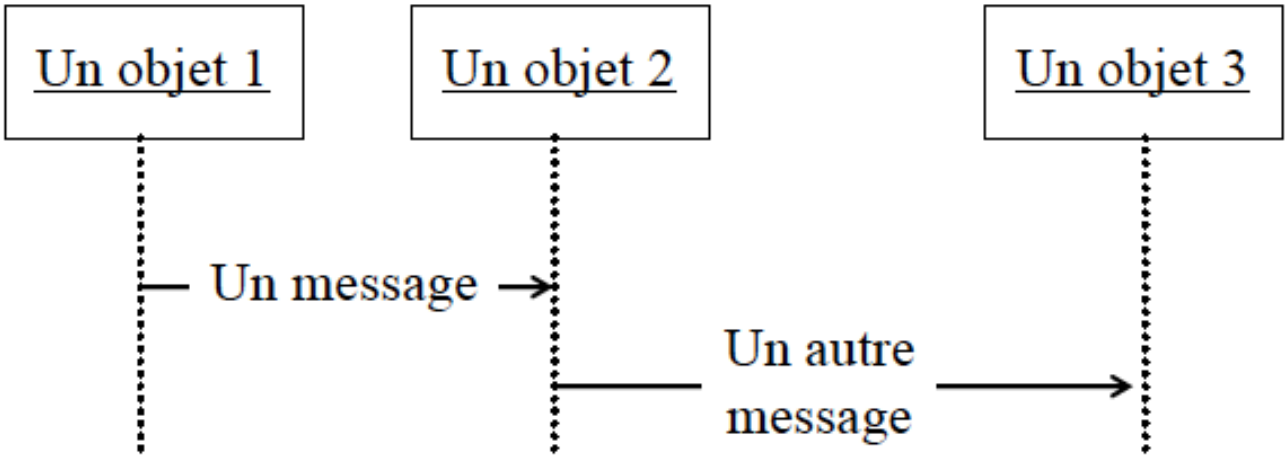


- La création et la destruction des objets



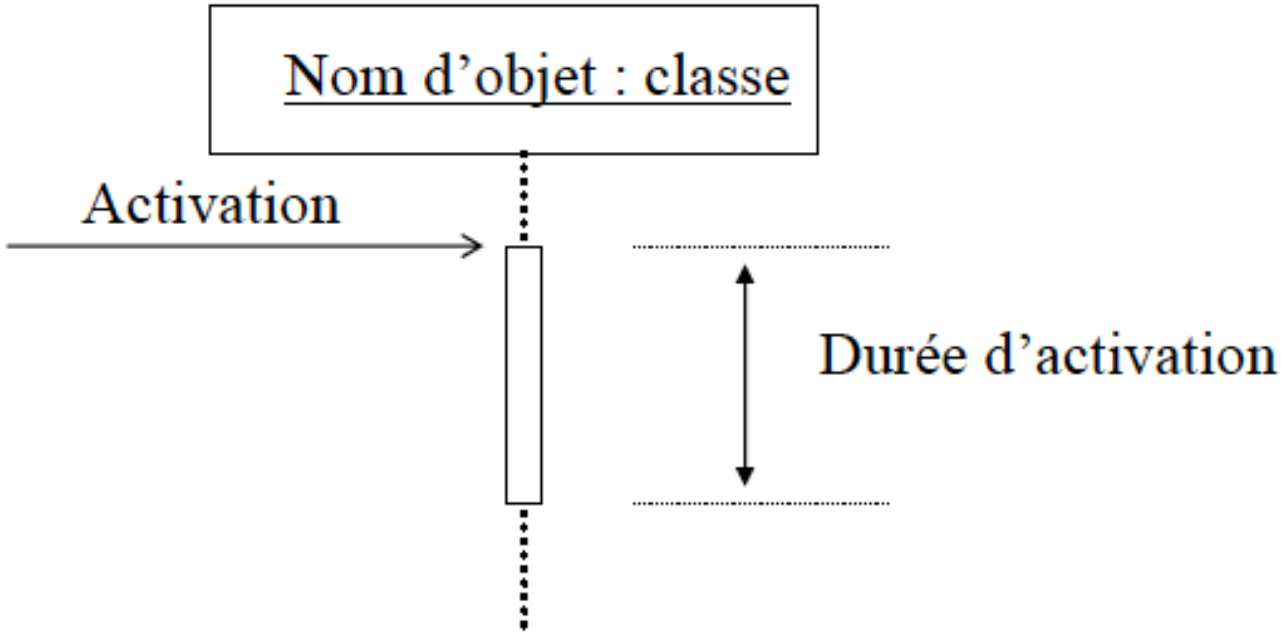
DSEQ représentant les interactions entre les objets

- La représentation des interactions



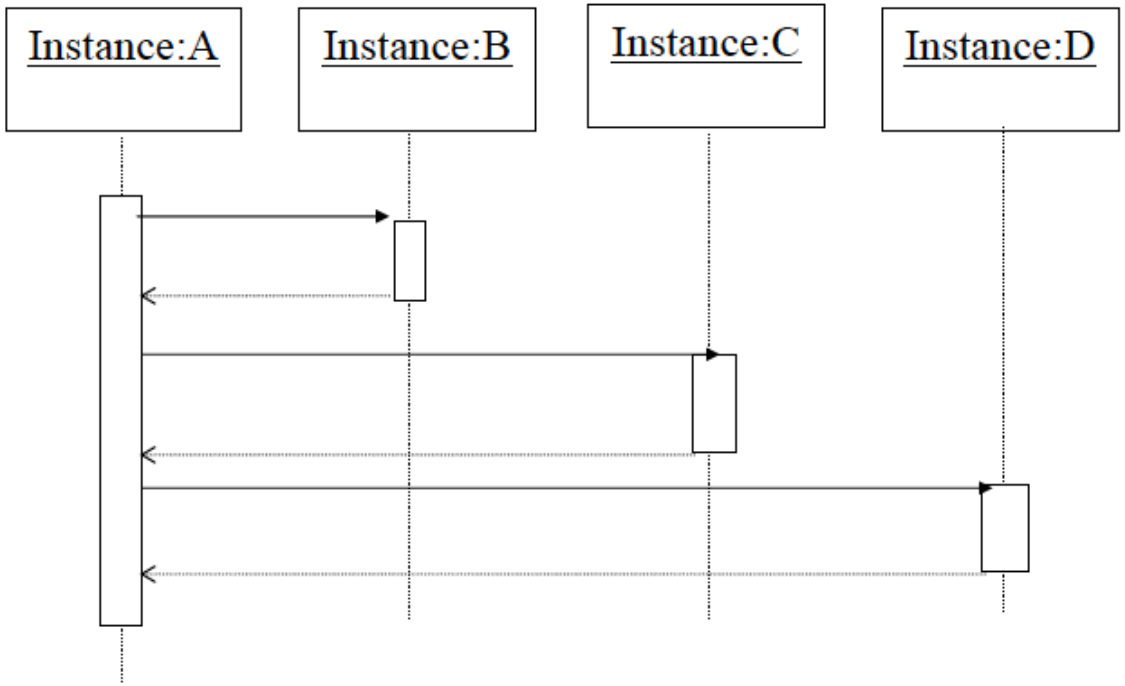
DSEQ représentant les interactions entre les objets

- La représentation des périodes d'activité des objets



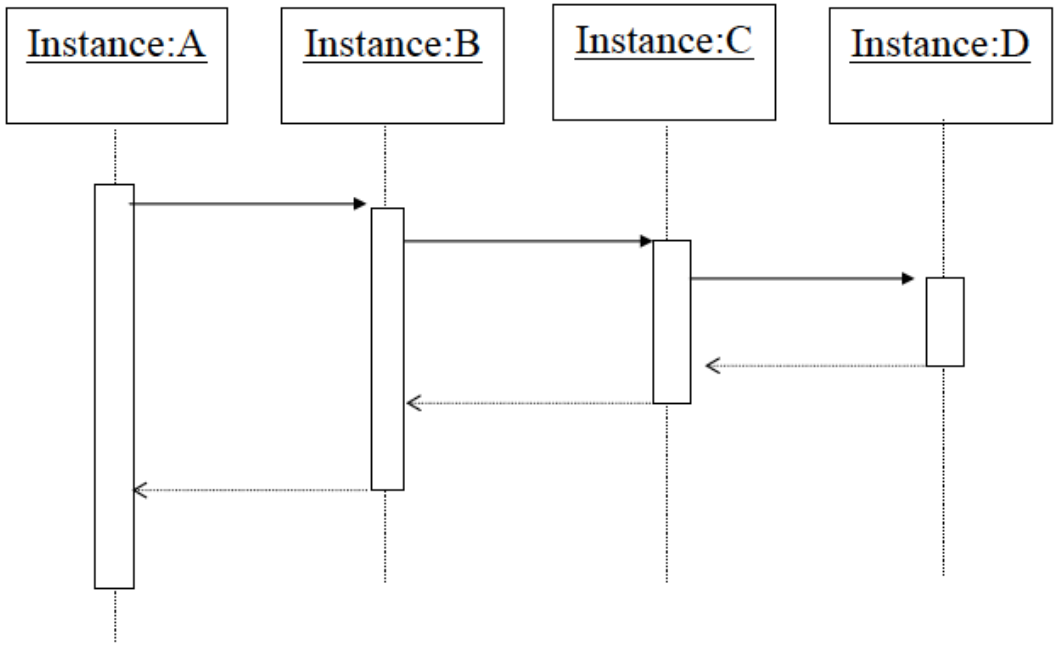
DSEQ représentant les interactions entre les objets

- Diagramme de séquence avec contrôle centralisé (en fourche)

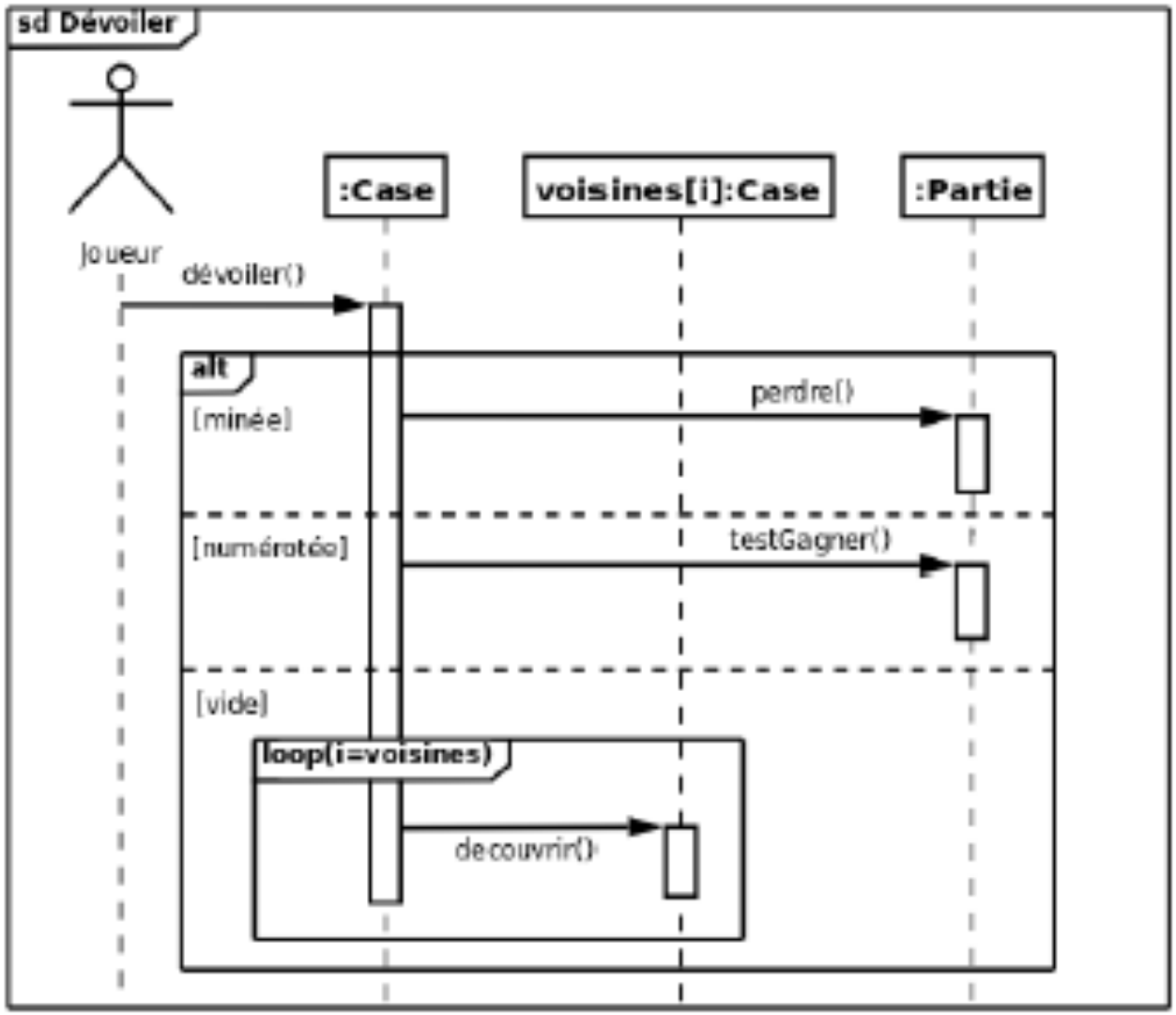


DSEQ représentant les interactions entre les objets

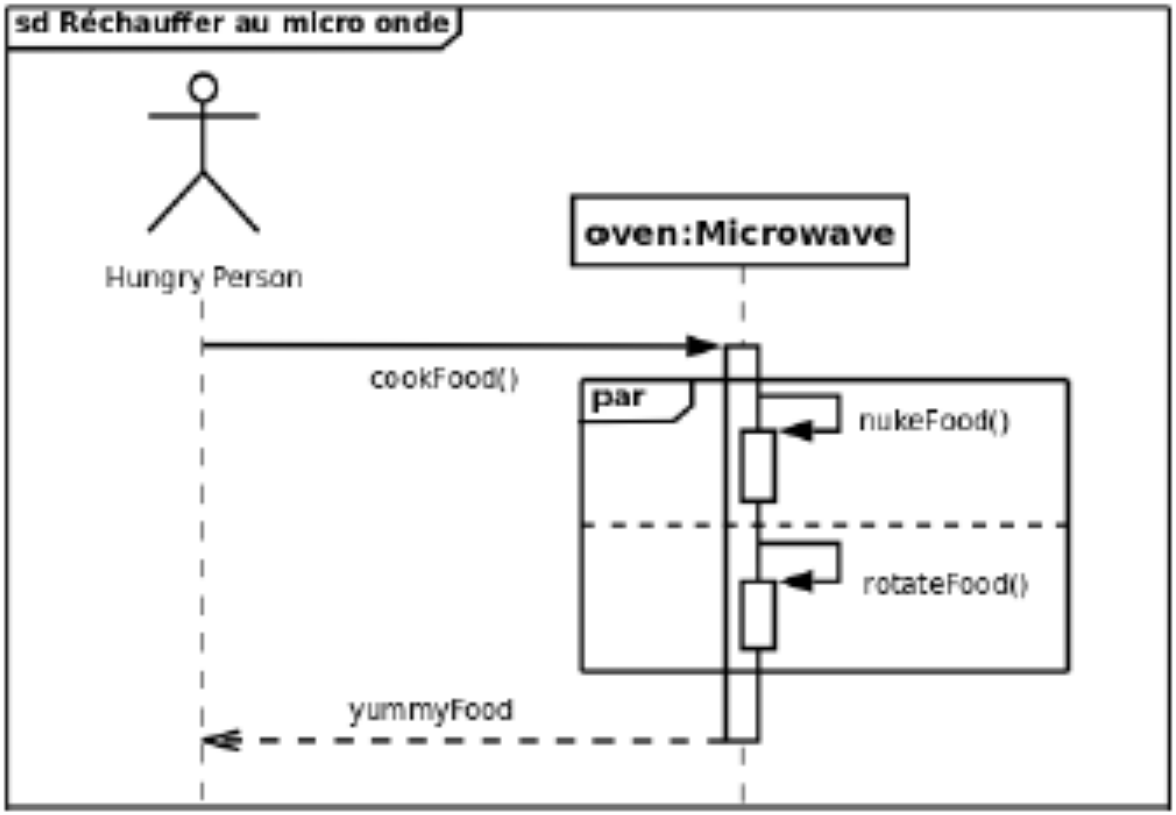
- Diagramme de séquence avec contrôle décentralisé (en escalier)



FRAGMENTS D'INTERACTION DANS DSEO

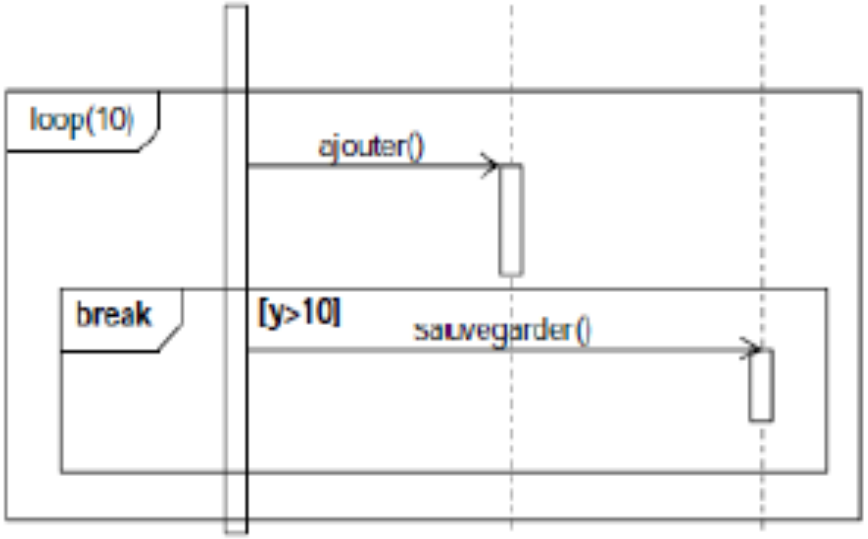


FRAGMENTS D'INTERACTION DANS DSEQ



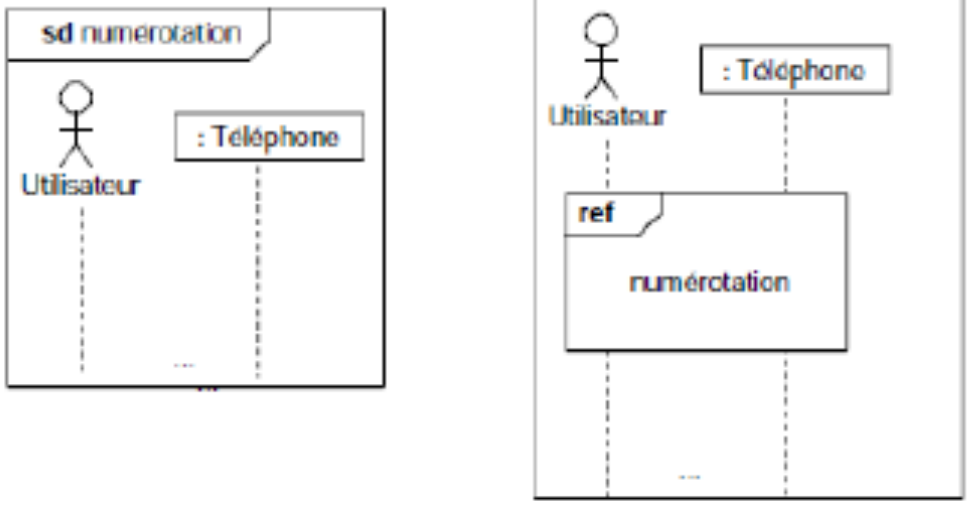
FRAGMENTS D'INTERACTION DANS DSEQ

- Interruption : `break`
  - Avec condition : exécuté lorsque la condition est VRAIE. Le reste du fragment d'interaction contenant (ex : loop) est ignoré



FRAGMENTS D'INTERACTION DANS DSEQ

- Référencement, réutilisation : `ref`
  - "Appel" à une interaction décrite dans un autre diagramme de séquence existant



DSEQ système « Retrait d'argent »

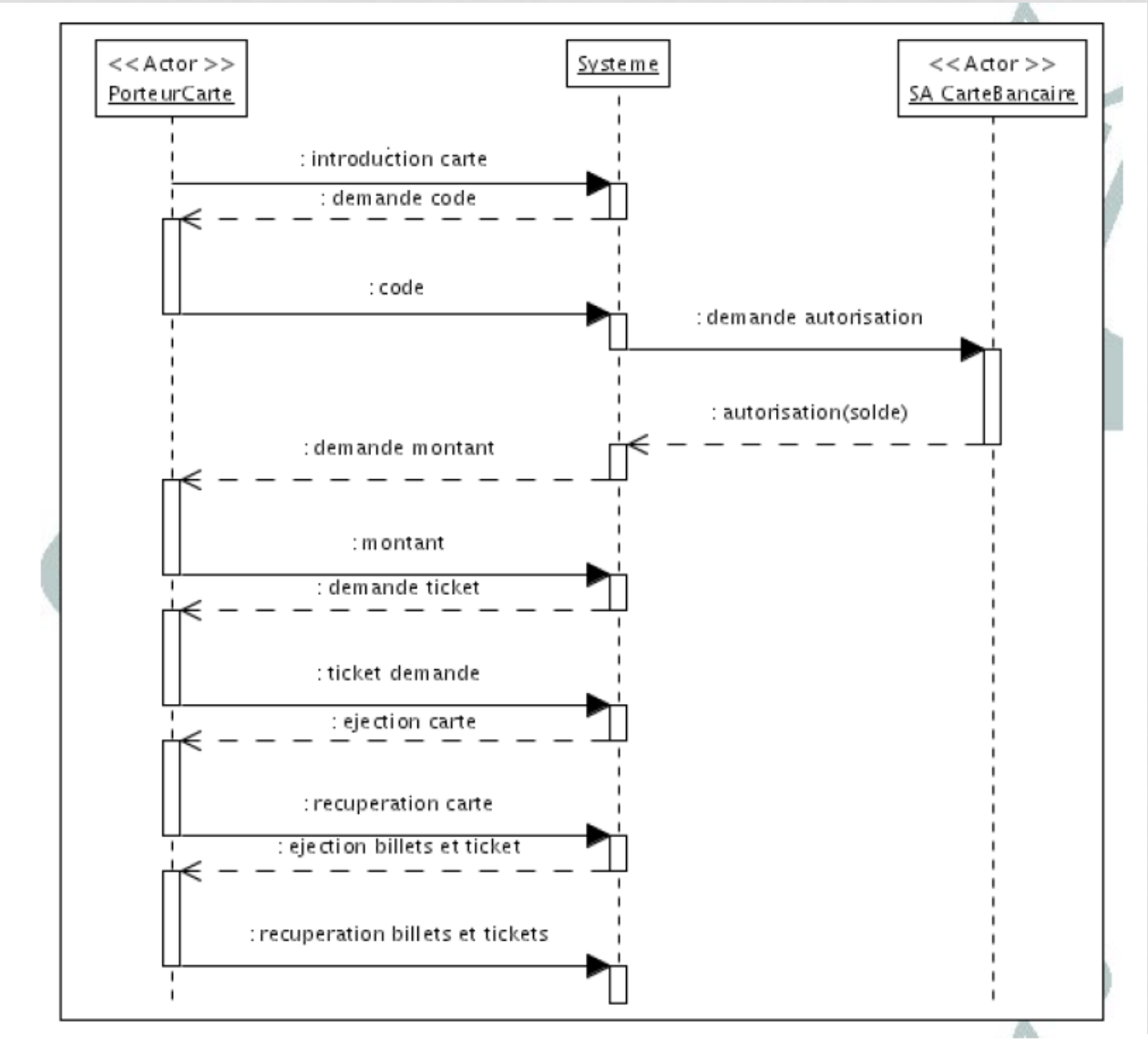


Diagramme état-transition (DET)

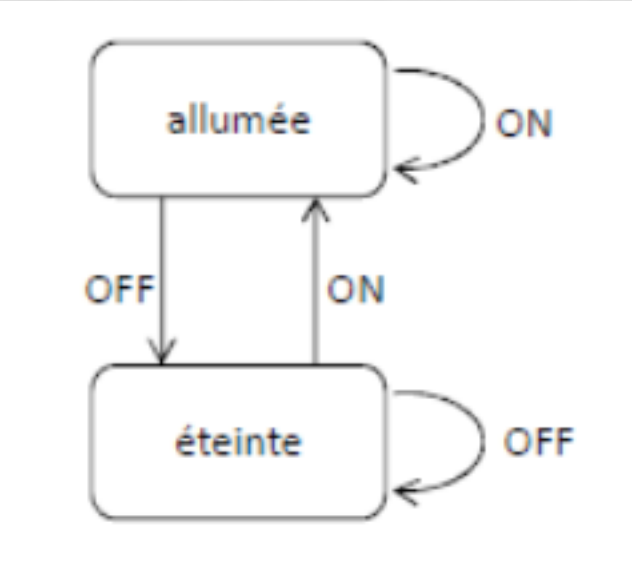
DÉFINITION

Un DET décrit le comportement des objets d'une classe au moyen d'un automate d'états associé à la classe

Un DET comporte:

* Nœuds = états possibles des objets
* Arcs = transitions d'état à état.
* Transition = soit exécution d'une action ou réaction de l'objet sous l'effet d'une occurrence d'événement ()

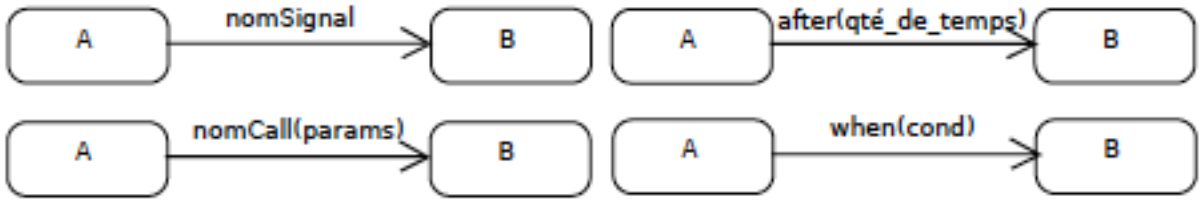
ELÉMENTS DE NOTATION



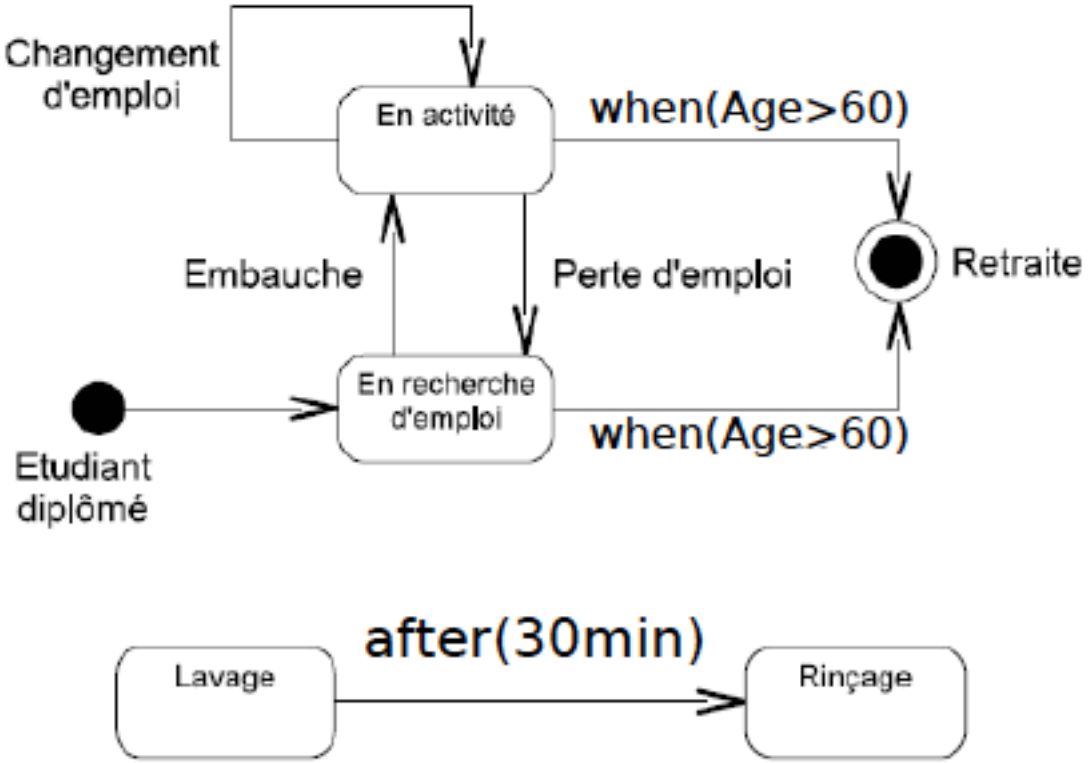
Eléments de notation

Différentes sortes d'événements

- *signal* réception d'un signal asynchrone, explicitement émis par un autre objet
- *call* appel de méthode sur l'objet courant (méthode déclarée dans le diagramme de classe).
- *after* causé par l'expiration d'une temporisation
- *change* causé par la satisfaction d'une condition booléenne
- *completion event* fin d'une activité liée à un état, de type do/ (déclenchement d'une transition "automatique", sans évènement déclencheur explicite)



EXEMPLE


Changement d'emploi
En activité
when(Age>60)
Embauche
Perte d'emploi
Retraite
En recherche d'emploi
Etudiant diplômé
when(Age>60)
after(30min)
Lavage
Rinçage

Comment compléter le DCL avec les opérations à partir du DSEQ et DET

DSEQ «RETIRERARGENT»

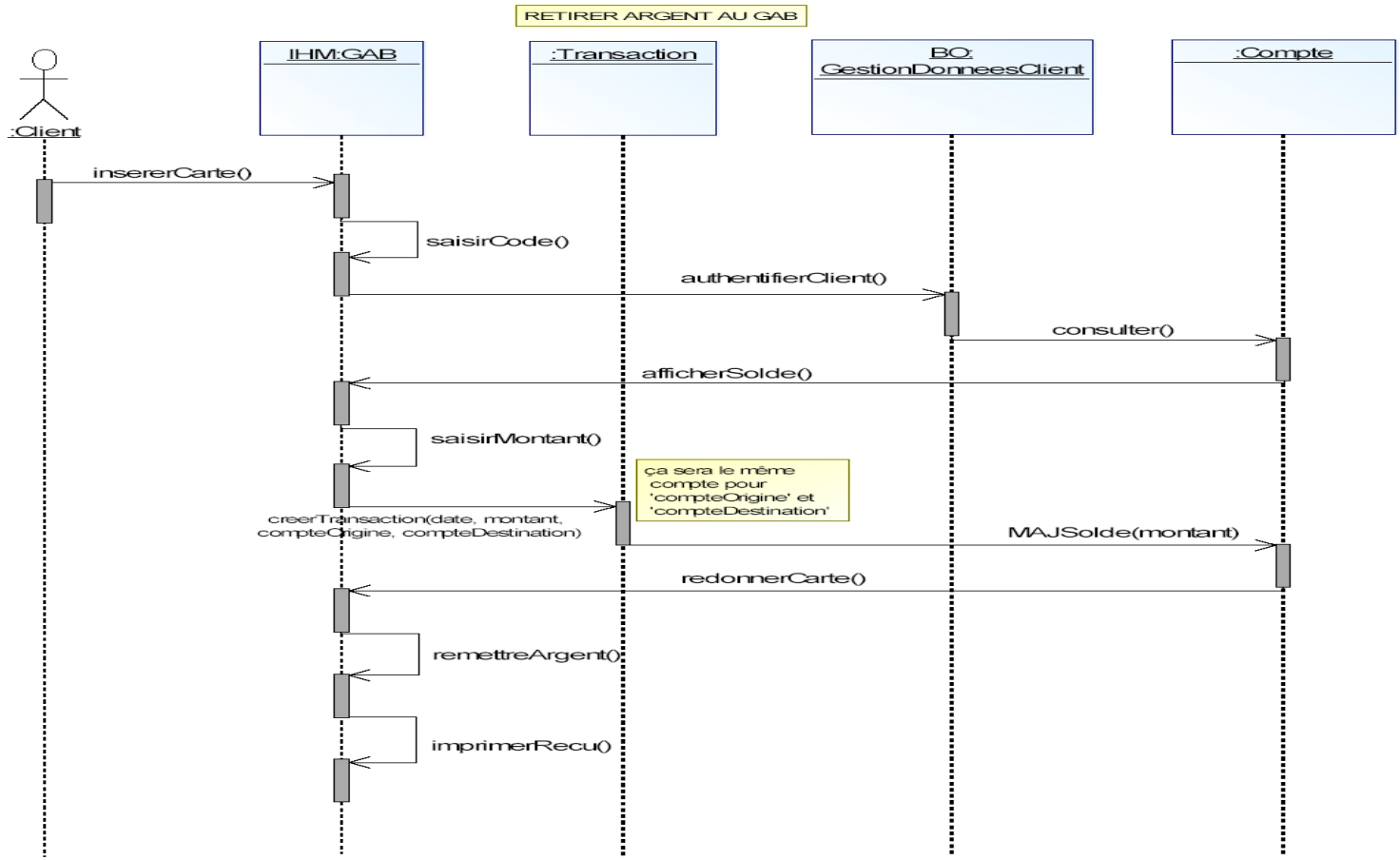
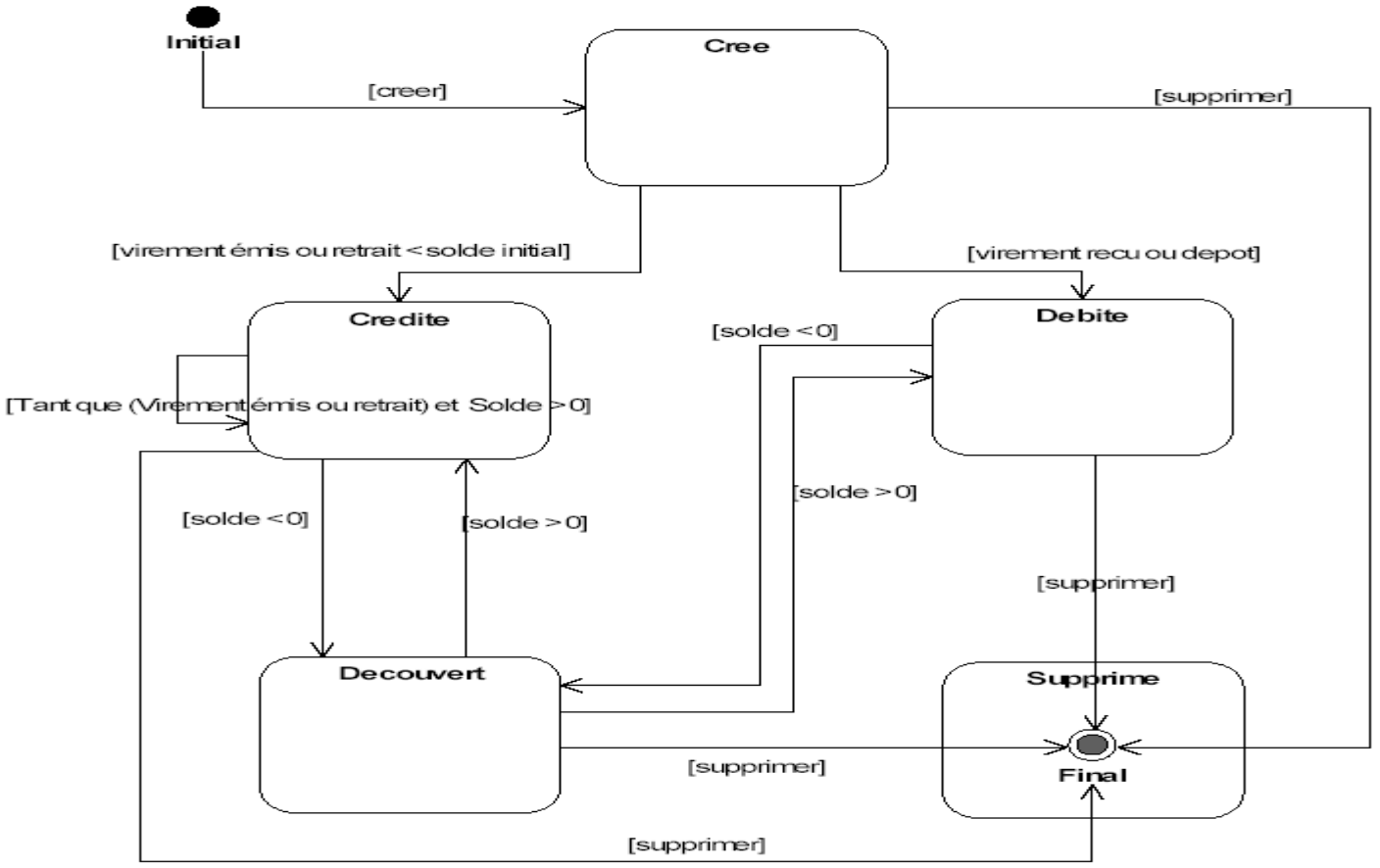
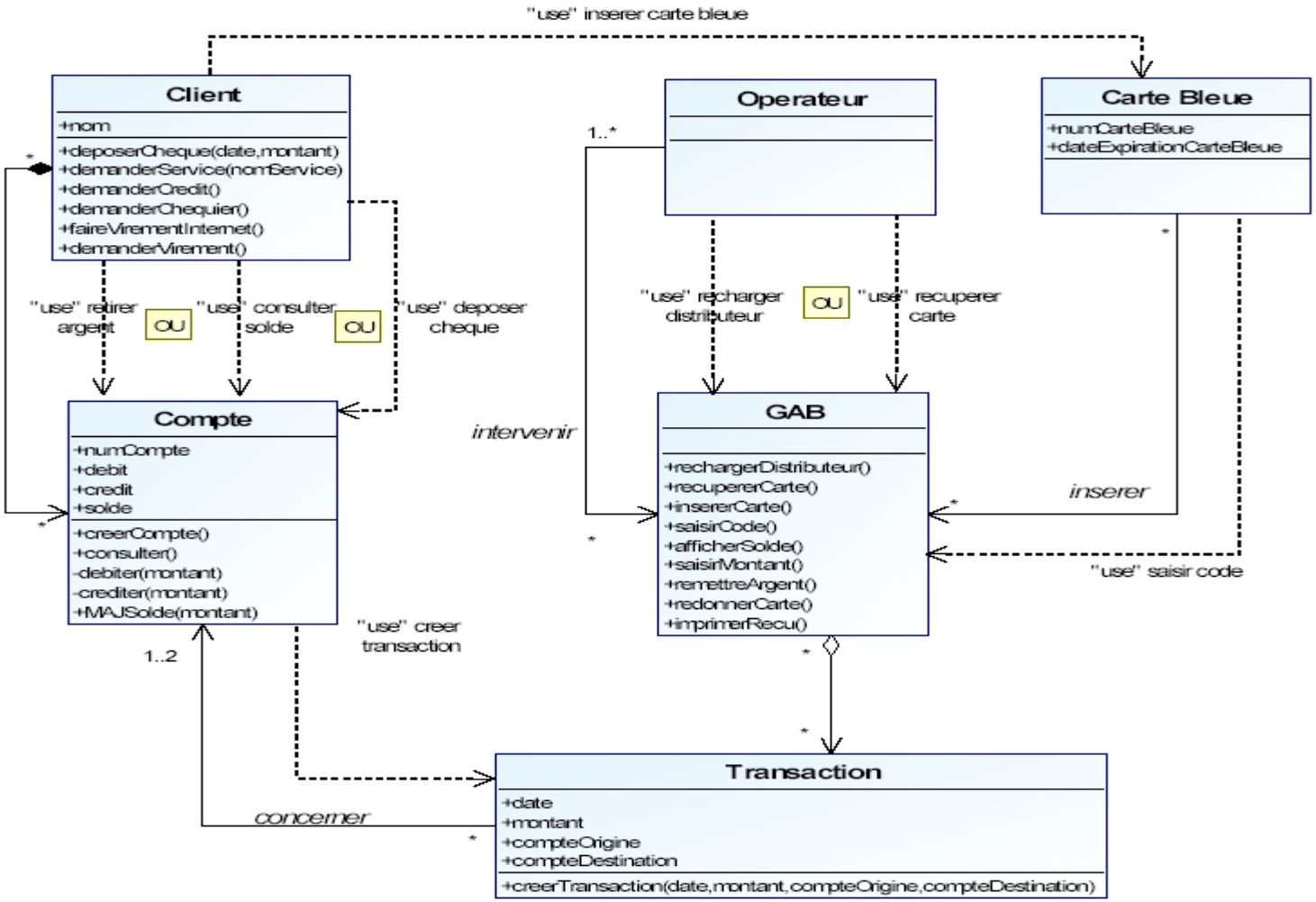


DIAGRAMME D'ÉTATS-TRANSITIONS D'UN COMPTE BANCAIRE



DIAGRAMMES DE CLASSE (AVEC LES MÉTHODES)



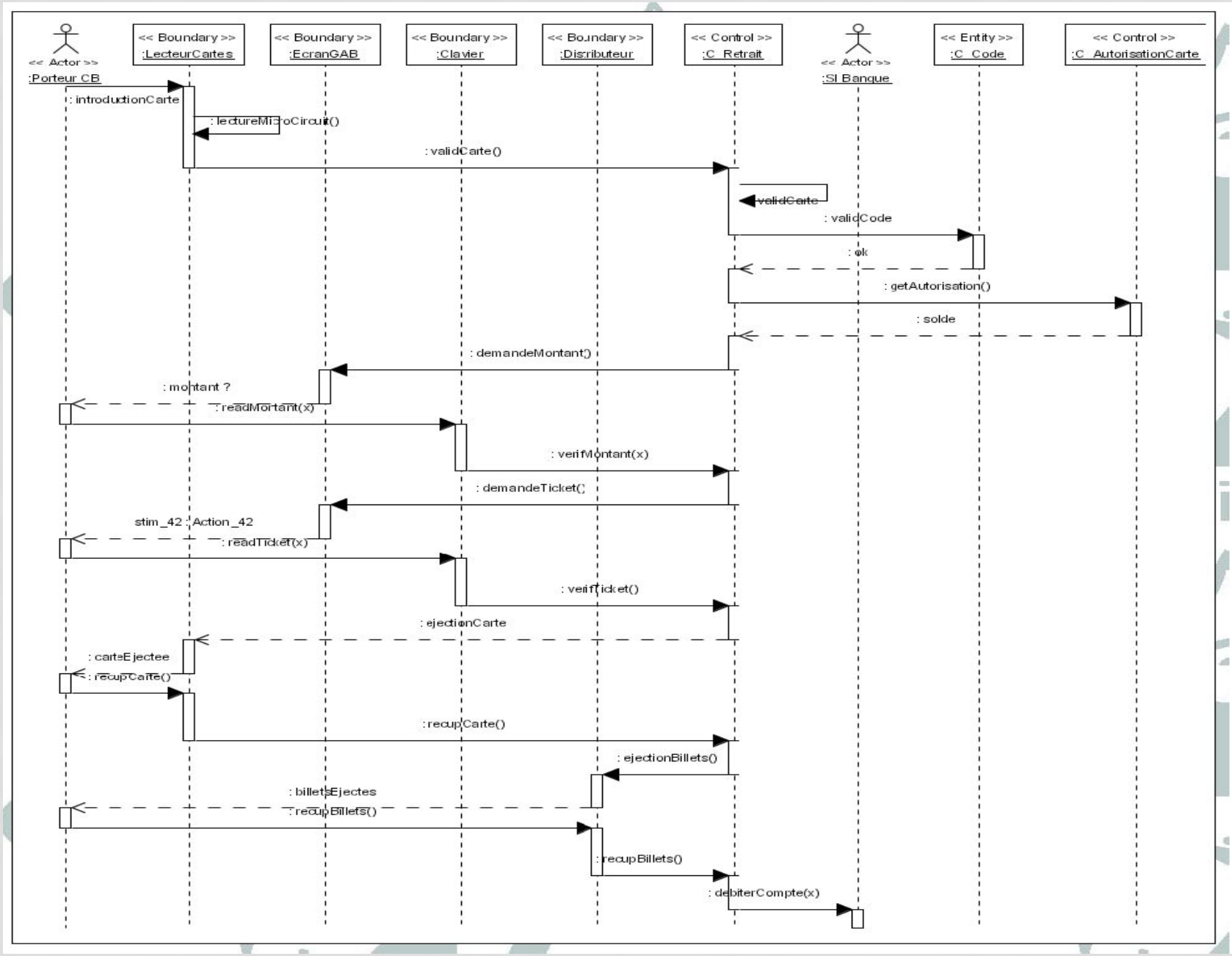
DSEQ avec des éléments de conception

PATTERN ENTITY-CONTROL-BOUNDARY (SIMPLIFICATION DE MVC)

- **Entities** are objects representing system data: Customer, Product, Transaction, Cart, etc.

- **Boundaries** are objects that interface with system actors: UserInterface, DataBaseGateway, ServerProxy, etc.

- **Controls** are objects that mediate between boundaries and entities. They orchestrate the execution of commands coming from the boundary by interacting with entity and boundary objects. Controls often correspond to use cases.


<< Actor >>
:Porteur CB
<< Boundary >>
:LecteurCartes
<< Boundary >>
:EcranGAB
<< Boundary >>
:Clavier
<< Boundary >>
:Distributeur
<< Control >>
:C_Retrait
<< Actor >>
:SI Banque
<< Entity >>
:C_Code
<< Control >>
:C_AutorisationCarte
: introductionCarte
: lectureMicroCircuit()
: validCarte()
: validCarte
: validCode
: ok
: getAutorisation()
: solde
: demandeMontant()
: montant ?
: readMontant(x)
: verifMontant(x)
: demandeTicket()
stim_42 : Action_42
: readTicket(x)
: verifTicket()
: ejectionCarte
: carteEjectee
: recupCarte()
: recupCarte()
: ejectionBillets()
: billetsEjectes
: recupBillets()
: recupBillets()
: debiterCompte(x)